UNIVERSITÉ DE MONTRÉAL

MODEL-CHECKING SYMBOLIQUE POUR LA VÉRIFICATION DE SYSTÈMES ET SON APPLICATION AUX TABLES DE DÉCISION ET AUX SYSTÈMES D'ÉDITIONS COLLABORATIVES DISTRIBUÉES

MANAL NAJEM
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION DU DIPLÔME DE
MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AOÛT 2009

© Manal Najem, 2009.

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

MODEL-CHECKING SYMBOLIQUE POUR LA VÉRIFICATION DE SYSTÈMES ET SON APPLICATION AUX TABLES DE DÉCISION ET AUX SYSTÈMES D'ÉDITIONS COLLABORATIVES DISTRIBUÉES

présenté par : Mme. NAJEM Manal.
en vue de l'obtention du diplôme de : Maîtrise ès Sciences Appliquées
a été dûment accepté par le jury constitué de :

M. GALINIER Philippe, Doct., président.
Mme. BOUCHENEB Hanifa, Doctorat, membre et directrice de recherche.
M. MERLO Ettore, Ph.D, membre et co-directeur de recherche.
M. ANTONIOL Giuliano, Ph.D., membre.

Remerciements

En préambule à ce mémoire, je souhaite adresser mes remerciements les plus sincères aux personnes qui m'ont apporté leur aide et qui ont contribué à l'élaboration de ce mémoire.

Je tiens à remercier sincèrement le professeur Hanifa Boucheneb, qui, à titre de directrice de projet, s'est toujours montrée à l'écoute et a été très disponible tout au long de la réalisation de ce travail. Son support, ses conseils ainsi que son inspiration ont été sans aucun doute une source continuelle de motivation tout au long du projet.

Mes remerciements s'adressent également au professeur Ettore Merlo qui, en tant que codirecteur de projet, a su faire preuve d'une grande patience malgré ses charges académiques et professionnelles et a su éclaircir mon jugement dans plusieurs étapes du projet.

Je désire aussi exprimer ma gratitude au corps professoral et au personnel de l'École Polytechnique de Montréal ainsi qu'à mes collègues de laboratoire.

Enfin, mais non moins important, j'adresse mes vifs remerciements à tous mes proches, amis et à ma famille qui m'ont toujours soutenue et encouragée au cours de la réalisation de ce mémoire. Plus particulièrement, j'aimerais exprimer ma gratitude et mon grand amour à mes parents Najem et Mariette, mon frère Nidal, mes deux sœurs Layal et Étoile ainsi qu'à ma chatte Tabby.

Merci à tous et à toutes.

Résumé

Dans le cycle de vie de tout système logiciel, une phase cruciale de formalisation et de validation au moyen de vérification et/ou de test induit une identification d'erreurs probables infiltrées durant sa conception. Cette détection d'erreurs et leur correction sont avantageuses dans les premières phases de développement du système afin d'éviter tout retour aux travaux ardus d'analyse de spécifications et de modélisation du système précédant sa réalisation. Par conséquent, cette étape mise en œuvre à travers des méthodes et des outils formels dans les phases amont de la conception contribue à augmenter la confiance des concepteurs et utilisateurs vis-à-vis de la fonctionnalité du système.

L'objectif de cette maîtrise s'insère dans le cadre d'une recherche qui vise à exploiter une technique formelle spécifique d'analyse de programmes et de spécifications : l'exécution symbolique combinée au model-checking. Cette technique représente une approche émergente à laquelle les chercheurs ont porté une attention particulière ces dernières années.

D'une part, l'exécution symbolique permet d'explorer les chemins d'exécution possibles d'un programme modélisant un système avec des variables d'entrée non initialisées, en d'autres termes en manipulant des variables abstraites ou "symboliques". Ces chemins caractérisent ainsi le comportement du programme de manière abstraite. D'autre part, le model-checking permet d'explorer systématiquement ces différents chemins d'exécution à l'aide d'une énumération exhaustive des états accessibles afin de générer ultérieurement des contre-exemples en cas de violation de propriétés du système.

De ce fait, l'exécution symbolique combinée au model-checking englobe les points forts de ces deux techniques octroyant aux concepteurs du système une compréhension accrue des situations d'erreur dans les contre-exemples ainsi générés.

Dans le présent mémoire, nous envisageons cette approche en utilisant l'extension

Symbolic JPF du model-checker logiciel Java PathFinder qui présente une maturité assez élevée dans le monde de vérification et de test. A l'aide de cet outil, nous procédons à la génération de contre-exemples sous forme de cas de test en entrée lors de la vérification des propriétés à la fois dépendantes et indépendantes de l'application et ce, dans deux champs d'application différents.

En effet, nous avons dans un premier temps vérifié des propriétés d'accessibilité et d'invariant dans notre première étude de cas : Un simulateur de vol modélisé formellement par des tables de décision spécifiant les relations entre les variables d'entrée du système ou les conditions et ses variables de sortie ou ses actions résultantes.

La représentation concise et rigoureuse des spécifications de ce système à travers les tables de décision comprenait néanmoins une hiérarchie à respecter : Les variables de sortie d'une table peuvent être des variables d'entrée d'une autre table. De ce fait, un ordre précis dans le calcul des variables intégrées dans ces tables a été exigé.

De plus, l'une des difficultés inhérentes à la vérification de ces systèmes émergeait de la qualité de calcul qui s'appuie sur des opérations arithmétiques et trigonométriques arbitraires et complexes. Plus spécifiquement, au sein de tels systèmes à domaines infinis, le model-checking se heurte au problème redoutable d'explosion d'états découlant de la nature exhaustive d'exploration des états lors de la vérification. Ici se manifeste l'importance de notre approche exploitée d'exécution symbolique combinée au model-checking qui nous a permis d'abstraire les variables lors de l'exécution permettant de ce fait l'identification des cas critiques violant des propriétés déterminées du système. Ces cas de violation sont identifiés en fonction des variables d'entrée du système. L'approche suivie dans cette étude de cas est originale dans le cadre de vérification de tables de décision modélisant un système avec des contraintes non linéaires.

Similairement, nous avons adopté la technique d'exécution symbolique combinée au model-checking dans notre seconde étude de cas : Un système d'éditions collaboratives distribuées caractérisé par une réplication d'un objet partagé entre plusieurs utilisateurs séparés physiquement mais interconnectés par un réseau pair-à-pair.

Dans ces systèmes, un objet partagé subit des modifications concurrentes et séquentielles de la part des utilisateurs sur chaque site à travers un échange d'opérations d'édition. De ce fait, une inconsistance dans ces systèmes peut résulter en une divergence dans les contenus des documents répliqués entre les différents utilisateurs,

aspect non désiré dans ces systèmes dont le but essentiel est d'assurer une interaction collaborative tout en conservant des copies convergentes sur chaque site. Cette convergence est prétendument assurée par l'application, dans des conditions données, de transformées opérationnelles moyennant des algorithmes d'intégration précis.

Dans le cadre de notre travail, nous avons obtenu par le biais des cas de test en entrée, générés à partir de notre approche, des scénarios complets de divergence et ce, pour les cinq algorithmes envisagés. L'ensemble complet de tous les contre-exemples trouvés pour chaque algorithme constitue l'un de nos principaux apports au sein de ces systèmes où la vérification formelle reste limitée. Une autre contribution d'autant plus importante au niveau de notre travail dans cette étape réside dans l'intégration d'une abstraction à la description du modèle antérieur. Cette initiative nous a permis de gagner à la fois en temps d'exécution et en mémoire relativement au modèle concret.

Dans ce mémoire, nous nous sommes limités dans la seconde étude de cas à identifier les scénarios de divergence pour chaque algorithme vérifié. Il serait intéressant de s'appuyer sur ces scénarios afin d'apporter des corrections au niveau de chacun de ces algorithmes.

Abstract

Verification is one crucial activity in any software life cycle. Its major role is to ensure an identification of potential design and implementation flaws integrated in the software system during its development process. Such an identification leads to eventual corrections in the early steps of the development cycle, thus avoiding tedious work otherwise required in the system requirements' reanalysis as well as in its remodelling preceding its deployment. As a consequence, the verification step is rigorously put into practice through formal methods and tools. Given such a formalisation contributes to give another level of insurance to both the system's designers and users.

This thesis is related to a research which aims at applying one specific formal method in program and requirements analysis: symbolic execution intertwined with model checking. This technique has known a major development in the past few years, thus raising interest among researchers in the field.

On one hand, symbolic execution explores all possible execution paths of a program modelling a system using uninitialised input variables. As its name implies, this specific execution deals with abstract or "symbolic" variables. Hence, those visited paths characterise the abstract program behaviour. On another hand, model checking ensures a systematic exploration of those different execution paths through an exhaustive visit of all reachable states. This approach is necessary for subsequent generation of counterexamples in case of property violations within the system.

Therefore, symbolic execution along with model checking is a resulting approach enforced with advantages of both techniques. This yields a higher degree of interpreting the retrieved flaws provided through generated counterexamples, for even the most sophisticated systems.

In the present work, we apply this technique using the extension Symbolic JPF of the software model checker Java PathFinder. Our choice relies on the high level of maturity of the tool in the verification of a wide range of applications. Thus, using

this tool, we direct our verification purposes on two different applications albeit both end up in one unique outcome: counterexamples representing input test cases refuting some property. It is to be noted that the verification process handles several classes of properties ranging from application-dependent to application-independent properties.

In fact, our first goal is to verify reachability and invariance properties in the context of a flight simulator especially modelled in a well-known formal tabular notation—decision tables. These tables concisely point out the relations between system input variables or conditions and system output variables corresponding to the resulting actions.

However, those encountered decision tables had to respect some hierarchy: One table's output variables could be fed as input variables to other tables. Accordingly, we were compelled to follow a specific computation order imposed by the tables' structure.

Moreover, one of the challenges encountered during the verification of such systems resides in its computation process which includes random and complex arithmetic and trigonometric operations. More specifically, in the context of such systems with infinite domains, model checking is constrained with the state explosion problem due to the exhaustive state exploration in the verification of property violations. Taking into account the latter limitation, our combined approach of symbolic execution with model checking allows the abstraction of computed variables, thus drastically reducing the number of explored states. As a result, the identification of critical cases violating special system properties is made possible. Those generated cases identify the specific combinations of system input variables disproving the verified property. The applied approach in our case study framework is original in the verification of decision tables modelling a system governed by non-linear constraints.

Similarly, we target our considered approach on our second case study: A replication-based collaborative editing system where a common object is shared between several users physically separated but connected via a peer-to-peer network.

In such systems, users on each site could modify the shared object by some exchange of editing operations in concurrent and consecutive fashions. As a consequence, a divergence in the contents of the replicas between different users is possible leading to the system inconsistence. This contradicts with the main purpose of such

systems that originally allows a collaborative interaction between users while preserving the copies convergence in each site. This convergence is allegedly ensured by the application of operational transformations, under special conditions, through well-defined integration algorithms.

In our case study framework, the applied approach of symbolic execution combined with model-checking resulted in generating input test cases illustrating the divergence scenarios for all five considered algorithms. Since formally verifying such systems is still limited, we could say that our approach is original in collecting all retrieved counterexamples violating the convergence property. Moreover, another major contribution in our work was in the use of further abstractions in the description of the previous system model. Hence, using this abstract model, we gained in both execution time and memory space relatively to the previous approach.

Finally, It should be noted that we restrained our work, in the second case study of the thesis, to solely identifying divergence scenarios for each verified algorithm. It would be interesting to extend our work to rectifying those retrieved flaws after their analysis thus overcoming the hard task of designing new algorithms.

Table des matières

Remerciements iii

Résumé iv

Abstract vii

Table des matières x

Liste des tableaux xiii

Liste des figures xiv

Liste des sigles et abréviations xviii

Chapitre 1 INTRODUCTION 1
1.1 Définitions et concepts de base 2
1.2 Éléments de la problématique 4
1.3 Objectifs de recherche 6
1.4 Plan du mémoire 6

Chapitre 2 REVUE DE LITTÉRATURE 8
Première partie : État de l'art 9
2.1 Model-checking 11
2.2 Model-checking symbolique 13
2.2.1 Model-checking symbolique basé sur les BDDs 14
2.2.2 Model-checking symbolique sans les BDDs 23
2.3 Model-checking : de l'analyse à la génération de tests 25
2.4 Exécution symbolique 29
2.4.1 Définition 30
2.4.2 Intégration de l'exécution symbolique dans un langage de programmation 30
2.4.3 Arbre d'exécution symbolique 34

2.4.4 Intégration de l'exécution symbolique dans le model-checking et le test 37
2.4.5 Abstractions combinées à l'exécution symbolique 40
Seconde partie : Présentation des outils 43
2.5 NuSMV 43
2.6 Alloy Analyzer 45
2.7 JForge 46
2.8 Java PathFinder et ses extensions : JPF-SE et "Symbolic JPF" . . . 48
2.8.1 Préliminaires 48
2.8.2 Que vérifie JPF ? 50
2.8.3 Implémentation des propriétés 50
2.8.4 Model-checking avec JPF 51
2.8.5 Avantages du model-checking avec JPF vis-à-vis du test 52
2.8.6 Structure haut niveau de JPF 54
2.8.7 Instrumentation des applications avec "Verify" 55
2.8.8 Extensibilité de JPF 55
2.8.9 Extensions de JPF : JPF-SE puis "Symbolic JPF" 58

Chapitre 3 PREMIÈRE ÉTUDE DE CAS : SIMULATEUR DE VOL 62
3.1 Tables de décision 63
3.1.1 Définition 63
3.1.2 Avantages de la notation tabulaire 66
3.1.3 Traitement des tables de décision 67
3.2 Modèle de notre étude de cas 68
3.3 Exemple illustratif dans le contexte d'industrie aéronautique 70
3.3.1 De l'analyse syntaxique à la génération de code 70
3.3.2 Expérimentations avec NuSMV 76
3.3.3 Expérimentations avec Alloy Analyzer 79
3.4 Étude de cas : Simulateur de vol 85
3.4.1 Présentation de notre étude de cas 86
3.4.2 Implémentation des propriétés à vérifier 88
3.4.3 Expérimentations avec JForge 90
3.4.4 Expérimentations avec JPF et son extension Symbolic JPF . . 97

Chapitre 4 SECONDE ÉTUDE DE CAS : SYSTÈME D'ÉDITIONS COLLABORATIVES DISTRIBUÉES . . . 114
4.1 Transformées opérationnelles et algorithmes OT . . . 116
4.1.1 Définition du concept de transformées opérationnelles . . . 117
4.1.2 État de l'art des algorithmes OT . . . 120
4.2 Élaboration du modèle concret . . . 132
4.2.1 Notions de base . . . 133
4.2.2 Motivations . . . 134
4.2.3 Description des constantes et variables d'entrée . . . 137
4.2.4 Comportements des sites . . . 142
4.2.5 Intégration du modèle concret pour son exécution symbolique 148
4.2.6 Vérification de la propriété de convergence . . . 154
4.2.7 Limitations du modèle concret . . . 158
4.3 Abstractions dans le modèle concret : Modèle abstrait . . . 159
4.3.1 Intégration des matrices de bornes DBMs dans le modèle . . . 160
4.3.2 Vérification de la consistance du modèle . . . 176

Chapitre 5 CONCLUSION . . . 180
5.1 Synthèse des travaux . . . 180
5.2 Limitations de la solution proposée . . . 182
5.3 Avenues futures . . . 183

Références . . . 184

Liste des tableaux

TABLEAU 3.1 Association entre les composantes des tables de décision et l'arbre syntaxique généré 74
TABLEAU 3.2 Évaluation des cas de test obtenus 111
TABLEAU 4.1 Résultats obtenus dans le cadre d'une approche basée sur les démonstrateurs de théorèmes 154
TABLEAU 4.2 Résultats obtenus dans le cadre de notre travail pour *NbSites = 2* et *MaxIter = 2* 155
TABLEAU 4.3 Résultats obtenus dans le cadre de notre travail pour *NbSites = 2* et *MaxIter = 3* 156
TABLEAU 4.4 Résultats obtenus pour *NbSites = 3* et *MaxIter = 3* 157
TABLEAU 4.5 Résultats obtenus pour *NbSites = 3* et *MaxIter = 4* 157

Liste des figures

FIGURE 1.1 Vérification d'un système par model-checking 3

FIGURE 2.1 Diagramme de décision binaire réduit représentant la fonction $f(x;y) = \overline{x+y}$ 15

FIGURE 2.2 Processus de model-checking symbolique (Source [2]) 16

FIGURE 2.3 Programme simple "Calc" à opérations arithmétiques linéaires 34

FIGURE 2.4 Arbre d'exécution correspondant au programme "Calc" 35

FIGURE 2.5 Exécution symbolique et model-checking (Source [16]) 39

FIGURE 2.6 Architecture du système NuSMV (source [19]) 44

FIGURE 2.7 Fonctionnalité générale de JForge 47

FIGURE 2.8 Fonctionnalité générale de Java PathFinder (Source [44]) . . . 49

FIGURE 2.9 Interface "SearchListener" 56

FIGURE 2.10 Interface "VMListener" 57

FIGURE 2.11 Architecture de JPF-SE (Source [25]) 59

FIGURE 3.1 Table de décision décrivant le phénomène de "Depressurization Hazard" 64

FIGURE 3.2 Table de décision représentant l'action CONTROLVALVEOPENED et le format XML équivalent 65

FIGURE 3.3 Processus de vérification des tables de décision 69

FIGURE 3.4 Analyse syntaxique et lexicale des tables de décision avec JAVACC 72

FIGURE 3.5 Arbre syntaxique et lexical généré 73

FIGURE 3.6 Extrait de notre visiteur pour la construction de *IVarBuffer* et *IValueBuffer* 75

FIGURE 3.7 Récupération des variables après la visite de l'AST 76

FIGURE 3.8 Code SMV généré pour la table de décision CONTROLVALVEOPENED 77

FIGURE 3.9 Trace d'exécution NUSMV 78

FIGURE 3.10 Contre-exemple obtenu dans NUSMV 78

FIGURE 3.11 Contre-exemple simulant une erreur lors de l'élaboration des requis dans NUSMV 79

FIGURE 3.12 Entête du code ALLOY généré 80

FIGURE 3.13 Représentation d'une transition dans ALLOY 82
FIGURE 3.14 Représentation des prédicats et des assertions dans ALLOY . . 83
FIGURE 3.15 Vérification de la complétude de notre système dans ALLOY . 84
FIGURE 3.16 Contre-exemple dans ALLOY 84
FIGURE 3.17 Tables de décision modélisant le simulateur de vol 85
FIGURE 3.18 Variables et opérations caractérisant le simulateur de vol . . . 88
FIGURE 3.19 Implémentation de l'arithmétique d'intervalles en Java 95
FIGURE 3.20 Entête du code Java 95
FIGURE 3.21 Annotation JFSL pour la vérification du code Java 96
FIGURE 3.22 Fichier d'entrée "DTInputFile" correspondant aux préconditions des variables d'entrée 101
FIGURE 3.23 Extraction de l'information pour la génération des préconditions 101
FIGURE 3.24 Génération de la classe mère dans l'exécution symbolique . . . 102
FIGURE 3.25 Fichier d'entrée "ConditionInputFile" correspondant aux variables et leurs conditions 103
FIGURE 3.26 Extraction de l'information pour la génération de l'entête du code 104
FIGURE 3.27 Entête du code modélisant notre système 105
FIGURE 3.28 Fichier d'entrée "DTFile" correspondant au calcul des tables de décision 106
FIGURE 3.29 Code modélisant nos tables de décision 107
FIGURE 3.30 Configuration dans SYMBOLIC JPF 108
FIGURE 3.31 Filtrage des cas de test pertinents 110
FIGURE 3.32 Un cas de test correspondant à la propriété $[FPM_az==_unknown_]$ 112
FIGURE 4.1 Divergence des copies due à l'absence de transformées opérationnelles 121
FIGURE 4.2 Convergence des copies après les transformations avec intégration 122
FIGURE 4.3 Algorithme IT d'Ellis (Source [34]) 126
FIGURE 4.4 Algorithme IT de Ressel (Source [34]) 127
FIGURE 4.5 Algorithme IT de Sun (Source [34]) 128
FIGURE 4.6 Algorithme IT de Suleiman (Source [34]) 129
FIGURE 4.7 Algorithme IT d'Imine (Source [34]) 131
FIGURE 4.8 Méthode "*Concurrent*" implémentant la relation de concurrence entre deux opérations 142

FIGURE 4.9 Méthode *"garde"* testant la possibilité d'exécution d'une opération . . . 143
FIGURE 4.10 Méthode "*Transformation*" déclenchant le processus effectif de transformation des algorithmes IT . . . 144
FIGURE 4.11 Méthode "*Operation*" exécutant l'opération sur le texte partagé 145
FIGURE 4.12 Situation de divergence mettant en relief le problème de concurrence partielle . . . 146
FIGURE 4.13 Solution au problème de concurrence partielle . . . 147
FIGURE 4.14 Modèle des méthodes internes à la classe *Site* . . . 148
FIGURE 4.15 Propriété de convergence implémentée pour l'exécution symbolique . . . 150
FIGURE 4.16 Extrait d'un cas de test en entrée . . . 153
FIGURE 4.17 Scénario de divergence avec l'algorithme de Sun *et al.* pour *MaxIter = 2* . . . 155
FIGURE 4.18 Scénario de divergence avec l'algorithme de Sun *et al.* pour *MaxIter = 3* . . . 156
FIGURE 4.19 Scénario de divergence avec l'algorithme de Ressel *et al.* pour *NbSites = 3* et *MaxIter = 3* . . . 157
FIGURE 4.20 Scénario de divergence avec l'algorithme d'Imine *et al.* pour *NbSites = 3* et *MaxIter = 4* . . . 158
FIGURE 4.21 Format général d'une matrice de bornes . . . 164
FIGURE 4.22 Implémentation de la matrice de bornes initiale correspondant à l'algorithme d'Ellis . . . 165
FIGURE 4.23 Implémentation de la matrice de bornes initiale correspondant à l'algorithme de Sun . . . 167
FIGURE 4.24 Implémentation de la matrice de bornes initiale correspondant à l'algorithme de Suleiman . . . 168
FIGURE 4.25 justification=raggedright . . . 170
FIGURE 4.26 Procédure générale des transformées opérationnelles à partir d'une matrice initiale M_0 . . . 171
FIGURE 4.27 Matrice transformée M_1 correspondant au cas $(p_1 < p_2)$ . . . 172
FIGURE 4.28 Matrice transformée M_2 correspondant au cas $(p_1 = p_2)$ . . . 173
FIGURE 4.29 Matrice transformée M_3 correspondant au cas $(p_2 < p_1)$ . . . 173

FIGURE 4.30 Réordonnement des opérations pour leur transformation dans *Site 2* 176
FIGURE 4.31 Pseudo-code associé à l'algorithme de Floyd-Warshall 177
FIGURE 4.32 Contre-exemple abstrait avec l'algorithme de Sun *et al.* pour *3 Ins* 178

Liste des sigles et abréviations

SCR	Software Cost Reduction
LTL	Linear Temporal Logic
CTL	Computation Tree Logic
SMV	Symbolic Model Verifier
BDD	Binary Decision Diagram
RSML	Requirements State Machine Language
BMD	Binary Moment Diagram
HDD	Hybrid Decision Diagram
RFF	Reduction Finite Focus
JML	Java Modelling Language
JFSL	JForge Specification Language
FIR	Forge Intermediate Representation
BFS	Breadth-First Search
DFS	Depth-First Search
MJI	Model Java Interface
POR	Partial Order Reduction
AST	Abstract Syntax Tree
JPF	Java PathFinder
VM	Virtual Machine
P2P	Peer-to-Peer
OT	Operational Transformation
TP	Transformation Property
IT	InTegration
DBM	Difference-Bound Matrix

Chapitre 1

INTRODUCTION

Le cycle de vie de tout système logiciel désigne un ensemble interdépendant d'étapes de développement, depuis sa conception jusqu'à sa qualification puis sa mise en production et finalement sa maintenance. L'objectif d'une telle catégorisation est de permettre de définir des jalons intermédiaires permettant la validation du développement logiciel—la conformité du logiciel avec les besoins exprimés— et la vérification du processus de développement—la concordance des méthodes mises en œuvre.

La vérification fait implicitement partie des étapes d'intégration et de qualification dans ce cycle de vie. Le but crucial de cette technique consiste en une détection des erreurs au plus tôt et en une maîtrise accrue de la qualité du logiciel, des délais de sa réalisation et des coûts associés.

Dans ce contexte, on comprend aisément qu'il est vital que la vérification repose sur des méthodes rigoureuses qui garantissent la conformité des systèmes vis-à-vis de leurs requis et de leurs fonctionnalités désirées. Il faudrait en effet modéliser le comportement des systèmes pour ensuite procéder à leur analyse, à leur vérification d'une manière formelle et éventuellement à l'identification de leurs disfonctionnements potentiels lors des phases amont de la conception.

Dans ce chapitre d'introduction, nous présenterons les concepts de base à partir desquels nous élaborerons les éléments de la problématique. Par la suite, nous décernerons les objectifs de recherche. Enfin, nous conclurons par un aperçu du plan du mémoire.

1.1 Définitions et concepts de base

La *vérification* est une approche s'appuyant sur un raisonnement mathématique qui permet de prouver que la description formelle d'un système satisfait certaines propriétés souhaitées. Ces propriétés s'étendent des propriétés indépendantes du système sous vérification comme la consistance et la complétude au niveau de ses spécifications aux propriétés relatives à ses comportements et ses fonctionnalités désirés.

Plus spécifiquement, la *vérification formelle* comporte trois étapes globales : la modélisation du système, la spécification des propriétés attendues du système et finalement la preuve que le système modélisé possède bien les propriétés attendues.

Nous définissons par la suite la première étape indispensable pour toute vérification d'un système qui n'est autre que sa *modélisation formelle*. Cette étape vise à décrire clairement et sans ambiguïté, au moyen d'un modèle ou d'un langage, le comportment d'un système.

Un *modèle* est une représentation abstraite et simplifiée d'une entité qui peut représenter un processus ou un système du monde réel en vue de le décrire, de l'expliquer ou de le prévoir.

Concrètement, un *modèle* permet de réduire la complexité d'un phénomène en éliminant les détails qui n'influencent pas son comportement de manière significative. Il reflète ce que le concepteur croit important pour la compréhension et la prédiction du phénomène modélisé. Certes, il serait important de noter que les limites du système modélisé dépendront alors des objectifs du modèle.

Le sujet de notre projet de recherche touche primordialement à la vérification de systèmes par model-checking symbolique. Par conséquent, nous définissons tout d'abord le model-checking pour ensuite aborder une présentation du model-checking symbolique.

Le *model-checking* représente l'une des techniques possibles de la dernière étape de la vérification formelle. Son rôle illustré dans la figure 1.1 est de vérifier qu'une propriété donnée représentée par une formule logique φ satisfait le modèle établi M du système.

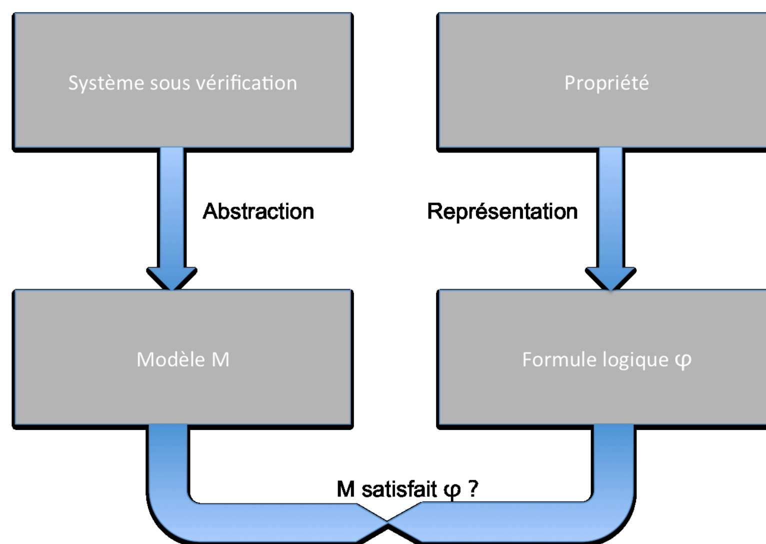

FIGURE 1.1 Vérification d'un système par model-checking

Le critère le plus intéressant du *model-checking* est sa possibilité de générer, dans le cas de violation de propriétés, des contre-exemples indispensables à l'interprétation des erreurs ou des disfonctionnements potentiels.

Le *model-checking symbolique* est une technique de model-checking jumelé à des méthodes diverses d'abstractions lui conférant l'aptitude à la vérification de tout système, fini ou infini. Les systèmes logiciels infinis peuvent inclure des contraintes non linéaires ou des méthodes employant la récursivité. Dans le cadre de systèmes infinis, le model-checking standard "souffre" du problème d'explosion d'états. D'où l'importance du *model-checking symbolique* qui se dote des abstractions dans la vérification de ce type de systèmes.

Par conséquent, le *model-checking symbolique* est non seulement adaptable à la grande taille des systèmes mais permet également la vérification d'une gamme plus vaste de propriétés auparavant impossibles à prouver.

1.2 Éléments de la problématique

C'est dans le cadre de la vérification de deux études de cas que nous exploiterons la technique de model-checking symbolique. En effet, notre motivation principale à recourir à cette technique découle de la nature des systèmes dont on désire vérifier certaines propriétés.

La présence de contraintes arithmétiques non linéaires comprenant des nombres à virgule flottante d'une part et le caractère interactif et imprévisible des systèmes d'autre part, constituent des facteurs déterminants dans l'augmentation considérable de la taille de l'espace d'états. Ce phénomène représente un obstacle majeur dans le model-checking qui procède à une exploration de tous les états possibles (en nombre infini ou très étendu) afin de vérifier si une propriété est satisfaite ou pas. C'est à ce niveau qu'émerge le rôle crucial des abstractions dans le model-checking symbolique contribuant à une atténuation du problème d'explosion d'états.

Nous présentons dans ce qui suit les critères problématiques relatifs aux deux études de cas envisagés dans le cadre de notre présente vérification.

Notre première étude de cas consiste en un *simulateur de vol* appartenant à la classe des systèmes embarqués réactifs. Dans un contexte d'industrie aéronautique, ce système a été modélisé dans un format spécifique de notation tabulaire mettant l'emphase sur l'interaction entre les variables d'entrée et les variables de sortie via des combinaisons de conditions et d'actions résultantes.

L'une des difficultés inhérentes à la vérification de ce système est engendrée par la présence de nombres à domaines infinis comme les variables réelles non seulement au niveau des valeurs de sortie mais également dans les transitions entre les états. En effet, dans ces systèmes, les transitions d'état dépendent des prédicats qui consistent en des contraintes sur des valeurs numériques à virgule flottante.

On manipule plus spécifiquement des systèmes à contraintes arbitraires et complexes sur des domaines finis et infinis. Une multitude d'états peut se présenter au niveau de chaque contrainte de ce système. De ce fait, au fur et à mesure de l'ex-

ploration de toutes les contraintes intégrées dans le système, nous assistons à une prolifération de la taille de l'espace d'états, aspect non désiré dans le model-checking. Le recours au model-checking symbolique est rendue, dans ce cas, une nécessité pour la vérification de ce système.

Notre seconde étude de cas consiste en un *système d'éditions collaboratives distribuées*. Par définition, ce système souvent appelé *système de fichier en réseau pair-à-pair* représente un système qui partage une même ressource, un fichier par exemple, entre plusieurs sites dans le réseau. La démarche principale consiste à stocker l'objet partagé sur des machines distantes de manière à ce que, du point de vue de l'utilisateur, tout se passe comme si les données étaient stockées localement. On parle alors de réplication de données entre les différents utilisateurs qui peuvent ainsi traiter une requête ou une opération d'édition sur leur propre copie locale.

Toutefois, l'architecture interactive de ce type de systèmes pose des problèmes de contrôle de concurrence. En outre, plusieurs processus qui travailleraient de manière incontrôlée sur les mêmes données pourraient remettre en cause la cohérence globale du système.

Afin de résoudre ces problèmes et dans le but de supporter une édition collaborative efficace, des transformées d'opérations non locales—transformées opérationnelles—implémentées dans des algorithmes d'intégration ont été proposées dans la littérature. Leur but principal consiste à garantir la propriété primordiale de consistance de ces systèmes : La convergence des copies au niveau de chaque site.

Toutefois, l'obstacle majeur envisagé dans ces algorithmes découle de l'infinité d'états compris dans ces systèmes. En effet, le caractère interactif imprévisible et inhérent à ces systèmes, combiné à la réplication de données, contribuent considérablement dans l'augmentation du nombre d'états.

Par conséquent, vérifier la bonne fonctionnalité de ces algorithmes représente un véritable défi dans le model-checking. Ici aussi, nous proposons d'intégrer le model-checking symbolique dans le processus de vérification des propriétés relatives au système considéré.

1.3 Objectifs de recherche

Depuis plusieurs années, le model-checking symbolique a constitué l'une des percées de la vérification formelle de systèmes logiciels. Néanmoins, son application en combinaison avec l'exécution symbolique pour la génération de contre-exemples sous forme de cas de test en entrée demeure limitée.

Par conséquent, l'objectif de cette recherche consiste à exploiter cette démarche dans la vérification de deux systèmes critiques.

Dans la première étude de cas, nous nous intéresserons à développer un outil permettant d'interfacer le simulateur de vol modélisé à l'aide de tables de décision avec un model-checker approprié, afin d'en vérifier des propriétés pertinentes.

Notre contribution dans cette étude de cas réside dans la vérification des propriétés d'accessibilité par model-checking symbolique dans un système dominé par des contraintes arithmétiques non linéaires nécessitant une résolution des contraintes au fur at à mesure de l'exploration des états du système.

Dans la seconde étude de cas, nous élaborerons un modèle décrivant le comportement des systèmes d'éditions collaboratives distribuées afin d'en vérifier les propriétés de convergence. L'originalité dans ce travail repose sur l'application d'une méthode formelle et plus spécifiquement, le model-checking symbolique, dans l'identification des scénarios de divergence entre les copies des utilisateurs du réseau pair-à-pair.

1.4 Plan du mémoire

Le présent mémoire comprend quatre chapitres. Le premier chapitre est l'introduction. Le second chapitre fait dans une première partie un tour de l'horizon sur les techniques adoptées dans la vérification formelle des systèmes pour ensuite présenter dans une seconde partie les outils considérés dans notre projet de recherche tout en discernant leurs avantages et leurs limitations dans le contexte de nos études de cas.

Les troisième et quatrième chapitres sont consacrés respectivement à la présentation des deux études de cas envisagés : Le simulateur de vol et le système d'éditions collaboratives distribuées. Nous discuterons dans ces chapitres de la démarche adoptée dans la vérification des propriétés relatives à chaque système et des résultats obtenus. Finalement, le cinquième chapitre remettra en relief le jumelage de l'exécution symbolique et du model-checking puis évoquera les avenues futures quant à l'application de cette technique.

Chapitre 2

REVUE DE LITTÉRATURE

La conception de tout système se déroule suivant quatre étapes classiques et consécutives : Après la description informelle du système à concevoir aboutissant à l'élaboration des cahiers de charge, une phase de formalisation et de validation du système au moyen de test et/ou de vérification suit résultant en d'éventuelles corrections. A ce niveau, la réalisation du système prend place suivie par la dernière phase de test et de mises au point.

Les méthodes formelles ont été introduites dans le cycle de développement des systèmes, plus spécifiquement dans la seconde étape, afin d'accroître la confiance dans les spécifications, surtout au sein des systèmes critiques du point de vue sûreté et sécurité.

Depuis plus d'une dizaine d'années, les outils basés sur les méthodes formelles ont constitué la préoccupation primordiale d'une multitude de chercheurs dans le domaine et ce, dans le but de détecter les erreurs et de vérifier les propriétés non seulement au niveau des systèmes matériels et logiciels mais également, dans leurs spécifications et leurs requis. Leur importance réside dans la détection des erreurs dans les premières phases de développement du système vu que la correction de ces erreurs dans des phases plus avancées du cycle est de loin plus coûteuse que celle dans des phases antérieures.

Dans ce chapitre, nous ferons dans un premier temps un tour d'horizon en présentant un état de l'art de la vérification formelle s'étendant du model-checking symbolique à l'intégration de l'exécution symbolique dans le model-checking. La contribution dans nos deux études de cas sera implicitement mise en évidence relativement aux travaux évoqués. Dans un second temps, une présentation concise des outils considérés tout au long de notre projet de recherche justifiera les techniques pratiques adoptées au sein de nos travaux.

Première partie : État de l'art

La vérification formelle comporte, en général, trois étapes :

- Modélisation du système : vise à décrire de façon claire et non ambiguë un système. Elle aboutit à un modèle ou un programme
- Spécification des propriétés attendues du système
- La vérification effective : Preuve que le système modélisé possède bien les propriétés attendues

Bien entendu, la vérification ne se limite pas au travail ardu de la modélisation du système et de spécification de propriétés attendues. Elle doit prouver que le système les satisfait. Il existe plusieurs méthodes de vérification qui peuvent être classées en deux grandes catégories : les *méthodes syntaxiques* et les *méthodes sémantiques*.

Les *méthodes syntaxiques* sont des preuves au sens mathématique du terme mais elles sont difficiles à automatiser. La démonstration de théorèmes est une approche qui tend à être de plus en plus automatisée et assistée par ordinateur. La preuve automatique de théorèmes consiste à laisser l'ordinateur prouver les propriétés automatiquement, étant donné une description du système, un ensemble d'axiomes et un ensemble de règles d'inférences. Cependant, la recherche de preuves est connue pour être un problème non décidable en général, c'est-à-dire que l'on sait qu'il n'existe (et n'existera jamais) aucun algorithme convergent (permettant de décider en temps fini si une propriété est vraie ou fausse). Il existe des cas où le problème est décidable (fragment gardé de la logique du premier ordre par exemple) et où des algorithmes peuvent donc être appliqués. Cependant, même dans ces cas, le temps et les ressources nécessaires pour que les propriétés soient vérifiées peuvent dépasser des temps acceptables ou les ressources dont dispose l'ordinateur. Dans ce cas, il existe des outils interactifs qui permettent à l'utilisateur de guider la preuve. La preuve de théorèmes comme avec le Theorem Prover PVS, par rapport au model-checking, a l'avantage d'être indépendante de la taille de l'espace des états, et peut donc s'appliquer sur des modèles avec un très grand nombre d'états, ou même sur des modèles dont le nombre d'états n'est pas déterminé (modèles génériques).

Les *méthodes sémantiques* se basent sur l'exécution du modèle pour vérifier si les

propriétés sont satisfaites. Elles sont applicables uniquement si le modèle a un nombre fini (et pas très grand) d'états (ou de zones d'états). L'approche la plus populaire est le *model-checking* qui s'appuie sur deux formalismes : automate et logique temporelle. Plus spécifiquement, le model-checking consiste à vérifier des propriétés par une énumération exhaustive (selon les algorithmes) des états accessibles.

Un avantage du model-checking relativement aux démonstrations de théorèmes est leur automatisation complète. L'efficacité de cette technique dépend en général de la taille de l'espace d'états accessibles et trouve donc ses limites dans les ressources de l'ordinateur pour manipuler l'ensemble des états accessibles. Des techniques d'abstractions (éventuellement guidées par l'utilisateur) peuvent être utilisées pour améliorer l'efficacité des algorithmes. Un autre avantage crucial des model-checkers est leur aptitude à trouver les contre-exemples—exécutions du modèle qui ne satisfont pas une propriété. Ces contre-exemples sont forts utiles à la compréhension des situations d'erreur et à la correction lors des phases amont de la conception de tout système.

La génération de ces contre-exemples justifie notre motivation à opter pour l'approche de model-checking dans la vérification des systèmes considérés dans le présent mémoire. De nombreux systèmes dans la littérature ont déjà été effectivement vérifiés formellement. On en cite deux : Les protocoles de contrôle de Bang & Olufson et de Philips à l'aide de l'outil de vérification UPPAAL [1] et un système logiciel critique déployé en avionique TCAS II (Traffic Alert and Collision Avoidance System II) à l'aide de l'outil de vérification SMV [2].

En somme, nous pouvons conclure que la démonstration de théorèmes est la méthode optimale de vérification dans l'absence d'informations concernant le modèle en question, ce qui nous incite à le modéliser au moyen d'axiomes. En revanche, dans le cas où des fonctionnalités précises du modèle sont fournies, comme dans notre projet de recherche, le model checking nous semble plus adéquat, d'où notre motivation de développer dans ce chapitre l'état de l'art du model checking en mettant en relief l'impact de nos contributions dans notre projet de recherche.

2.1 Model-checking

L'importance des méthodes formelles a été principalement associée à l'analyse statique des spécifications et du code mais, depuis plusieurs années, ces dernières prennent également de l'envergure dans le domaine du test, plus particulièrement en utilisant les model-checkers lors de la validation des systèmes. Les méthodes formelles, typiquement utilisées dans les phases de spécification et d'analyse lors du développement de systèmes logiciels, offrent à la fois l'opportunité de réduire les coûts du test et la possibilité d'augmenter la confiance dans le système sous développement.

Le recours aux méthodes formelles a largement réduit la possibilité d'introduction d'erreurs dans les premières phases de développement de systèmes. De plus, vu que la phase de test est une étape coûteuse relativement au budget total accordé à chaque projet, les méthodes formelles offrent un atout considérable dans la réduction de ces coûts. De ce fait, des contre-exemples générés au moyen de model-checkers représentent également des cas de test utiles. Gargantini et Heitmeyer ont utilisé des model-checkers afin de générer des tests pour des systèmes avec des spécifications représentées en SCR (Software Cost Reduction) [3].

Le *model-checking* a été principalement employé dans le but d'améliorer la qualité de développement des spécifications des systèmes durant leur cycle de vie. D'ailleurs, les model-checkers, basés sur les méthodes formelles, ont été originalement conçus dans le but de vérifier que les machines à états sont conformes aux spécifications exprimées dans une logique temporelle.

Par conséquent, la spécification d'un model-checker est divisée en deux parties :

– Le *modèle* qui n'est autre que la machine à états définie par un ensemble de variables et leurs valeurs initiales, ainsi qu'une description des conditions affectant les valeurs de ces variables. L'espace des états est alors défini par toutes les combinaisons possibles de valorisation des variables. Formellement, un système de transition d'états $\langle$ Q, R, I $\rangle$ consiste en un ensemble d'états Q, une relation de transition d'états $R \subseteq Q \times Q$ et un ensemble d'états initiaux $I \subseteq Q$. Un chemin d'exécution est une séquence infinie d'états telle que chaque paire consécutive d'états est incluse dans R. L'ensemble des états Q est souvent re-

présenté par un ensemble de variables d'état, tel que chaque état correspond à une évaluation ou valorisation de variables et en sorte qu'il n'existe pas d'états distincts correspondant à la même valorisation [2].

– Les *contraintes logiques temporelles* associées aux états et aux chemins d'exécution. Théoriquement, un model-checker visite tous les états accessibles et vérifie que les propriétés logiques temporelles, formulées en LTL (Linear Temporal Logic) ou CTL (Computation Tree Logic), sont satisfaites et ce, pour chaque chemin d'exécution possible. De ce fait, on peut dire que le model-checker détermine si la machine à états représente un véritable modèle de la formule logique temporelle. Cette formule peut être représentée sous forme de proposition qui consiste en une combinaison booléenne quelconque de prédicats sur les variables d'état. Elle peut également être formée d'une combinaison booléenne de formules ou sous la forme CTL par exemple, AG f, AF f, EG f, or EF f, où f est une formule. Chaque formule est évaluée dans un état défini q. Une proposition est dite satisfaite en q si q satisfait la proposition. L'opérateur A signifie "pour tous les chemins d'exécution à partir de q", E signifie "il existe un chemin d'exécution à partir de q", G signifie "pour chaque état le long du chemin d'exécution" et F signifie "il existe un état le long du chemin d'exécution". Nous citons quelques exemples pratiques de propriétés temporelles CTL :

  AG (request $\longrightarrow$ AF response) : Une requête est toujours suivie d'une réponse dans un certain état dans le futur. Cette propriété est dite propriété de vivacité ou "liveness property".

  AG EF restart : Il est possible de redémarrer le système dans un état accessible quelconque. Cette propriété est dite propriété d'accessibilité ou "reachability property".

Ces deux propriétés se classent dans une catégorie de propriétés indispensables à la vérification de tout système logiciel dans les phases de spécifications et d'analyse. Pour plus de détails concernant la syntaxe et la sémantique de la logique temporelle CTL et LTL, se référer à [5] et [4].

Pour récapituler, on peut dire qu'un model-checker satisfait une formule si cette

dernière est satisfaite dans tous les états initiaux. Dans le cas de violation de propriétés, le model-checker génère, s'il y a lieu, un contre-exemple sous forme de traces ou de séquences d'états.

L'approche de model-checking dans le cadre des méthodes formelles est considérée dans la littérature comme une approche émergente à laquelle les chercheurs en génie logiciel ont porté une attention particulière ces dernières années. Par conséquent, le champ d'application du model-checking s'est étendu des modèles matériels à l'analyse des protocoles, des systèmes d'exploitation, des systèmes réactifs et des systèmes de sécurité. Toutefois, de nos jours, le model-checking joue un rôle important non seulement dans l'analyse pure de ces systèmes mais surtout dans leur développement.

La majorité de la recherche et des études découlant du model-checking se concentrait en premier lieu sur des propriétés indépendantes de l'application ou du système en cours d'analyse. Pourtant, même si ces dernières sont critiques pour la vérification de tout système, elles restent néanmoins insuffisantes pour juger catégoriquement de l'*exactitude* d'un système en termes de spécifications. D'où la nécessité de concentrer la vérification non seulement à des fins de complétude, d'exactitude et de consistance mais aussi la diriger vers des propriétés pertinentes aux fonctionnalités désirées ou attendues du système. La vérification de cette catégorie de propriétés dépendantes de l'application nécessite le recours à des abstractions diverses dans les spécifications.

A ce niveau, l'introduction du model-checking symbolique jumelé à des techniques d'abstractions diverses prend de l'ampleur dans la littérature. Il existe actuellement plusieurs méthodes d'abstractions associées au model-checking afin d'atténuer le problème d'explosion d'états comme le model-checking symbolique que nous détaillerons dans la section suivante.

2.2 Model-checking symbolique

Bien que les méthodes formelles offrent l'avantage de la vérification automatisée, les techniques existantes découlant de cette théorie restent pratiquement non adaptables aux systèmes à zone d'états étendue. Davantage, elles nécessitent une orienta-

tion humaine extensive ou sont limitées à la vérification de propriétés simples—bien que primordiales comme la consistance et la complétude.

Ici se manifeste le rôle crucial du model-checking symbolique qui consiste en une technique de vérification automatique basée sur les BDDs (Binary Decision Diagrams). Le model-checking symbolique est non seulement adaptable à la grande taille des systèmes mais aussi permet la vérification d'une gamme plus vaste de propriétés auparavant impossibles à prouver.

2.2.1 Model-checking symbolique basé sur les BDDs

Définition

Dans les techniques de model-checking explicite, la valeur de vérité d'une formule CTL est déterminée sous forme de graphes en traversant le diagramme d'états avec une complexité de temps linéaire à la taille de l'espace d'états et à celle de la formule. Malheureusement, l'obstacle majeur dans ce genre de model-checking réside dans la taille de l'espace d'états qui est souvent exponentielle à la taille de la description du système, ce qui engendre le problème de model-checking connu sous le nom de *problème d'explosion d'états*.

L'une des importantes percées dans le model-checking met en relief l'introduction du model-checking symbolique : Contrairement à la visite des états individuels dans l'exploration conventionnelle de l'espace d'états, les model-checkers symboliques visitent un ensemble d'états simultanément. Cet ensemble d'états peut être représenté par des prédicats sur les variables d'états tel qu'un état appartient à l'ensemble si et seulement si le prédicat est satisfait dans ce même état. Les représentations succinctes jumelées aux manipulations efficaces de ces prédicats contribuent aux avantages du model-ckecking symbolique relativement au model-checking traditionnel [5].

Si l'espace d'états est fini, on peut supposer que les variables d'états sont booléennes et en nombre fini. Un prédicat sur ces variables est simplement une fonction booléenne représentée par des diagrammes binaires de décision, les BDDs réduits et ordonnés. Par définition, un *BDD* est similaire à un arbre binaire de décision à la dif-

férence que les sous arbres isomorphes doivent être fusionnés résultant en un graphe acyclique dirigé comme illustré dans la figure 2.1. Cet arbre a un nœud initial unique suivi de nœuds intermédiaires étiquetés avec des variables booléennes et finalement de nœuds terminaux étiquetés avec des ’0’ et des ’1’. Chaque nœud intermédiaire possède exactement deux arcs dirigés du nœud lui-même à deux autres nœuds-fils : les deux arcs sont implicitement étiquetés avec ’0’ et ’1’ et représentés respectivement par une ligne pointillée et une ligne continue. Le BDD résultant dans la figure 2.1 permet ainsi de représenter la fonction $f(x;y)=\overline{x+y}$ de manière plus compacte que dans les tables de vérité. De plus, d’après [5], tester la satisfiabilité et la validité d’une fonction booléenne représentée par un BDD est plus simple qu’avec une représentation de la même fonction avec des formules propositionnelles ou des tables de vérité.

A noter que pour maintenir une efficacité dans la représentation des BDDs, chaque chemin d’exécution peut contenir une variable donnée une seule fois au maximum et doit suivre un ordre linéaire fixe des variables. Pour plus de détails concernant les BDDs et plus spécifiquement la construction des BDDs réduits et ordonnés, se référer à [5].

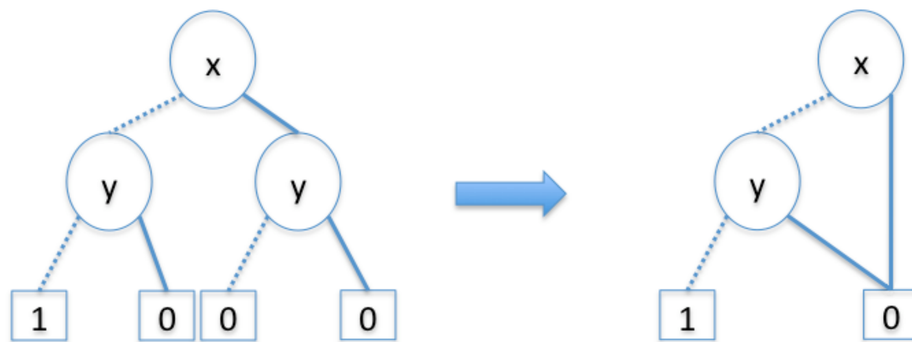

FIGURE 2.1 BDD réduit représentant la fonction $f(x;y) = \overline{x+y}$

L’utilisation des BDDs dans le model-checking a contribué à un essor considérable dans la vérification au début des années 1990.

L’un des privilèges associés à cette représentation est sa forme canonique : Pour la même fonction et le même ordre de variables, il existe un BDD unique représentant cette fonction. En plus, les BDDs offrent l’avantage d’exécuter des opérations de

conjonction, disjonction ou négation dans une complexité de temps polynomial.

Les BDDs sont plus ou moins petits de taille mais leur taille dépend majoritairement de l'ordre des variables [2].

Un grand nombre de model-checkers symboliques basés sur les BDDs ont été employés dans la vérification de circuits matériels. Ils représentent des ensembles d'état et souvent la relation de transition en BDDs. On cite un model-checker symbolique basé sur les BDDs, SMV qui a été utilisé dans [2]. La figure 2.2 illustre le processus de model-checking symbolique adopté dans le but de vérifier le modèle contre des propriétés.

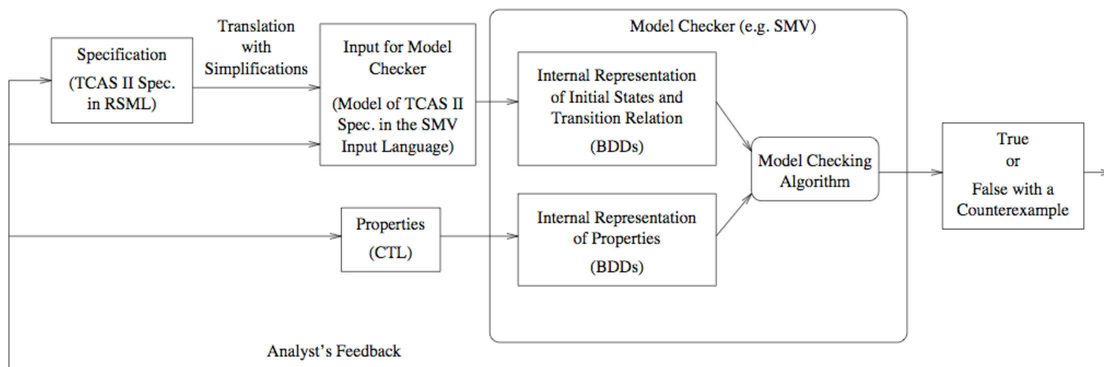

FIGURE 2.2 Processus de model-checking symbolique (Source [2])

Limitations

L'un des obstacles majeurs de ce genre de model-checking réside dans la limitation des BDDs quant à la nature des variables manipulées. En plus des variables booléennes et des énumérations, les variables d'entrée à TCAS II dans [2] sont majoritairement des nombres comme, par exemple, l'altitude et les angles de vision. Ces nombres peuvent même être des entiers ou des nombres réels.

Bien que les BDDs peuvent représenter efficacement les égalités et inégalités entre des expressions linéaires, ces derniers restent néanmoins dans l'impossibilité de re-

présenter efficacement la multiplication ou la division de variables. Afin de rendre possible l'utilisation des BDDs, Chan *et al.* ont été dans l'obligation de supposer que les variables d'entrée sont des variables entières bornées. En plus, une abstraction a été dirigée dans le but de considérer ces variables comme une partie d'un modèle non déterministe.

Vu que la performance des algorithmes basés sur les BDDs est directement reliée à leur taille, des techniques de changement dynamique d'ordre des variables et de partitionnement conjonctif ont été essentielles dans la mesure d'améliorer considérablement la vérification tout en réduisant la taille des BDDs. Toutefois, la taille des BDDs est toujours affectée par les bornes des variables numériques.

Ici aussi, la manipulation de cette classe de variables s'est avérée problématique vu que la réduction de la taille des BDDs a induit une réduction des bornes de ces variables ce qui a résulté en un modèle avec des comportements non désirés et, par conséquent, des résultats d'analyse invalides.

Introduction du model-checking dans la vérification des systèmes hybrides

La vérification de propriétés dans [2] comprend les deux classes de propriétés dépendantes et indépendantes de l'application. Pourtant, la plus grande limitation dans les expériences menées consiste en l'absence de l'environnement continu inhérent à ces types de modèles avioniques. En outre, la vérification des systèmes hybrides se concentre sur ce problème en modélisant l'environnement par un ensemble de valeurs réelles correspondant aux variables limitées par des contraintes de leurs dérivées.

Par définition, un *système hybride* est un système dynamique aux comportements dynamiques discrets et continus. Ce genre de model-checking devient plus compliqué, mais certains model-checkers spécifiques aux systèmes hybrides comme HYTECH ont été conçus pour s'attaquer à ce problème. Ces model-checkers restent toutefois inaptes à manipuler des multiplications et restent limités à des spécifications de petite taille, aspect désavantageux surtout dans le cadre de systèmes à nombre infini d'états et à complexité élevée.

Dans [6], des expériences ont été menées dans le but d'étendre la syntaxe du model-checker UPPAAL afin qu'un modèle hybride—un mécanisme de positionnement ou actuateur d'un robot mobile—modélisé et développé en UPPAAL soit passé à un autre

model-checker HYTECH à des fins de vérification. Notons que le robot modélisé est un système hybride caractérisé par son contrôleur qui, à partir des signaux discrets reçus, est capable de contrôler son actuateur, sa caméra et ses capteurs.

Par conséquent, le langage de modélisation doit permettre à la fois l'intégration des comportements discrets et continus. Mutfi *et al.* ont ainsi développé un adaptateur interconnectant les deux model-checkers en question : UPPAAL et HYTECH, des outils de vérification basés respectivement sur les automates temporisés et les automates hybrides.

Cette approche, bien qu'intéressante relativement à l'intégration des model-checkers dans la vérification des systèmes hybrides, demeure limitée dans le sens qu'elle n'offre pas un cadre générique de vérification. Ceci est dû à la limitation de la vérification de systèmes à partir d'UPPAAL qui est principalement un outil de modélisation, simulation et vérification de systèmes temps réel modélisés en une extension d'automates temporisés. Ce type de model-checker et son langage de modélisation résultant ne concorde pas avec les aspects critiques de nos modèles qui se concentrent sur des critères de contrôle pour déterminer les transitions possibles et où la notion de temps ne figure pas nécessairement dans la modélisation. En plus, on remarque que le modèle d'actuateurs décrit en HYTECH comprend simplement deux variables analogiques/réelles "distance" et "rotation" et une horloge "x" à valeurs réelles, mais il n'existe aucune opération manipulant ces variables. Plutôt, Mutfi *et al.* testent uniquement la valeur nulle ou non nulle (par exemple, "dist $\geq$ 0") de cette variable afin de déterminer l'état suivant ou la transition à effectuer sans recourir à des opérations arithmétiques comme dans notre première étude de cas. En effet, un exemple de contraintes arithmétiques dans notre système inclut une condition sur la valeur positive ou négative de la variable d'entrée *VelocityBodyX* formée elle-même d'un ensemble d'opérations arithmétiques non linéaires :

$$VelocityBodyX = CXX * northvelocity + CXY * eastvelocity + CXZ * downvelocity$$

où *CXX*, *CXY* et *CXZ* représentent des cosinus directionnels calculés préalablement alors que *northvelocity*, *eastvelocity* et *downvelocity* sont des variables réelles définies.

Abstractions dans le model-checking symbolique

L'explosion de l'espace d'états est un obstacle redoutable dans le model-checking de tout système dû à sa technique exhaustive d'exploration d'états et ce, pour les différents chemins d'exécution possibles.

Par conséquent, le recours aux abstractions en parallèle avec le model-checking symbolique demeure un atout, voire une nécessité, dans l'analyse de tout système logiciel même ceux à complexité moyenne. Pour être utiles, ces abstractions doivent préserver les propriétés de la spécification originale, qui nous intéressent.

Une variété de réductions ou d'abstractions existe et se concentre sur deux catégories majeures : l'*exactitude (soundness)* et la *complétude (completeness)* .

1. Dans une abstraction *exacte*, les propriétés de la spécification réduite ou abstraite sont également des propriétés de la spécification originale. L'exactitude évite les faux positifs (présence de contre-exemples qui ne devraient pas pourtant exister).
2. Dans une abstraction *complète*, les propriétés de la spécification originale sont aussi des propriétés de la spécification réduite. La complétude évite les faux négatifs (absence de contre-exemples qui devraient pourtant exister).

La littérature se concentre sur l'application du model-checking symbolique jumelé à diverses abstractions dans le développement de systèmes logiciels, plus spécifiquement dans les phases d'analyse et de spécification de ces systèmes, dans le but d'analyser diverses propriétés de la spécification des requis du système en question.

Bharadwaj et Heitmeyer ont formalisé une abstraction dans [7], appelée *RP1*, qui retire les entités non pertinentes lors du model-checking. Ainsi, afin de vérifier s'il existe une certaine propriété q qui satisfait la spécification, il est possible de retirer les variables d'entrée et les variables intermédiaires qui ne se rapportent pas à q. De plus, ils ont contribué à la formalisation de *RP2* qui abstrait les variables surveillées ou les variables d'entrée. Par exemple, la variable WaterPres, qui est une variable discrète s'étendant dans l'intervalle 0 à 2000 est directement quantifiée à l'une des variables TooLow, Permitted, ou High et peut être de ce fait retirée. Les deux

abstractions précédentes sont exactes et complètes pour l'analyse.

Dans [2], des expériences ont été menées par Chan *et al.* dans le but de tester si le model-checking symbolique est aussi efficace dans la vérification de systèmes logiciels que dans celle des modèles matériels. Pour ce faire, une large portion de la spécification des requis du système de TCAS II représentée en RSML (Requirements State Machine Language) a été traduite en une entrée à un model-checker symbolique SMV afin d'analyser quelques propriétés du système. Ces propriétés comprennent des propriétés générales de sûreté ainsi que d'autres propriétés de sécurité spécifiques au domaine de l'avionique, notamment :

- **AG safe** : Tous les états accessibles sont sécurisés.
- **AG AF stable** : Le système est stable infiniment souvent.
- **AG (request $\longrightarrow$ AF response)** : Une requête est toujours suivie d'une réponse dans un certain état dans le futur.
- **AG EF restart** : Il est possible de redémarrer le système dans un état accessible quelconque.

Chan *et al.* utilisent une autre méthode de réduction de l'espace des états tout en procédant au model-checking symbolique dans [2]. La plupart des spécifications définissent des bornes de temps sur les intervalles entre les événements. La spécification ordinaire conserve le temps dans une variable discrète, utilise des variables pour stocker les temps d'occurrence des événements et possède des prédicats sur la différence des temps. Alternativement, dans une spécification abstraite, Chan *et al.* exploitent des horloges (bornées) mesurant le temps à partir des événements. Une fois la borne atteinte, les horloges entrent dans un état "satisfait" (ou non "satisfait").

Suite aux abstractions combinées au model-checking symbolique, on peut dire que la vérification est améliorée dans la phase de spécifications. Plusieurs autres expériences ont été menées dans le but de contourner le problème des opérations arithmétiques non linéaires comme la multiplication.

Des model-checkers spécifiques [8] ont été conçus où le contrôle est représenté par des BDDs alors que les opérations sur les entiers sont représentées en utilisant des diagrammes de moment binaires BMDs (Binary Moment Diagrams) [9]. De plus, des HDDs (Hybrid Decision Diagrams) ont été utilisés pour représenter les fonctions à

variables entières existant dans la vérification de circuits arithmétiques permettant la représentation précise du produit de deux entiers par exemple. Cependant, ces model-checkers demeurent restreints à la vérification de circuits matériels et non adaptables à la vérification de spécifications de systèmes logiciels. En plus, une autre limitation dans cette technique réside dans le fait qu'il est possible de représenter de façon concise la variable xy mais, une fois une égalité comme $xy=z$ est nécessaire dans une représentation, on passe de nouveau à une taille exponentielle de diagrammes, aspect désavantageux quant à la performance des model-checkers.

Les algorithmes traditionnels du model-checking basé sur les BDDs ont été étendus dans [10] afin de vérifier des propriétés de sécurité de systèmes à contraintes arithmétiques non linéaires. Chan *et al.* optent pour la représentation de chaque contrainte par une variable sous forme de BDD (chaque contrainte est remplacée par une variable booléenne abstraite et équivalente), après avoir utilisé l'information fournie par un solutionneur de contraintes dans le but d' "élaguer" les BDDs tout en éliminant les chemins d'exécution qui correspondent à des contraintes impossibles. Leur technique a été mise en jeu dans un exemple simple en utilisant le model-checker symbolique SMV.

Dans [10], la limitation du model-checking symbolique basé sur les BDDs est mise en relief dans les systèmes à complexité élevée due à ses contraintes non linéaires. Par exemple, si les entiers x, y et z sont représentés par des BDDs, le BDD résultant de la contrainte non linéaire $xy=z$ est de taille exponentielle. D'où leur motivation de combiner un solutionneur de contraintes avec un model-checker basé sur les BDDs pour éliminer à la volée les chemins d'exécution non faisables tout en diminuant le nombre de BDDs résultants.

Il est important de noter que notre étude de cas appartient à la même catégorie de système visé à la vérification dans [10] : La classe de systèmes embarqués réactifs comprenant une composante de contrôle à état fini avec des données numériques en entrée qui mesurent des quantités comme la vitesse, l'angle de vision, de rotation, et autres. Dans ces systèmes, les transitions entre états dépendent des prédicats ou contraintes sur ces valeurs numériques.

Le problème qui découle de la technique adoptée dans [10] est le suivant : La théorie est bien développée sauf que son application pratique au niveau du model-checking

symbolique sur la classe de systèmes problématiques se base sur un exemple simple modélisant un système de régulateur de vitesse avec six contraintes arithmétiques non linéaires. Chan *et al.* n'ont pas exploré le côté pratique de cette technique au sens large du terme et aucun outil d'application a été envisagé. Davantage, notre étude de cas est non seulement dominé par une logique de contrôle découlant des contraintes arithmétiques non linéaires mais aussi de calculs éventuels et des transitions d'états basés sur les résultats de ces variables intermédiaires. Dans [10], chaque contrainte, assez compliquée soit-elle, est représentée sous forme de variables booléennes. Qu'en est-il alors des variables dérivées de ces contraintes et les variables dépendantes des résultats de ces contraintes ? La méthodologie quant à l'application de ces cas semble ambiguë et loin d'être réellement applicable dans la vérification de systèmes larges et réactifs.

Dans [11], Rushby adopte la vérification formelle automatisée qui utilise les techniques de déduction elles-mêmes automatisées et basées sur une combinaison de démonstration de théorèmes et de model-checking. Cette vérification procède à l'exécution des "calculs logiques" qui permettent de vérifier une classe de propriétés contre une classe de descriptions de systèmes.

Sa théorie se résume en ce qui suit : Au lieu d'opter pour un système abstrait ou simplifié construit à la main à partir d'un système original, on "calcule" la description du système abstrait assurant de ce fait son exactitude. Etant donnée une fonction d'abstraction reliant les deux espaces d'états original et abstrait, une condition de vérification peut être générée pour chaque paire d'états abstraits qui spécifie les conditions sous lesquelles aucune transition ne doit être franchie entre ces états dans la description du système abstrait (La condition est qu'il n'existe pas de transitions entre une paire quelconque d'états originaux associée à la paire d'états abstraits). Si la condition de vérification peut être prouvée (à l'aide de procédures automatiques de démonstration), la transition peut être omise dans la description du système abstrait. Sinon, cette transition est conservée dans le modèle abstrait.

De la sorte, cette technique d'abstraction appelée "abstraction omniprésente" réduit, d'après [11], les difficultés d'une part, et augmente la productivité et l'automatisation dans l'analyse formelle des systèmes simultanés d'autre part. Ce genre d'abstractions désigne la construction de différentes descriptions de systèmes abstraits dans

plusieurs points différents dans une analyse et pour des motivations diversifiées. Cette approche combine les deux points forts des démonstrateurs de théorèmes d'un côté et des model-checkers d'un autre côté. Le seul inconvénient réside dans la nécessité de modifier les outils afin de permettre l'échange mutuel de valeurs symboliques (par exemple, l'ensemble d'états accessibles ou un contre-exemple généré) au lieu de transmettre le succès ou l'échec de l'analyse locale dans chaque outil.

Dans le cadre de notre projet, le recours à de telles abstractions ne concorde ni avec la nature concurrente des systèmes visés dans ces expériences, ni dans l'utilisation des démonstrateurs de théorèmes justifiée par notre objectif de personnaliser la génération de contre-exemples, aspect absent dans la technique développée.

2.2.2 Model-checking symbolique sans les BDDs

Le model-checking symbolique basé sur les BDDs offre l'opportunité de vérifier efficacement des systèmes avec des centaines de variables d'état (plus de 10^{20} états). Toutefois, son application dans le cadre de systèmes larges à contraintes non linéaires souffre de l'impossibilité de représenter le grand nombre de BDDs générés relativement aux capacités des ordinateurs.

Davantage, la génération d'un ordre optimal de variables, menant à une minimisation désirée de la taille des BDDs, ne semble pas remédier à ce problème vu que c'est une procédure qui consomme excessivement du temps et nécessite une intervention manuelle.

A ce niveau intervient le model-checking symbolique basé sur les procédures de décision propositionnelles—notées de manière interchangeable par SAT—afin de pallier la vérification de systèmes larges et complexes.

Introduction au model-checking borné

L'approche suivie dans SAT se réduit au problème de déterminer si les variables d'une formule booléenne donnée peuvent être assignées de façon à rendre l'évaluation de la formule à "Vrai". En revanche, cette même formule est évaluée à "Faux" s'il n'existe aucune assignation possible pour ces variables. Dans ce dernier cas, on peut

conclure que la fonction ne peut pas être satisfaite ; sinon la fonction peut être satisfaite. Il reste à mettre en relief la nature booléenne de la fonction et de ses variables dans ces procédures de décision propositionnelles.

Contrairement aux BDDs, les SATs n'utilisent pas les formes canoniques. Ils ne souffrent pas du problème d'explosion d'états rencontré fréquemment dans les BDDs, et par ailleurs, peuvent supporter des systèmes avec des milliers de variables d'état. Les techniques basées sur SAT ont été appliquées avec succès dans plusieurs domaines comme la vérification de systèmes matériels et la vérification formelle de systèmes de contrôle des chemins de fer.

Biere *et al.* présentent une technique de model-checking symbolique basé sur les procédures de décision SAT dans [12]. L'idée de base est de considérer des contre-exemples de taille déterminée k à l'aide du model-checking borné et de générer par la suite, une formule propositionnelle qui peut être satisfaite si et seulement si un tel contre-exemple existe. Consécutivement, Biere *et al.* démontrent que le model-checking borné, jumelé à la logique propositionnelle linéaire LTL peut être réduit à un SAT en temps polynomial.

Par définition, le *model-checking borné* suppose que les bogues infiltrés dans un système peuvent être détectés à proximité de l'état suspect c.-à-d. dans un nombre limité de pas de recherche relativement au nombre d'états explorés. Cette approche n'est pas complète quant à la recherche de la totalité de l'espace d'états et s'est avérée efficace dans la détection d'erreurs se localisant près de l'emplacement suspect. Le model-checking borné est le mieux adapté à la vérification des impasses "deadlocks", des exceptions non traitées et des conditions de race.

Avantages du model-checking borné

En raison de l'impact privilégié des procédures de décision SAT, leur combinaison avec le model-checking borné peut être considérée comme étant une technique possible contournant l'obstacle redoutable lors de la vérification de systèmes—le problème d'explosion d'états. On cite ci-dessous les principaux apports du model-checking borné, où la "borne" dénote la taille maximale d'un contre-exemple.

- Le model-checking borné trouve des contre-exemples très rapidement. Ceci est illustré dans les résultats expérimentaux de [12], où un contre-exemple d'une taille de 2 MB est généré instantanément lors de la vérification d'une propriété de vivacité "liveness property", alors que la construction des BDDs dans les états initiaux n'était pas encore achevée. Ce résultat découle de la nature inhérente aux procédures de recherche SAT basées sur un parcours en profondeur. Certes, ce critère de rapidité de génération de contre-exemples est crucial au sein de tout model-checker.
- Le model-checker borné trouve des contre-exemples de taille minimale. Ceci facilite la compréhension et l'analyse ultérieures des contre-exemples générés.
- Il utilise moins d'espace relativement aux approches basées sur les BDDs.
- Finalement, et contrairement aux techniques de sélection manuelle de l'ordre des variables et de réordonnement dynamique adoptées dans les BDDs, des heuristiques de division fixées par défaut sont suffisantes [12].

Cette approche de model-checking borné basé sur des procédures de décision propositionnelles SAT a été appliquée sur des circuits synchrones et asynchrones dans [12] avec succès et avec des performances considérables par rapport aux techniques antérieures intégrant les BDDs.

Dans le cadre de nos études de cas, nous pouvons donc dire que cette approche s'avère intéressante et adaptable aux objectifs visés dans notre projet.

2.3 Model-checking : de l'analyse à la génération de tests

Les spécifications à haut niveau des systèmes logiciels tendent à être non adaptables aux outils formels, et de ce fait, doivent subir des réductions ou abstractions avant de procéder à une analyse efficace dirigée par un model-checker, et ce quel que soit le champ d'application du système en question.

La réduction, de préférence automatisée, permet de "transformer" des machines à états larges ou même infinis à plusieurs composantes disposées à toute manipulation

formelle. Bien que les techniques de réduction ou d'abstraction dont nous venons de discuter dans la section précédente, représentent un apport remarquable dans l'analyse de systèmes logiciels, elles restent toutefois concentrées sur une pure analyse de propriétés.

Par ailleurs, nous portons un intérêt particulier à la génération de contre-exemples au moyen d'un model-checker dans le but d'analyser plusieurs fonctionnalités et comportements désirés du système. Ces contre-exemples sont, en d'autres termes, des cas de test de la spécification originale mettant en relief les différentes combinaisons de variables d'entrée aboutissant à une violation d'une propriété prévue. Par définition, un cas de test dénote un ensemble de conditions et de variables qui permettent à un "testeur" de déterminer si une application ou un système logiciel fonctionne correctement. Ces cas de test connus sous le nom de cas de test formels, sont caractérisés par des variables d'entrée connues et des variables de sortie prévues.

C'est dans ce cadre que nous distinguons, dans la littérature, plusieurs travaux et expériences se concentrant sur l'application du model-checking dans le domaine du test, plus spécifiquement dans la génération de cas de test. Leur objectif principal consiste à mettre en relief l'impact de la réduction non seulement dans l'analyse de propriétés, mais également dans la génération automatique d'ensembles de test. On en cite la technique d'abstraction développée par Ammann et Black [13] et appliquée dans la génération de tests logiciels par model-checking.

RFF (Reduction Finite Focus) Cette réduction se résume comme suit : Les variables aux domaines larges ou non bornés sont associées à un sous-ensemble fixé de valeurs possibles. Par exemple, une variable entière x peut être modélisée par une variable correspondante $\mathrm{x}_{modèle}$ tout en ayant un intervalle borné formé de 0, 1 et 2. Par conséquent, l'idée de la réduction appelée "réduction à concentration finie" et notée RFF, consiste en un regroupement de toutes les variables à comportement non pertinent dans un état appelé "autre" ; les autres variables étant chacune représentée par son état adéquat.

La seule modification à apporter avant de recourir au model-checker reste à lui

indiquer la nécessité d'ignorer tout comportement et toute opération relatifs aux états étiquetés "autre". Ceci est réalisé en ajoutant deux autres états "sound" et "unsound".

La transition d'un état "sound" à un état "unsound" déclenche le passage à un état "autre". Par conséquent, le model-checker doit ignorer toute inconsistance ou tout comportement découlant de l'état "unsound" de façon à générer uniquement les cas de test problématiques dans le modèle original.

L'exactitude de cette technique de réduction a été prouvée dans [13] et a été appliquée sur une pile d'une machine virtuelle Java afin de démontrer la possibilité de réduire un modèle non borné à un modèle pouvant générer un ensemble—quoique simple—de tests pour une analyse de mutation. A noter qu'une analyse de mutation est une méthode de test logiciel qui procède à modifier le code source du programme de façon à générer des "mutations". Une telle analyse permet aux testeurs de développer des tests efficaces pour vérifier l'exactitude d'une implémentation d'un système logiciel donné.

Dans le paragraphe suivant, nous définissons le test par boîte blanche et nous introduisons la notion de "test formel" rencontré dans [14].

Test formel Par définition, une *boîte blanche* est un module d'un système dont on peut prévoir le fonctionnement interne car on connaît les caractéristiques de fonctionnement de l'ensemble des éléments qui le compose.

Le test par boîte blanche (white-box testing) représente une méthode de test logiciel visant à analyser un programme informatique dont on connaît exactement le fonctionnement interne : on peut même utiliser le code source du programme.

En revanche, le test par boîte noire est utilisé pour tester un programme en vérifiant que les sorties obtenues sont bien celles prévues pour des entrées données : Il n'y aucune information sur la structure interne ni sur le fonctionnement interne d'un système.

Vu que le test par boîte blanche, nommé test structurel, concorde plus avec nos objectifs en termes de l'information que nous possédons du système, nous développerons, dans ce qui suit, la notion de "test formel" développée dans [14].

Les développeurs de logiciels utilisent fréquemment les "modèles" afin de raisonner efficacement sur la conception de leurs systèmes. Cependant, garder une fidélité accrue entre les deux représentations des modèles d'une part et du code source d'autre part, est une tâche méticuleuse. Pourtant, elle reste cruciale dans la mesure où une divergence quelconque—assez infime soit-elle—aboutit à des obstacles non négligeables dans les phases aval de la conception incluant des erreurs de conception, une documentation inconsistante, et une obligation de correction particulièrement coûteuse.

C'est à ce niveau que le test par boîte blanche est privilégié dans la mesure où il permet aux développeurs d'établir une certaine fidélité entre les modèles et leur code durant le développement.

L'approche décrite dans [14], et appelée "test formel", est un processus de test basé sur les spécifications, et utilise les techniques de model-checking pour vérifier, organiser et générer des tests par boîte blanche durant la phase de développement.

Un model-checker, privilégié dans sa capacité d'analyser directement un modèle contre des propriétés diverses, sert aussi à manipuler et organiser un environnement de test. Dans [14], cette technique de "test formel" est réalisée au moyen du model-checker SPIN où le modèle est spécifié au moyen du langage Promela à travers un ensemble fini de processus asynchrones et simultanés qui interagissent à l'aide de variables partagées et de canaux de communication.

Contrairement au model-checker symbolique basé sur les BDDs, le model-checker SPIN procède "explicitement" en explorant l'arbre de calcul (computation tree) qui représente une structure théorique consistant en un ensemble infini de tous les chemins d'exécution possibles. Cette exploration sert à l'identification des chemins d'exécution satisfaisant une propriété déterminée. La génération de contre-exemples, suivant ce principe, est déclenchée lorsqu'il existe des chemins d'exécution dans l'arbre de calcul qui violent des assertions.

Toutefois, la limitation de cette technique découle de la nature même du model-checker SPIN qui est inapte de remédier au problème de contraintes arithmétiques non linéaires dominantes dans notre étude de cas.

2.4 Exécution symbolique

L'objectif commun à tous les travaux évoqués, dans le cadre de notre revue de littérature, se concentre sur la question suivante : "Est-ce que le programme modélisant notre système répond aux requis fournis ?".

A ce niveau, le model-checking symbolique borné semble représenter la méthodologie la plus performante au sein de systèmes larges à contraintes arithmétiques non linéaires avec des spécifications formelles. Néanmoins, dans l'absence de ces spécifications formelles, la technique de test paraît une solution plus ou moins optimale pour combler ce manque de formalisme. Suivant cette technique, nous distinguons deux choix extrêmes : le *test de programmes* et la *démonstration de programmes*.

Dans le *test de programmes*, le programmeur peut s'assurer que les échantillons de test à exécuter génèrent les résultats prévus, prouvant ainsi le bon fonctionnement du programme pour ces échantillons. Cependant, l'exécution correcte pour des variables d'entrée non comprises dans les échantillons fournis est non garantie.

La *démonstration de programmes*, quant à elle, est une approche méticuleuse où le programmeur prouve formellement que le programme répond à ses spécifications et ce, pour toutes les exécutions sans la nécessité de lancer le programme. Cette approche est redoutable en raison de la précision requise dans la construction à la main de spécifications dénotant le comportement exact du programme en plus de la nécessité de concevoir des procédures de preuves formelles pour démontrer la consistance entre le programme et la spécification.

Ici intervient le rôle de l'*exécution symbolique* prôné dans [15] qui consiste en une approche intermédiaire entre le test de programmes et leur démonstration.

King fournit une définition rigoureuse de l'exécution symbolique dont on résume les critères les plus pertinents dans le paragraphe suivant.

2.4.1 Définition

Dans l'exécution symbolique, un programme est exécuté "symboliquement", comme son nom l'indique, pour un ensemble de "classes" d'entrées à l'opposition du test ordinaire qui l'exécute pour un ensemble d'échantillons d'entrées.

De ce fait, l'exécution symbolique de programmes se caractérise par le remplacement des variables d'entrée concrètes d'un programme comme, par exemple, les nombres par des symboles représentant des valeurs arbitraires. De ce fait, l'exécution se déroule de façon similaire à l'exécution normale sauf que les valeurs peuvent être des formules symboliques comprenant des symboles d'entrée. Ainsi, tout résultat découlant d'une exécution symbolique peut être l'équivalent d'un grand nombre de cas de test ordinaires.

Déterminer la "classe" d'entrées dans toute exécution symbolique dépend du flot de contrôle du programme suivant ses variables d'entrée.

Si le flot de contrôle du programme est complètement indépendant des variables d'entrée, une seule exécution symbolique est suffisante pour vérifier toutes les exécutions possibles du programme. Si le flot de contrôle du programme dépend des entrées, une analyse de cas de test est nécessaire.

Avant d'aborder la discussion des travaux relatifs à ce sujet, nous présenterons, dans ce qui suit, la notion de base de l'application de l'exécution symbolique à un langage de programmation déterminé tel que discuté dans [15].

2.4.2 Intégration de l'exécution symbolique dans un langage de programmation

Chaque langage de programmation possède ses propres sémantiques d'exécution décrivant les structures de données qui représentent les variables du programme ainsi que leur manipulation par des déclarations, et le flot de contrôle à travers ces dernières.

Il est éventuellement possible de définir des sémantiques similaires de l'exécution symbolique pour un langage de programmation où les structures de données concrètes peuvent être représentées par des symboles arbitraires.

Par conséquent, l'exécution symbolique est une extension naturelle de l'exécution normale, dans le sens où les définitions de calcul pour les opérateurs de base du langage en question sont étendues afin d'accepter des variables d'entrée symboliques et de générer en sortie des formules symboliques sur ces variables.

Extension des sémantiques du langage de programmation Considérons un simple langage de programmation. Supposons que, dans ce langage, les variables du programme sont exclusivement de type "entier". En plus, supposons que ce langage comporte des expressions simples, comme les expressions "IF–THEN–ELSE" et quelques moyens d'obtenir les entrées comme des variables globales, des opérations de lecture, etc.

Limitons aussi les expressions arithmétiques aux opérateurs de base : addition '+', soustraction '-' et multiplication '*', ainsi que les déclarations booléennes dans "IF" au simple test de la positivité d'une expression (c.-à-d. expr ≥ 0).

En effet, nous remarquons que les "sémantiques d'exécution" changent pour s'adapter à l'exécution symbolique. Par contre, ni la syntaxe du langage ni les programmes individuels écrits dans ce langage ne sont modifiés.

Extension des variables d'entrée La seule opportunité d'introduire les données symboliques (représentant dans ce cas les entiers) est à travers les variables d'entrée du programme. À des fins de simplicité, supposons qu'à chaque fois qu'une nouvelle valeur d'entrée du programme est nécessaire, ce dernier choisit "symboliquement" cette valeur dans une liste de symboles $\{\alpha_1\ , \alpha_2, \alpha_3, ...\}$.

D'ailleurs, dans le but de manipuler des entrées symboliques, les valeurs des variables sont associées aux α_i et à des constantes entières.

Extension des expressions arithmétiques Les règles d'évaluation des expressions

arithmétiques utilisées dans les attributions et les déclarations IF doivent être étendues afin de manipuler les valeurs symboliques. Les expressions ainsi formées d'entiers—un ensemble de symboles $\{\alpha_1 , \alpha_2, \alpha_3, ...\}$—, de parenthèses et des opérations '+', '-' et '*' sont des polynômes d'entiers sur ces symboles. En permettant les variables de programme de gérer ces polynômes d'entiers comme valeurs, l'exécution symbolique des déclarations et des attributions suit naturellement : L'expression du côté droit d'une attribution est évaluée, possiblement en substituant les expressions polynomiales par des variables. Le résultat est un polynôme (un entier dans les cas les plus simples) qui est attribué au côté gauche de la déclaration d'attribution comme étant la nouvelle valeur de la variable.

Représentation d'un état L'état concret, dans une exécution d'un programme, comprend les valeurs des variables du programme et un compteur de déclaration indiquant la déclaration en cours d'exécution.

L'état symbolique—ou état dans une exécution symbolique d'un programme—nécessite l'ajout d'une expression booléenne comprenant les entrées symboliques $\{\alpha_i\}$. Cette expression n'est autre qu'une condition de chemin "path condition" notée *pc*.

Cette dernière ne contient jamais des variables de programme, et dans le cas de notre simple langage de programmation, est une liste d'expressions de conjonction de la forme $(R\geq 0)$ ou $\neg(R\geq 0)$, où R est un polynôme sur $\{\alpha_i\}$. La formule 2.1 représente un simple exemple de chemin de conditions :

$$\{(\alpha_1 \geq 0) \wedge (\alpha_1 + 2 * \alpha_2 \geq 0) \wedge \neg(\alpha_3 \geq 0)\} \tag{2.1}$$

La condition de chemin *pc* est alors un accumulateur de propriétés que les variables d'entrée doivent satisfaire pour que l'exécution se déroule selon le chemin d'exécution associé. Toute exécution symbolique commence par un *pc* initialisé à "Vrai". Tout au long des chemins d'exécution, *pc* est mis à jour en lui ajoutant la conjonction de chaque déclaration satisfaite q correspondant au chemin choisi.

Alors, comme résultat d'exécution, nous obtiendrons :

(*1*) pc $\longleftarrow$ pc $\wedge$ q, ou
(*2*) pc $\longleftarrow$ pc $\wedge\neg$ q

Il est important de noter ici deux catégories d'exécution lors de la rencontre de ces déclarations :

– **L'exécution "sans branchement"**

Suivant la rencontre d'une déclaration (expression)—une déclaration IF dans notre cas—si exactement une expression entre (*1*) et (*2*) est vraie, on continue l'exécution de la déclaration IF en passant le contrôle à THEN si (*1*) est vraie ; ou ELSE si (*2*) est vraie.

– **L'exécution "avec branchement"**

C'est un cas intéressant où ni exactement l'expression (*1*), ni exactement l'expression (*2*) est vraie. Dans ce type d'exécution, il existe au moins un ensemble de variables d'entrée du programme satisfaisant *pc* et prenant le chemin THEN, et il existe au moins un autre ensemble de variables d'entrée satisfaisant *pc* et prenant le chemin ELSE. Vu que les deux chemins sont possibles dans ce cas, la seule approche complète est d'explorer à la fois les deux chemins d'exécution. Ainsi, l'exécution symbolique introduit deux branchements "parallèles" : l'un suivant THEN et l'autre suivant ELSE.

Il reste à noter qu'une exécution d'une déclaration particulière IF peut être "avec branchement", alors qu'une exécution suivante de la même déclaration IF peut être "sans branchement".

Dans les exécutions "sans branchement", *pc* ne change pas vu qu'il n'existe pas de suppositions ajoutées à l'expression. De plus, *pc* ne peut jamais être égal à "faux" vu que sa valeur initiale est "vraie" et la seule opération modi-

fiant pc est une attribution de la forme :

$$pc \longleftarrow pc \wedge r \tag{2.2}$$

où r représente q ou $\neg q$ mais dans le seul cas où l'expression $(pc \wedge r)$ peut être satisfaite.

2.4.3 Arbre d'exécution symbolique

Il est possible de générer un "arbre d'exécution" caractérisant les chemins d'exécution suivis durant l'exécution symbolique du programme.

```
    int Calc(int y) {
    int x;

1:  si (y>=2)
2:      x:=y+1;
3:  sinon
4:      x:=y-1;
5:  return(x);

    }
```

FIGURE 2.3 Programme simple "Calc" à opérations arithmétiques linéaires

L'arbre est construit de la manière suivante : On associe un nœud à chaque déclaration exécutée (ce nœud est étiqueté avec le numéro de la déclaration). En plus, chaque transition entre ces déclarations est un arc dirigé connectant les nœuds associés. Pour chaque exécution d'une déclaration IF "avec branchement", le nœud correspondant aura deux arcs générés à partir du nœud et étiquetés "V" et "F" pour "Vrai" (THEN) et "Faux" (ELSE) respectivement.

Finalement, on associe à chaque nœud un état d'exécution courant ; l'état symbolique étant formé, comme précédemment mentionné, des valeurs des variables, du compteur de déclaration (numéro de l'instruction dans le code source) et de la condition de chemin pc.

Pour illustrer la construction d'un arbre d'exécution symbolique, nous considérons le simple programme "Calc" dans la figure 2.3 et son arbre d'exécution symbolique correspondant dans la figure 2.4.

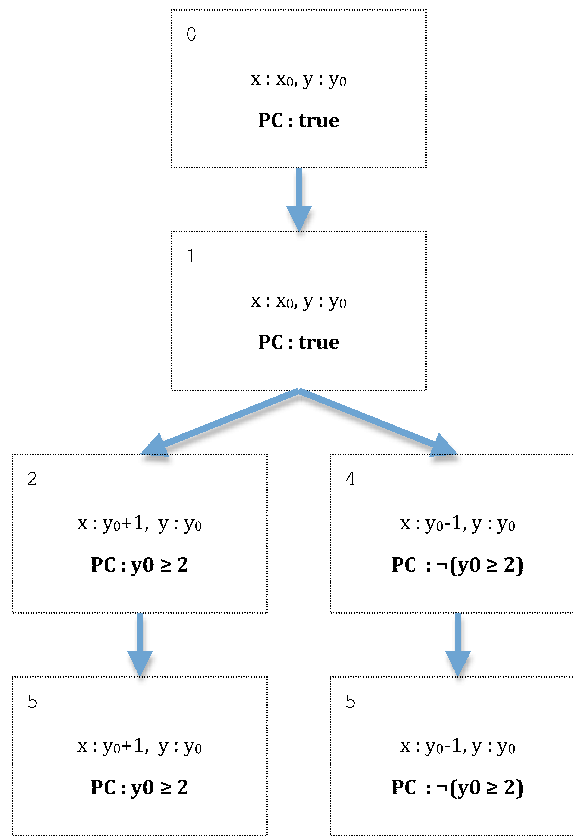

FIGURE 2.4 Arbre d'exécution correspondant au programme "Calc"

Introduction à l'outil EFFIGY King a développé dans [15] un exécuteur symbolique EFFIGY qui permet de réaliser du débogage au moyen de l'exécution symbolique. En plus, des techniques supplémentaires basées sur l'exécution symbolique ont été développées incluant un gestionnaire de test "exhaustif" et un vérificateur de programmes.

Le gestionnaire de test "exhaustif" est disponible pour explorer systématiquement les choix présentés dans l'arbre d'exécution symbolique. Le système peut automatiquement vérifier les résultats de cas de test vis-à-vis des assertions de sortie, si elles

existent.

Finalement, le système offre un vérificateur de programmes qui utilise l'exécution symbolique ainsi que les assertions fournies par l'utilisateur dans le but de générer les conditions de vérification.

Les aspects primordiaux des techniques de débogage établies dans [15] sont les suivants :

- **Traçage** : L'utilisateur peut demander d'avoir accès au numéro de la déclaration, la source de la déclaration et les résultats de calcul.
- **Points d'arrêt "BreakPoints"** : L'utilisateur peut insérer des points d'arrêt avant ou après toute déclaration ou entre toute paire de déclarations. L'exécution est, de ce fait, interrompue au niveau de ces points et le contrôle passe au terminal de l'utilisateur. Ce dernier peut ainsi vérifier l'état de l'exécution, l'ensemble des variables et peut, par la suite, reprendre l'exécution.
- **Sauvegarde des états** : Tout au long de l'exploration des différents chemins d'exécution du programme, l'utilisateur pourrait, si désiré, sauvegarder l'état d'exécution pour y retourner plus tard et parcourir les chemins d'exécution alternatifs. "SAVE" et "RESTORE" sont des opérations disponibles dans EFFIGY pour répondre à ces besoins.

  Il est intéressant de noter que cette sauvegarde des états est un critère crucial dans les déclarations "avec branchement". Dans ce cas, en utilisant "SAVE" et "RESTORE", c'est à l'utilisateur de sauvegarder l'état correspondant à son choix ("gotrue", "gofalse" ou "assume P") et c'est également à lui d'y retourner plus tard pour explorer les chemins alternatifs.

Limitations de l'outil EFFIGY L'une des limitations de l'approche considérée dans [15] est dans le langage pour lequel l'exécution symbolique est possible dans EFFIGY. En effet, les opérateurs arithmétiques, comprises dans ce langage, se limitent à l'addition '+', la soustraction '-', la multiplication '*', la division '/', ainsi qu'à la valeur absolue ABS et le modulo MOD.

Les expressions arithmétiques impliquées dans notre simulateur de vols, comme nous allons voir dans le chapitre suivant, s'étendent à des opérations arithmétiques complexes (racine carrée, exposant, etc.) et même des opérations trigonométriques (sin, cos, $\tan^{-1}$, etc.).

Le but primaire de l'exécuteur symbolique EFFIGY se concentre sur l'introduction du concept d'exécution symbolique dans des applications pratiques. De ce fait, il constitue un outil pionnier implémentant l'exécution symbolique dans la vérification de programmes séquentiels avec un nombre fixé de variables entières. En outre, l'exécution symbolique est également utile dans l'analyse de programmes incluant la génération de cas de test et dans l'optimisation de programmes comme nous allons voir un peu plus loin dans ce manuscrit.

2.4.4 Intégration de l'exécution symbolique dans le model-checking et le test

Le model-checking est privilégié pour ses performances automatisées d'analyse d'un système donné. En revanche, l'un de ses inconvénients réside dans le problème d' "explosion d'états", dans sa nécessité de contrôler des systèmes fermés—englobant à la fois le système et son environnement—et dans sa restriction des tailles des variables d'entrée.

L'exécution symbolique, automatisant la génération de cas de test, contourne ce problème en permettant le model-checking de programmes simultanés ayant des variables d'entrée dans des domaines non bornés et à structure complexe. En plus, cette technique d'analyse de programmes aide à combattre l'explosion de l'espace d'états en utilisant, entre autres, les réductions d'ordre partiel et de symétrie.

Nous présenterons, à ce niveau, la généralisation de l'exécution symbolique conventionnel telle qu'évoquée dans [16].

En premier temps, Khurshid *et al.* définissent une traduction "source-à-source"

pour instrumenter un programme. Ceci a comme but d'utiliser un model-checker standard à des fins d'exécution symbolique sans avoir recours à des outils dédiés. Par conséquent, tout model-checker supportant un choix non déterministe et le rebroussement de chemin, peut alors exécuter "symboliquement" le programme instrumenté. Le model-checker vérifie le programme en explorant automatiquement les différentes configurations de tas des programmes et en manipulant les formules logiques sur les valeurs associées aux données du programme (en utilisant une procédure de décision).

En deuxième temps, Khurshid *et al.* présentent un nouvel algorithme d'exécution symbolique qui permet la manipulation des éléments avancés de programmation comme les structures de données dynamiquement allouées : Listes et arbres.

A noter que les propriétés d'exactitude, vérifiées dans [16], sont fournies sous forme d'assertions dans le programme et de spécifications temporelles. En plus des contributions citées précédemment, l'exécution symbolique dans [16] joue un rôle primordial dans :

- La réduction du problème d'explosion de l'espace d'états : La vérification du comportement du code est rendue possible en utilisant les valeurs symboliques qui représentent des données dans des domaines étendus au lieu d'énumérer et de vérifier un ensemble limité de valeurs concrètes.
- La modularité : Vérifier des programmes avec des variables non initialisées permet de vérifier une unité de compilation en isolation.
- La vérification de propriétés d'exactitude de systèmes en choisissant des variables d'entrée à partir de domaines non bornés avec une structure complexe.
- Le profit dans les capacités inhérentes au model-checking, comme les différentes stratégies de recherche, la vérification de propriétés temporelles ainsi que les réductions d'ordre partiel et de symétrie.

Ces contributions diverses prônées dans [16] ont été illustrées dans la figure 2.5.

Pour récapituler, afin que l'exécution symbolique soit possible dans un model-checker, Khurshid *et al.* instrumentent le programme original en procédant à une traduction "source-à-source". Cette dernière ajoute le concept de non déterminisme et le support de formules représentant les conditions de chemin.

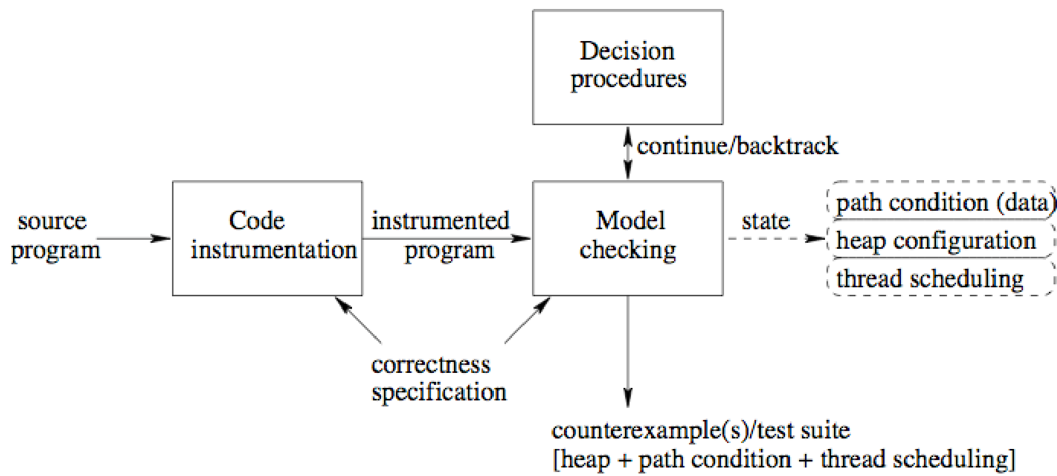

FIGURE 2.5 Exécution symbolique et model-checking (Source [16])

Par la suite, le model-checker vérifie le programme dûment instrumenté en utilisant ses propres techniques d'exploration de l'espace d'états. Un état déterminé inclut une configuration de tas, une condition de chemin et un ordonnancement de fils d'exécution. A chaque fois qu'une condition de chemin est mise à jour, elle est passée à une procédure de décision appropriée pour vérifier si elle peut être satisfaite. Si elle ne l'est pas, le model-checker rebrousse chemin.

Le model-checker employé dans [16] est le model-checker explicite Java PathFinder alors que la décision de procédure est la libraire Omega de Java PathFinder pour des contraintes entières linéaires.

Les spécifications d'exactitude peuvent être représentées sous forme d'assertions ou, plus généralement, sous forme de propriétés de sécurité.

L'exécution symbolique permet de trouver des contre-exemples violant ces propriétés de sécurité. Le travail développé dans [16] permet ainsi de prouver l'exactitude des programmes possédant des arbres d'exécution finis et des contraintes de données décidables. Plus spécifiquement, leur approche adoptée a été appliquée pour la vérification d'un algorithme distribué dans des programmes multitâches avec des structures

de données complexes, les listes chaînées entre autres.

En outre, nous citons l'application la plus pertinente relativement à notre sujet de recherche : La génération de cas de test pour un logiciel spécifique à une entreprise aéronautique. L'algorithme de parcours en largeur utilisé dans le model-checking a permis à Khurshid *et al.* de générer des tests d'entrée couvrant toutes les conditions avec un temps d'exécution de moins de 22 minutes.

Cependant, la restriction des procédures de décision et des solveurs de contraintes quant aux nombres à virgule flottante et aux contraintes non linéaires représente un véritable obstacle dans notre étude de cas : Le simulateur de vol.

Toutefois, nous jugeons l'exécution symbolique durant le model-checking comme étant une approche puissante dans l'analyse et dans la génération de cas de test des systèmes logiciels.

Dans la section suivante, nous exposons quelques abstractions jumelées à l'exécution symbolique afin de la rendre plus extensible—critère primordial à la vérification de systèmes logiciels larges.

2.4.5 Abstractions combinées à l'exécution symbolique

Dans le but de limiter ou contracter l'espace d'états, les travaux considérés antérieurement ont proposé la combinaison de l'exécution symbolique avec le model-checking pour l'analyse de programmes : Il y a une limitation de la taille des variables d'entrée du programme et/ou de la profondeur de la recherche du model-checker. Dans [17], cette approche de model-checking borné a été remplacée par des techniques de couplage d'états "state matching". Ces techniques s'appuient sur une comparaison entre les états symboliques suivant le concept suivant : Un état symbolique s_1 "subsume" un autre état symbolique s_2 si l'ensemble des états concrets représentés par s_1 est un sur ensemble de l'ensemble des états concrets représentés par s_2.

Basés sur ce principe, Anand *et al.* vérifient, tout au long de l'exécution symbo-

lique, si chaque état symbolique est "subsumé" par un autre état symbolique. Une multitude d'approches d'exploration de l'espace d'états, basées sur l'exécution symbolique et la vérification d'un ensemble d'états "subsumés", ont été implémentées dans l'outil Java PathFinder dans [17].

Toutefois, dû au nombre infini (possiblement) des états symboliques, cette méthode de couplage d'états n'est pas suffisante pour assurer la terminaison de l'exécution.

De ce fait, des techniques d'abstractions supplémentaires ont été exposées dans [17] pour calculer et collecter les états abstraits durant l'exécution symbolique. Plus précisément, ces abstractions ont été mises en œuvre dans des programmes manipulant des listes et des tableaux. Vérifier si deux états sont subsumés détermine si un état abstrait est revisité ; dans ce cas, le model-checker doit rebrousser chemin : C'est l'analyse d'une "sous approximation" des comportements de programmes. On parle ici d'abstractions sur la structure du programme plus que sur les contraintes numériques—ou abstractions sur les prédicats.

En somme, l'application de cette approche s'avère intéressante dans la détection d'erreurs dans des programmes Java. En revanche, elle ne semble pas s'attaquer au problème de vérification d'un système contre des propriétés données. Un moyen possible de réaliser la vérification à l'aide des abstractions évoquées dans [17] est de sauvegarder l'état abstrait et de commencer l'exécution symbolique à partir de cet état.

Généralement, le model-checking avec abstractions, surtout l'abstraction sur les prédicats et l'analyse statique, sont des techniques de "sur approximation" des comportements du programme.

L'approche de "sous approximation" mentionnée ci-haut est complémentaire aux méthodes de "sur approximation" et peut être utilisée en conjonction avec des méthodes similaires dans la mesure où on est intéressé dans la génération de contre-exemples "faisables". Par conséquent, vu que les techniques de "sur approximation" sont impliquées dans la vérification et la preuve de systèmes logiciels, nous considérons combiner l'apport de ces deux techniques complémentaires dans un même outil.

D'où notre motivation d'utiliser l'outil Java PathFinder que nous présenterons dans une section suivante.

Avant de conclure cette section de la revue de littérature et avant d'aborder la présentation des outils considérés tout au long de notre projet de recherche, nous citons quelques outils procédant sous le concept de l'exécution symbolique.

Notamment, PreFix est un outil de recherche de bogues qui a remporté un grand succès dans des applications commerciales diverses. PreFix analyse des programmes écrits en C/C++ et vise la détection de défauts concernant la gestion dynamique de la mémoire. Cependant, PreFix ne vérifie pas des propriétés riches comme des propriétés d'invariance sur les structures de données ni des propriétés d'accessibilité en présence de contraintes arithmétiques comme dans notre première étude de cas.

L'outil Extended Static Checker—noté *ESC*—utilise un démonstrateur de théorèmes pour vérifier l'exactitude partielle des classes annotées avec des spécifications JML (Java Modelling Language). L'outil ESC peut être utilisé pour vérifier l'absence d'erreurs telles que les accès aux données avec un pointeur nul, les violations des bornes de tableaux et les divisions par zéro. Par contre, il ne permet pas la vérification de propriétés plus générales.

Finalement, les deux outils VeriSoft et Java PathFinder opèrent directement sur des programmes écrits en C et Java respectivement. Une instrumentation adéquate de ces programmes permettra leur exécution symbolique durant le model-checking.

Toutefois, nous avons opté plus particulièrement pour l'outil Java PathFinder dans nos applications en raison de son extension Symbolic JPF. L'un des avantages de cette extension par rapport à l'outil Verisoft consiste en une intégration d'un solutionneur de contraintes avancé permettant la vérification de systèmes avec des contraintes arithmétiques à virgule flottante.

Seconde partie : Présentation des outils

Dans cette section, nous procédons à la présentation des outils considérés tout au long de notre travail et ce, suivant l'ordre logique de leur manipulation : Le passage d'un outil à un autre est justifié par un obstacle limitant son utilisation dans le cadre de notre travail. Par conséquent, nous entamons cette présentation par le model-checker NuSMV.

2.5 NuSMV

NuSMV est un outil de vérification formelle de systèmes à états finis. Il représente une nouvelle implémentation et une réingénierie du model-checker SMV, d'où son nom "New Symbolic Model Verifier".

Le model-checker NuSMV permet la vérification de systèmes contre des spécifications exprimées dans la logique temporelle CTL (ou LTL). La description des systèmes vérifiés à l'aide de NuSMV s'étend des systèmes complètement synchrones aux systèmes complètement asynchrones. Brièvement, le langage utilisé dans NuSMV a comme but principal la description de la relation de transition d'une structure Kripke finie en utilisant des expressions dans une logique propositionnelle. Une structure Kripke représente un quadruplet $M=\langle S, I, R, L\rangle$ tel que :

- S est un ensemble fini de tous les états.
- $I \subseteq S$ est l'ensemble des états initiaux
- La relation de transition $R \subseteq S \times S$ associe à tout état $s \in S$ un état $s' \in S$ tel que le couple (s ;s') $\in R$.
- Une fonction d'étiquetage L qui associe à tout état $s \in S$ un ensemble $L(s)$ de toutes les propositions atomiques valides dans cet état s.

Pour plus de détails sur les structures Kripke, se référer à [5].

Cette traduction des structures Kripke au langage de NuSMV résulte en une flexibilité élevée mais aussi en un risque d'inconsistance (provenant d'utilisateurs non

experts).

En somme, NuSMV prend en entrée un texte consistant en un programme décrivant le modèle ainsi que les spécifications sous forme de formules en logique temporelle. Comme résultat, NuSMV génère le mot "true" si les spécifications sont satisfiables ; sinon, il génère une trace montrant où la spécification est fausse pour le modèle représentant notre programme.

NuSMV a été explicitement conçu pour être un système ouvert, en d'autres termes, pour être facilement modifié, personnalisé ou étendu. L'architecture du système de NuSMV est divisée en plusieurs modules comme illustrés dans la figure 2.6.

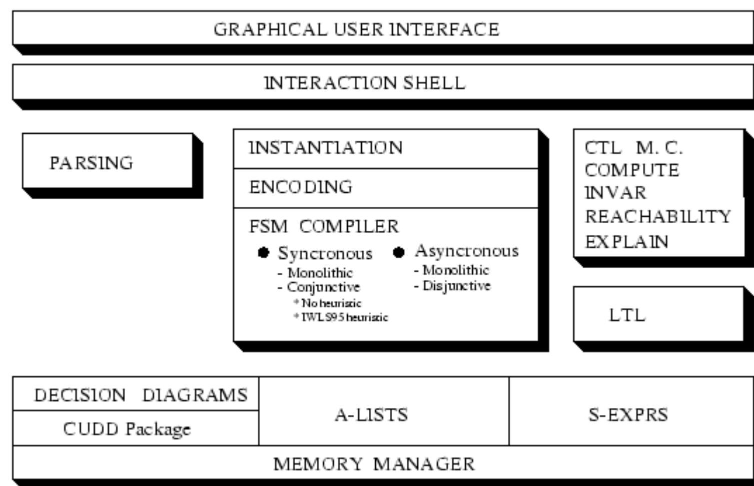

FIGURE 2.6 Architecture du système NuSMV (source [19])

Pour plus de détails concernant les fonctionnalités de chaque module et la communication entre eux, voir [19].

Vu que l'objectif principal de NuSMV est la description des machines à états finis, les seuls types de données dans le langage sont les données discrètes comme les variables booléennes, les scalaires et les tableaux à dimension fixée contenant des types primaires de données. Ce dernier critère représente un véritable obstacle dans la vérification de systèmes numériques contenant des variables réelles et des opérations trigonométriques complexes.

De ce fait, nous avons opté pour un autre outil de vérification, Alloy Analyzer, intégrant des abstractions afin de rendre possible la vérification de la classe de systèmes qui nous intéresse.

2.6 Alloy Analyzer

Dans Alloy Analyzer, les modèles ainsi que les spécifications sont transformés en relations. Le model-checking procède dans le cadre de cet outil à l'analyse d'une relation compatible entre le modèle d'une part et les spécifications d'autre part.

Le langage Alloy peut être considéré comme un langage d'analyse formelle adéquat à la représentation de ces transformations de modèles.

Basé sur la logique relationnelle, le langage Alloy représente un langage déclaratif textuel qui s'appuie sur une logique du premier ordre. A l'opposition du langage impératif utilisé dans NuSMV, un langage déclaratif est plus naturel et concis dans la description de modèles. Par exemple, pour modéliser un jeu donné, il suffirait de décrire les règles du jeu et les critères de victoire ou de perte. En revanche, dans un langage impératif, il devrait avoir une description détaillée de tous les mouvements aboutissant à la victoire ou à la perte, ce qui peut s'avérer impertinent dans le cadre d'analyse du système.

Sous le principe du langage relationnel, un modèle Alloy consiste en des signatures *Signatures*, des relations *Relations*, des faits *Facts* et des prédicats *Predicates*. Les signatures représentent les entités du système tandis que les relations illustrent l'interaction entre ces entités. Les faits et les prédicats spécifient les contraintes sur les signatures et les relations. De plus, le langage Alloy permet de manipuler des opérations arithmétiques (addition '+', soustraction '-', multiplication '*', etc.) et des opérations relationnelles ($\leq$, $\geq$, etc.) sur des nombres entiers (naturels et relatifs).

Alloy Analyzer est l'outil qui supporte l'analyse complètement automatique d'un système modélisé en langage Alloy. Il nous offre l'avantage de visualiser les solutions

ou contre-exemples obtenu(e)s à travers son outil de visualisation.

Alloy Analyzer est caractérisé par deux fonctionnalités majeures : La *simulation* qui assure la consistance du modèle—produisant une instance aléatoire du modèle et qui est conforme à la spécification—et les *assertions* qui sont des contraintes que le modèle doit satisfaire. La démarche adoptée par cet outil se résume par le suivant : Les formules Alloy sont traduites en expressions booléennes qui sont, par la suite, analysées par des solutionneurs SAT intégrés dans l'outil.

Il faut noter ici, que c'est à l'utilisateur de spécifier la portée "scope" des éléments du modèle lors de la vérification. Par exemple, une commande de vérification d'une assertion "check *OutputVariable* for 5" limite le nombre de signatures de *OutputVariable* à 5 pour la vérification. Ceci a comme but de limiter le domaine ou l'espace d'états. Si une instance violant une assertion est trouvée dans cette portée, on dit que l'assertion est invalide. Cependant, si aucune instance n'est trouvée, l'assertion peut toujours être invalide au-delà de cette portée. Ici intervient l'hypothèse de la portée réduite ou "small scope hypothesis" développée dans [20] qui justifie que la violation de propriétés est généralement localisée dans une portée assez limitée.

Bien que le langage Alloy supporte des structures de données complexes comme les arbres, ce langage comprend un système simple de types. Les seuls types *primitifs* qu'Alloy supporte sont les entiers "int". Par conséquent, il est impossible d'utiliser Alloy pour analyser des propriétés incluant des chaînes de caractères "String" ou des nombres réels comme dans notre première étude de cas. Des abstractions de ces types et l'utilisation de fonctions ont été adoptées dans notre travail. Néanmoins, ces techniques se sont avérées insuffisantes pour pallier le problème découlant des contraintes numériques non linéaires.

Pour plus de détails concernant Alloy et l'outil Alloy Analyzer, consulter la documentation offerte sur [42].

2.7 JForge

JForge est un outil d'analyse statique pour la vérification de programmes bornés écrits en Java. Afin de pouvoir analyser un programme à l'aide de JForge, ce pro-

gramme ainsi que sa spécification doivent être tous les deux traduits en un langage intermédiaire FIR (Forge Intermediate Representation) comme illustré dans la figure 2.7.

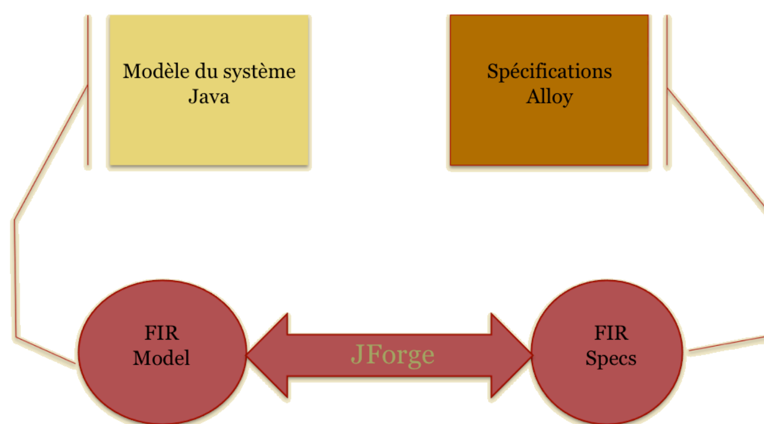

FIGURE 2.7 Fonctionnalité générale de JForge

De ce fait, le modèle à vérifier est écrit en langage Java supportant des types primitifs et non primitifs, des structures de données complexes et même offrant des méthodes qui opèrent sur ces données de façons linéaire et non linéaire—aspect désiré dans notre simulateur de vol. Les spécifications de ce modèle, quant à elles, sont infiltrées dans le code sous forme d'annotations représentées par des expressions du langage JFSL (JForge Specification Language). Ce langage relationnel du premier ordre combine des aspects de JML (Java Modelling Language) et d'Alloy et est détaillé dans [21].

Bien que cette méthodologie semble parfaitement adéquate dans notre cas, l'outil JForge considéré ne permettait pas d'ajouter des librairies externes telles que les classes implémentant les intervalles arithmétiques que nous détaillerons dans le prochain chapitre.

Cet outil étant toujours en cours de développement, de diverses limitations dans la configuration de JForge, nous ont incités à recourir à un autre outil de vérification caractérisé par son degré de maturité élevé au niveau des systèmes logiciels et consistant en un langage de modélisation vaste qui n'est autre que le langage Java.

D'où notre dernière motivation à l'utilisation de "Java PathFinder" introduit dans la section suivante.

Pour plus de détails concernant JForge, consulter la documentation sur [43].

2.8 Java PathFinder et ses extensions : JPF-SE et "Symbolic JPF"

2.8.1 Préliminaires

L'outil "Java PathFinder"—noté JPF— est un model-checker à état explicite pour les programmes Java. Il a été initialement développé en 1999 par le groupe de recherche "NASA Ames Research Center" jusqu'à son annonce publique comme logiciel libre à partir de l'année 2005.

Plus précisément, JPF est un système qui vérifie des programmes exécutables en codes objets ou bytecodes Java vu qu'il représente en soi une machine virtuelle Java. Par conséquent, JPF peut manipuler tous les aspects du langage Java mais, son critère privilégié réside dans son traitement de choix non déterministes exprimés sous forme d'annotations dans les programmes analysés. Ce dernier critère permet à la fois l'implémentation de mises à jour des conditions de chemin et l'initialisation des attributs. De ce fait, JPF supporte les annotations de programmes dans le but de causer l'exploration de l'espace d'états à rebrousser chemin lorsqu'une condition déterminée est évaluée à "Faux" : Ceci permet l'arrêt de l'analyse des chemins irréalisables dus à la présence de conditions de chemins non satisfiables.

Bien que le model-checking représente théoriquement une méthode de vérification sécuritaire et robuste dans le cadre de systèmes logiciels, son application pratique souffre du problème d'extensibilité. Par ailleurs, dans le but de pallier ce problème, le model-checker JPF s'appuie à la fois sur des heuristiques flexibles configurables par l'utilisateur (comme le parcours en largeur BFS (Breadth-First Search) et le parcours en profondeur DFS (Depth-First Search)) et sur des abstractions d'états lui octroyant deux points forts vis-à-vis des model-checkers existants : La configurabilité

et l’extensibilité.

Aujourd’hui, ce model-checker est prôné sur [44] :
"JPF is a swiss army knife for all sort of runtime based verification purposes", en d’autres termes, JPF offre un cadre général de model-checking pour une variété de techniques de vérification de bytecodes Java.

Basé sur la technique de model-checking, JPF exécute le programme plusieurs fois (à l’opposition d’une machine virtuelle standard qui l’exécute une seule fois) mais, théoriquement de toutes les façons possibles afin de vérifier les violations de propriétés comme les impasses ou états critiques de blocage "deadlocks " et les exceptions non traitées tout au long des chemins d’exécution possibles. Dans le cas d’identification d’erreur(s), JPF génère un rapport de vérification retraçant ainsi la totalité de l’exécution y aboutissant, et ce en collectant pas à pas les états jusqu’au défaut trouvé. Ce critère renforce son utilisation relativement aux débogueurs ordinaires.

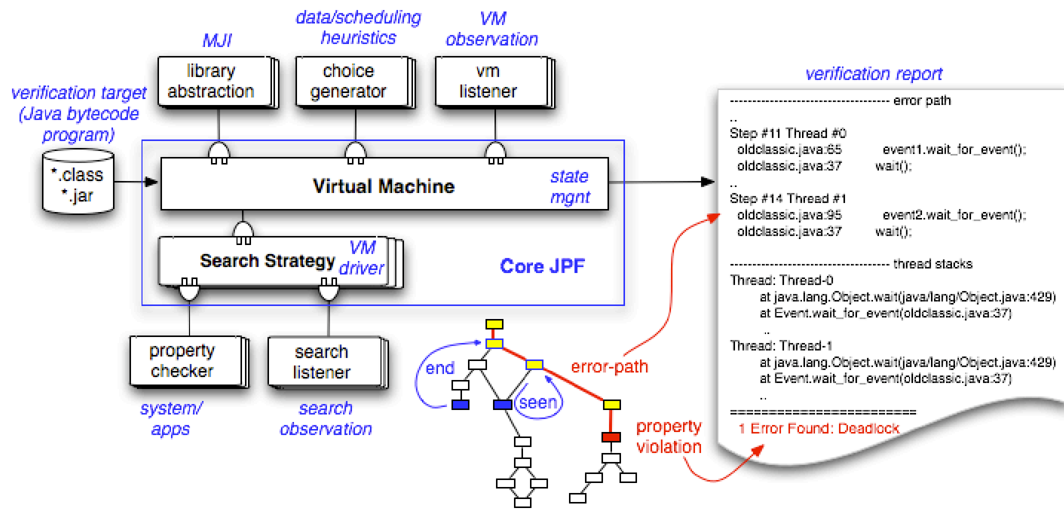

FIGURE 2.8 Fonctionnalité générale de Java PathFinder (Source [44])

La figure 2.8 illustre la fonctionnalité générale du model-checker JPF. Nous exposerons dans ce qui suit les diverses composantes de cet outil et leurs interactions décernant ainsi son apport remarquable dans le model-checking.

2.8.2 Que vérifie JPF ?

Le model-checker JPF a été primordialement conçu dans le but de détecter les erreurs de blocage "deadlocks" et des objets partagés "race conditions" rencontrées dans les systèmes dominés par le parallélisme.

De plus, JPF détecte les exceptions non traitées comme les violations d'assertions "AssertionErrors" concernant des assertions dans le code de la forme ASSERT <EXPRESSION BOOLÉENNE>, et les mauvais usages de la mémoire "NullPointerExceptions". Toutefois, l'utilisateur peut toujours inclure ses propres classes de propriétés (par exemple, des fichiers .ltl comprenant les formules LTL à vérifier) ou même peut écrire des classes étendant les observateurs pour implémenter d'autres vérifications de propriétés. L'implémentation de ces propriétés configurables sera présentée avec plus de détails dans la partie suivante de cette section.

Cependant, il faut noter que la machine virtuelle JPF ne peut pas exécuter du code natif ou spécifique à la plateforme utilisée. De ce fait, les librairies standards utilisées dans l'application sous le test sont restreintes. JPF ne supporte pas actuellement les librairies "java.awt" ni "java.net" et supporte quelques composantes de la librairie "java.io". Pour autant, il existe un mécanisme dans JPF rendant possible l'écriture des versions de librairies au moyen du MJI (Model Java Interface) discuté brièvement plus loin dans de chapitre.

2.8.3 Implémentation des propriétés

Il existe, dans JPF, deux catégories générales de vérification de propriétés :

- La classe "**gov.nasa.jpf.Property**" dont les instances encapsulent les propriétés désirées. Ces instances peuvent être configurées statiquement ou dynamiquement, et sont vérifiées par l'objet "Search" après chaque transition. Au cas où une implémentation de la méthode "*Property.Check(...)*" retourne faux et l'exécution doit se terminer, le processus de recherche est, à ce moment, arrêté et toutes les propriétés violées sont imprimées par la suite.

  Par défaut, JPF comprend trois classes génériques de "Property" :

1. "gov.nasa.jpf.jvm.NotDeadlockedProperty" : A chaque état non final, teste la présence d'un fil "Thread" toujours en exécution.
2. "gov.nasa.jpf.jvm.NoAssertionViolatedProperty" : teste si une expression d'assertion a été violée.
3. "gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty" : teste si une exception n'a pas été traitée dans l'application.

L'implémentation de propriétés supplémentaires est une démarche assez facile qui se limite à l'implémentation de l'interface "gov.nasa.jpf.Property".

– Les classes "**gov.nasa.jpf.SearchListener**" et "**gov.nasa.jpf.VMListener**" dont les instances peuvent être utilisées pour implémenter des vérifications plus complexes nécessitant de plus amples informations au niveau de l'exécution que dans la première catégorie.

2.8.4 Model-checking avec JPF

Afin de rendre possible le model-checking avec JPF, ce dernier doit supporter le non déterminisme. Effectivement, ceci est réalisé à l'aide de deux techniques majeures employées dans JPF : Le *rebroussement de chemins* et le *couplage d'états*.

Le *rebroussement de chemins* peut être visualisé dans la figure 2.8 par la flèche étiquetée "end" qui permet de restaurer des états d'exécution précédents afin de vérifier la présence d'autres choix de chemins d'exécution non explorés. Ce mécanisme peut être perfectionné en optimisant la collecte des états.

Le *couplage d'états* permet d'identifier les états déjà explorés lors de l'exécution. Ceci est utile dans la mesure où il n'y a pas d'intérêt de revisiter le même chemin d'exécution. Dans ce cas, JPF rebrousse chemin pour explorer le choix non déterministe non visité le plus proche. La flèche étiquetée "seen" dans la figure 2.8 reflète une conséquence de rebroussement de chemins dû au couplage d'états.

2.8.5 Avantages du model-checking avec JPF vis-à-vis du test

Afin de contourner le fameux problème d'explosion de l'espace d'états rencontré dans le model-checking de systèmes logiciels, JPF adopte les trois stratégies suivantes :

1. Stratégies configurables de recherche :
   Plusieurs heuristiques de recherche, incluant des heuristiques basées sur une couverture de test—par exemple, une couverture de conditions dans [18]—sont développées afin de guider la recherche du model-checker JPF pour identifier les erreurs plus efficacement c.-à-d. avec moins de ressources de calcul. Ceci est réalisé en concentrant la recherche d'autant plus sur la structure du programme à analyser que sur la propriété à vérifier. Cette recherche subsidiaire permet d'ordonner et de filtrer l'ensemble des états potentiels à explorer dépendamment de la pertinence de la propriété en question et également suivant la structure du programme.
2. Réduction du nombre d'états :
   C'est l'une des approches les plus performantes dans l'atténuation du problème d'explosion d'états et s'appuie sur une multitude de mécanismes. Parmi ces derniers, nous citons :
   - Les *générateurs de choix basés sur les heuristiques* (Choice Generator) dans la figure 2.8 qui confèrent au model-checker la possibilité de représenter un ensemble partiel—non complet—de choix dans un état d'exécution déterminé. L'exemple, qui illustre le plus l'avantage découlant de ce mécanisme, est mis en évidence dans l'utilisation des nombres à virgule flottante avec un comportement défini correspondant à une valeur de seuil, où il est impossible de générer toutes les valeurs possibles. Toutefois, dans le cadre de vérification du comportement du système, cet ensemble infini de choix est restreint à trois choix : plus petit, égal et plus grand que la valeur de seuil. Une fois la configuration de ces heuristiques est rendue possible, elles peuvent être étendues ou même adaptées aux besoins spécifiques de l'application.
   - La *réduction d'ordre partiel*—noté POR (Partial Order Reduction)—est le mécanisme le plus important dans la réduction de l'espace d'états, dans la mesure où des programmes concurrents sont impliqués. Cette réduction exploite la commutativité des transitions exécutées simultanément et aboutissant au même état dans des exécutions avec des ordres différents. De ce fait,

cette technique de réduction est la plus adéquate au sein de systèmes asynchrones. Pour atteindre cet objectif à la volée, la réduction d'ordre partiel dans le model-checker JPF utilise les codes objets Java ainsi que l'information d'accessibilité obtenue par le ramasse-miettes "garbage collector".

- *La délégation de l'exécution à la machine virtuelle hôte* : Comme mentionné plus haut dans ce manuscrit, JPF est une machine virtuelle qui roule "au-dessus" de la machine virtuelle hôte. Il serait optimal de déléguer l'exécution à cette dernière dans des composantes du programme non reliées à la propriété en question. Ceci est favorable vu que la machine virtuelle hôte ne collecte pas les états, ce qui constitue normalement un véritable "fardeau" dans l'exploration de l'infinité d'états au sein de la machine virtuelle JPF. L'interface de modèles Java MJI, illustrée dans la figure 2.8 et discuté un peu plus loin, représente un mécanisme correspondant à la communication entre ces deux machines, spécialement conçu pour simuler les entrées-sorties IO et les fonctionnalités d'autres librairies standards.
- L'*abstraction d'états* comme celle basée sur l'analyse des structures de données, développée dans une partie précédente dans ce chapitre, engendre une réduction considérable d'états.

3. Réduction dans la collecte d'états : Bien qu'elle ne reflète pas une mesure primaire à considérer pour alléger le problème d'explosion d'états, une collecte efficace d'états est obligatoire pour un model-checker logiciel. De ce fait, vu que les transitions d'états ne résultent pas en un grand changement au niveau des variables, le model-checker JPF utilise une technique qu'on appelle "state collapsing" ou réductions au niveau de l'état : Au lieu de collecter directement les valeurs modifiées, on procède à une réduction des ressources au niveau de la mémoire en collectant les indices dans des tables de hachage appropriées. Par conséquent, pour la comparaison d'états, JPF étend la réduction d'états en utilisant, dans le hachage, un seul nombre représentant l'identificateur de l'état ; d'où la réduction de la vérification d'égalité entre deux états à une simple comparaison d'entiers.

2.8.6 Structure haut niveau de JPF

Le model-checker JPF s'appuie sur deux abstractions majeures : la machine virtuelle VM (Virtual Machine) et l'objet "Search".

- *VM* est un générateur d'états spécifiques à l'environnement d'exécution. Au fil de l'exécution des instructions codes objets Java, *VM* génère des représentations d'états qui peuvent être :
  1. Comparés à des fins d'égalité (On évite ainsi la revisite du même état)
  2. Sous requête (pour s'informer sur les états des fils d'exécution, sur les valeurs des variables, etc.)
  3. Collectés
  4. Restaurés

  Dans le cadre de collaboration VM et "Search", il existe trois méthodes VM majeures :
  1. "forward" - génère l'état suivant, répond à la question : "est-ce que l'état généré a un successeur ?". Si oui, placer l'état en question dans une pile "backtrack" pour une restauration ultérieure efficace.
  2. "backtrack" - restaure le dernier état sur la pile "backtrack"
  3. "restoreState" - restaure un état arbitraire, pas nécessairement sur la pile "backtrack".

- L'objet "*Search*" est responsable de sélectionner l'état à partir duquel *VM* doit continuer. Ainsi, "*Search*" dirige *VM* soit à la génération de l'état suivant ("forward"), soit à rebrousser chemin jusqu'à un état préalablement généré ("backtrack"). Certes, on peut considérer les objets "*Search*" comme étant les "conducteurs" des objets *VM*. De plus, les objets "*Search*" configurent et évaluent les propriétés (par exemple, NotDeadlockedProperty, NoAssertionsViolatedProperty). Les implémentations primordiales de "*Search*" incluent le parcours en profondeur DFS et un parcours basé sur une file de priorité dont le paramètre est réglable et est basé sur la sélection de l'état le plus intéressant parmi la collecte de tous les successeurs d'un état donné.

Une implémentation de "*Search*" nous procure principalement une simple méthode de parcours incluant la boucle de base qui itère dans l'espace d'états pertinent jusqu'à la fin de son exploration, ou jusqu'à la violation d'une propriété [22].

2.8.7 Instrumentation des applications avec "Verify"

Vu que le model-checker JPF manipule des applications Java qui représentent des modèles d'autres systèmes, il s'avère nécessaire de pouvoir appeler des APIs JPF de l'application afin d'obtenir de l'information de JPF ou dans le but de diriger son éventuelle exécution. L'API de JPF se concentre sur la classe "**gov.nasa.jpf.jvm.Verify**" et comprend trois grandes catégories :

- Les *générateurs de données aléatoires* permettent de fournir des valeurs d'entrée non déterministes de façon à ce que JPF puisse systématiquement analyser toutes les valeurs pertinentes. On cite deux méthodes : "*Verify.randomBool*" pour la génération de variables booléennes et "*Verify.random(c)*" pour la génération d'entiers inférieurs ou égaux à *c*.
- Les *contraintes de parcours* sont les plus employées à des fins de contrôle du parcours de JPF. Cette catégorie nous semble pertinente dans la mesure où nous pouvons contracter l'espace d'états tout en limitant la vérification de JPF soit par le contrôle d'atomicité (via "*Verify.beginAtomic*" et "*Verify.endAtomic*"), soit par la restriction du parcours. Cette dernière technique, appliquée dans nos études de cas, est optimale dans la vérification de propriétés spécifiques à l'application où il nous est possible d' "élaguer" certaines valeurs non pertinentes à la propriété en perspective (via "*Verify.ignoreIf(data<someValue)*").
- L'*annotation d'états* peut aider JPF à l'identification d'un état pertinent du programme qui peut être, par la suite, utilisé pour des implémentations de "*Search*" (via "*Verify.interesting(data<someValue)*").

2.8.8 Extensibilité de JPF

L'extensibilité de JPF—le point fort majeur de ce model-checker—est décernée par les chercheurs dans le domaine de model-checking et de test. Cette section présente

brièvement les quatre mécanismes majeurs assignant à JPF cet atout :

1. Observateurs de recherche –Observateurs de la machine virtuelle
   Ces observateurs aident à étendre le modèle d'états interne de JPF et à ajouter des vérifications de propriétés plus complexes. De plus, ils jouent un rôle important dans la direction des recherches ou tout simplement dans la collecte des statistiques d'exécution.

   D'une part, les instances de l'interface "SearchListener"—dont une partie est illustrée dans la Figure 2.9—offrent l'opportunité de "surveiller" le processus de parcours de l'espace d'états, par exemple, en créant des représentations graphiques d'états à travers l'outil intégré **StateSpaceDot** , générateur de descriptions graphiques spécifiques à **GraphViz**.

```
public interface SearchListener {

 /* got the next state */
 void stateAdvanced (Search search);

 /* state was backtracked one step */
 void stateBacktracked (Search search);

 /* a previously generated state was restored
    (can be on a completely different path) */
 void stateRestored (Search search);

 /* JPF encountered a property violation */
 void propertyViolated (Search search);

 /* we get this after we enter the search loop, but BEFORE the
    first forward */
 void searchStarted (Search search);
```

   FIGURE 2.9 Interface "SearchListener"

   D'autre part, les instances de l'interface "VMListener"—illustrée dans la figure 2.10—sont utiles dans la "surveillance" de certaines instructions spécifiques à l'environnement d'exécution comme, par exemple, les instructions IF de Java dans l'analyse de couverture utilisée dans le cadre de nos travaux.

2. Le Model Java Interface MJI
   C'est un mécanisme qui permet de séparer et créer une communication entre l'exécution reliée aux états parcourus dans la machine virtuelle JPF et l'exécution non reliée aux états parcourus dans la machine virtuelle hôte (qui elle-même

```
public interface VMListener {
/* VM has executed next instruction
    (can be used to analyze branches, monitor PUTFIELD / GETFIELD and
    INVOKExx / RETURN instructions) */
 void instructionExecuted (VM vm);

 /* new Thread entered run() method */
 void threadStarted (VM vm);

 /* Thread exited run() method */
 void threadTerminated (VM vm);

 /* new class was loaded */
 void classLoaded (VM vm);

 /* new object was created */
 void objectCreated (VM vm);

 /* object was garbage collected (after potential finalization) */
 void objectReleased (VM vm);

 /* garbage collection mark phase started */
 void gcBegin (VM vm);

 /* garbage collection sweep phase terminated */
 void gcEnd (VM vm);

 /* exception was thrown */
 void exceptionThrown (VM vm);
}
```

FIGURE 2.10 Interface "VMListener"

exécute JPF). Ceci est avantageux en termes de réduction de l'espace d'états vu que ça représente des abstractions au niveau de la librairie standard.

3. Les générateurs de choix configurables Ils sont utiles afin d'implémenter des heuristiques spécifiques à l'application pour gérer le non déterminisme dominant dans la manipulation des données et les règles d'ordonnancement.

4. Les "usines de codes objets" ou "Bytecode Factories" Ils permettent de choisir entre différents modes d'exécution dans JPF. L'application la plus éminente de cette infrastructure est l'extension de JPF qui n'est autre que l'*exécution symbolique*. Cette exécution utilise l'interface "BytecodeFactory", plus spécifiquement l'instance "SymbolicInstructionFactory", afin de générer des cas de test concrets.

Cette extension de JPF sera exposée dans la prochaine section. Avant de conclure cette partie, nous présentons brièvement quelques applications du model-checker JPF. Dès sa conception, JPF a été surtout utilisé dans des systèmes concurrents dans le domaine de l'avionique. Il a été le pionnier dans la détection des "deadlocks" du système de contrôle du vaisseau spatial de NASA, et du robot d'exploration K9 consistant en 8000 lignes de code C++/Java. De plus, JPF a permis de vérifier le partitionnement du temps dans le système d'exploitation de DEOS suivant [23] dans une partie de 1000 lignes de C++ traduits en 1443 lignes de Java. Dans [24], Mehlitz démontre une approche de vérification des diagrammes d'états UML avec le code intégré dans les conditions ou gardes et les actions. Pour ce faire, Mehlitz établit une traduction d'UML en Java pour sa vérification ultérieure via le model-checker JPF. Des propriétés indépendantes et dépendantes de l'application (comme l'accessibilité à un état désiré) ont été vérifiées avec succès. Malgré la performance élevée du model-checker JPF, ce dernier génère en sortie un rapport de vérification en termes de lignes de code dans le programme modélisant le système en cours de vérification.

Dans le cadre de nos études de cas, nous ne désirons pas localiser l'erreur dans le code du programme, en d'autres termes, notre objectif ne touche pas à la détection d'erreurs de programmation. En revanche, la détection de situations—voire des contre-exemples—violant des propriétés générales et particulières à l'application représente notre but principal. D'où le recours aux extensions successives de JPF : JPF-SE et "Symbolic JPF".

2.8.9 Extensions de JPF : JPF-SE puis "Symbolic JPF"

JPF-SE représente une extension de JPF qui combine l'exécution symbolique avec le model-checking et la résolution des contraintes dans les programmes Java. De ce fait, JPF-SE utilise JPF dans le but de générer et d'explorer les chemins d'exécution symboliques tout en recourant à des procédures de décision déjà existantes pour manipuler les contraintes numériques.

L'architecture de JPF-SE [25] est illustrée dans la figure 2.11 : Les programmes

sont instrumentés afin de rendre l'exécution symbolique possible avec JPF. Les types concrets sont remplacés par les types symboliques correspondants alors que les opérations concrètes sont remplacées par des appels aux méthodes qui implémentent les opérations correspondant aux expressions symboliques [26].

A chaque fois qu'une condition de chemin est mise à jour, elle est vérifiée à l'aide d'une procédure de décision appropriée. Si cette condition de chemin n'est pas satisfiable, le model-checker rebrousse chemin comme dans le cas primaire de model-checking pur.

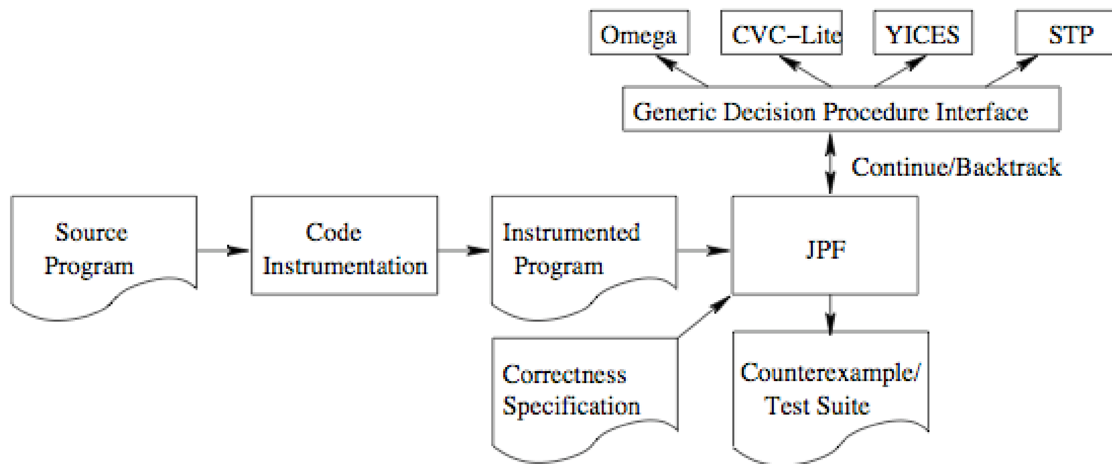

FIGURE 2.11 Architecture de JPF-SE (Source [25])

Parmi les procédures de décision utilisées dans JPF-SE, nous évoquons la libraire Omega qui supporte les contraintes entières linéaires et le solveur de contraintes RealPaver récemment intégré dans l'outil JPF-SE. L'importance de RealPaver réside dans le support des contraintes linéaires et non linéaires de nombres à virgule flottante. C'est l'adoption de ce solveur spécial en combinaison avec l'exécution symbolique dans le model-checking qui nous a primordialement incité à s'intéresser à JPF-SE. Le résultat d'une telle approche comporte une génération de cas de test en entrée à partir d'un modèle incluant des contraintes numériques non linéaires.

Dans [26], une présentation détaillée des techniques utilisées dans JPF-SE est suivie par une démonstration pratique de l'exécution symbolique au sein des programmes

Java séquentiels et concurrents.

Il faut noter que, dans [27], l'instrumentation de ces programmes est remplacée par un interpréteur de bytecodes non standard : On passe de JPF-SE à "Symbolic JPF". De plus, l'information symbolique est infiltrée dans le code à travers des attributs propagés dynamiquement durant l'exécution symbolique et associés aux variables de programmes et aux opérandes comme dans [27]. Basés sur ce principe, nous pouvons commencer l'exécution symbolique à tout moment et à partir de tout point du code. De plus, il est dorénavant possible de combiner l'exécution concrète à l'exécution symbolique [27].

L'une des applications éminentes, surtout dans le cadre de notre travail, est la génération automatique de cas de test en entrée et qui résulte en une couverture élevée du code, plus précisément, la couverture de chemins "path coverage"—avec des métriques de couvertures assez flexibles.

Conséquemment, quand le code source est disponible, il est possible de générer des cas de test en entrée qui satisfont un critère de test déterminé et ce, en utilisant "Symbolic JPF" pour trouver les chemins symboliques d'exécutions où le critère est satisfait. En écrivant le critère de test sous forme de propriété à vérifier dans JPF, ce dernier génère les contre-exemples représentant les exécutions qui satisfont ce critère. Les conditions de chemin, présentes dans les chemins symboliques d'exécution, sont ainsi résolues afin de trouver les valeurs concrètes des variables d'entrée et de changer les structures symboliques d'entrée en leurs équivalents concrets. Pour plus de détails sur l'outil Java PathFinder et ses extensions JPF-SE et "Symbolic JPF", consulter [44].

Sous le concept d'exécution symbolique avec le model-checking, la revue de littérature se concentre sur la détection d'erreurs dans les systèmes logiciels comme dans [26], ou sur la génération de cas de test en entrée satisfaisant une couverture de code dominé par des structures de données complexes comme dans [28], [29] et [30].

L'approche originale, adoptée dans nos deux études de cas, se caractérise ainsi par l'utilisation de "Symbolic JPF" dans le but de générer des cas de test en entrée pour la

vérification de systèmes numériques complexes à contraintes non linéaires modélisant un simulateur de vols d'un côté—première étude de cas—et des systèmes d'éditions collaboratives distribuées dominés par le contrôle de la réplication de données d'un autre côté—seconde étude de cas.

La première étude de cas fera ainsi l'objet du prochain chapitre.

Chapitre 3

PREMIÈRE ÉTUDE DE CAS : SIMULATEUR DE VOL

Notre première étude de cas consiste en un simulateur de vol, dans un contexte d'industrie aéronautique, qui appartient à la classe de systèmes embarqués réactifs. Ces systèmes comprennent naturellement une composante de contrôle à état fini avec des données numériques en entrée qui mesurent des quantités comme la vitesse de l'air WindVelocity, l'angle de rotation roll, de vision TrueHeading, etc.

Dans ces systèmes, les transitions d'états dépendent des prédicats, ou contraintes sur ces valeurs numériques. On manipule plus spécifiquement des systèmes à contraintes arbitraires et complexes sur des domaines finis et infinis.

Notre objectif consiste à développer un outil permettant d'interfacer le simulateur de vol modélisé à l'aide de tables de décision avec un model-checker approprié, afin d'en vérifier des propriétés pertinentes.

Dans ce chapitre, nous présentons dans une première section les tables de décision en mettant en relief les avantages de la notation tabulaire dans la modélisation de ce type de systèmes. Ensuite, le modèle global suivi le long de notre étude de cas sera présenté dans une seconde section. A des fins d'illustration, nous développons dans une troisième section les expérimentations de model-checking d'un exemple relativement simple dans le contexte d'industrie aéronautique. Finalement, la dernière section est consacrée à l'élaboration de notre contribution au sein du simulateur de vol.

3.1 Tables de décision

Dans le cycle de vie de tout système logiciel, les étapes amont du codage sont essentiellement vulnérables quant aux erreurs ce qui peut causer une grande influence sur la fiabilité, les coûts et la sécurité des systèmes. Par conséquent, les erreurs prenant naissance dans la phase des spécifications et des requis peuvent traîner jusqu'à la phase de conception. Une fois cette erreur détectée (si "heureusement" détectée), les ingénieurs doivent retourner à la première phase de spécification afin de résoudre ce problème avec des analyses supplémentaires.

Ceci ne cause pas seulement une perte de temps mais aussi la possibilité d'engendrer plusieurs autres erreurs dans les requis et les spécifications. De plus, une détection des erreurs dans des phases plus avancées du cycle est de loin plus coûteuse que celle dans des phases antérieures.

Si ces problèmes sont plus ou moins acceptables dans des systèmes non critiques du point de vue sûreté, ils ne le sont certainement pas dans les systèmes critiques.

Pour cette raison, il est nécessaire que les requis soient correctement spécifiés après discussion avec le client afin de générer des spécifications claires et exactes. D'où l'importance de plus en plus accrue des méthodes formelles dans la modélisation de systèmes. Les tables de décision représentent entre autres une solution adéquate à ce problème comme nous verrons dans ce qui suit.

3.1.1 Définition

Une table de décision est un tableau composé de lignes et colonnes et constitué de quatre éléments : les conditions et leurs différentes alternatives, les actions et leurs différentes valeurs. Cette notation tabulaire est privilégiée dans la modélisation de systèmes logiciels à multiples états et conditions d'entrée. Typiquement, ces tables de décision associent chaque entrée ou condition du système avec une ou plusieurs action(s) ou variable(s) de sortie.

Dans le cadre de notre travail, nous adoptons un format particulier de représentation de tables de décision propre à une entreprise aéronautique : Une table de décision est associée à chacune des actions ou transitions, et un système complet est générale-

Hazard Title: **Depressurization** Hazard Severity: Catastrophic					
	System States				
Hazard Condition	1	2	3	4	5
Relieve Overpressure Command Enabled and sent?	N	Y	Y	Y	Y
Module Isolated? (Hatch Closed?)	-	N	Y	Y	Y
Sensor A failed high? (Control Inhibit)	-	-	N	Y	Y
Sensor B failed high? (Control Inhibit)	-	-	-	N	Y
Resulting Actions					
Control Valve Opened				X	X
Control Valve Closed	X	X	X		
Isolation Valve Opened					X
Isolation Valve Closed	X	X	X	X	
No venting	X	X	X	X	
Depressurization - Venting (Hazardous condition)					X
Number of Faults	0	1	1	2	3

FIGURE 3.1 Table de décision décrivant le phénomène de "Depressurization Hazard"

ment modélisé par plusieurs tables de décision. En d'autres termes, chaque table de décision représente les variations des valeurs d'une seule variable de sortie (action) relativement à une combinaison donnée d'entrées (conditions).

La figure 3.1 illustre la table de décision modélisant le système complet décrivant le phénomène reconnu en avionique sous le nom de "Depressurization Hazard". En d'autres termes, il s'agit d'une modélisation d'un danger de dépressurisation dans un avion.

La figure 3.2 n'est autre qu'une seule table de décision associée à la variable de sortie ControlValveOpened parmi d'autres suivant notre représentation des tables mentionnée ci-haut. Nous retrouvons de même dans la figure 3.2 le format XML équivalent adopté pour représenter les tables de décision, particulièrement la table de

décision ControlValveOpened.

```
<Table> Control_Valve_Opened
    <HeaderRow>
        <HeaderInputCol> Relieve_Overpressure_Enabled
        </HeaderInputCol>
        <HeaderInputCol> Module_Isolated
        </HeaderInputCol>
        <HeaderInputCol> SensorA_Failed
        </HeaderInputCol>
        <HeaderInputCol> SensorB_Failed
        </HeaderInputCol>
        <HeaderOutputCol> Control_Valve_Opened
        </HeaderOutputCol>
    </HeaderRow>
    <Row>  State1
        <InputCell> Relieve_Overpressure_Enabled = Yes
        </InputCell>
        <InputCell> Module_Isolated = Yes
        </InputCell>
        <InputCell> SensorA_Failed = Yes
        </InputCell>
        <InputCell> SensorB_Failed = Yes
        </InputCell>
        <OutputCell> True
        </OutputCell>
    </Row> (...)
```

	Control_Valve_Opened				
	Input Variables				Output Variable
	Relieve_Overpressure _Enabled	Module_Isolated	SensorA_Failed	SensorB_Failed	Control_Valve_Opened
1	Y	Y	Y	Y	T
2	Y	Y	Y	N	T
3	Y	Y	N	-	F
4	Y	N	-	-	F
5	N	-	-	-	F

Figure 3.2 Table de décision et le format XML équivalent

Dans la figure 3.2, les variables sous l'entête "Input Variables" désignent les conditions ou les variables d'entrée du système alors que la variable sous l'entête "Output Variable" n'est autre que l'action correspondante. Dans notre format de tables, cette variable de sortie dénote en soi la table de décision considérée. En outre, chaque ligne représente une combinaison de variables d'entrées avec la sortie appropriée. Ceci n'est autre qu'un *état* défini du système.

Le tiret "-" représente une valeur aléatoire de la variable d'entrée décrivant de ce fait l'absence de la corrélation entre la valeur de cette variable particulière et l'action résultante dans l'état considéré.

Il est important de mentionner ici que toute table envisagée dans le contexte de notre étude de cas n'est pas nécessairement le résultat final de toute élimination des redondances, des contradictions et des situations impossibles dans les combinaisons conditions-actions. En effet, il existe d'autres outils formels comme SCR (Software Cost Reduction) dans [3] responsables de garantir ce résultat. Cette démarche sort alors du contexte de notre travail. Cependant, nous notons qu'elle peut affecter les cas de violation de propriétés discernés dans les étapes ultérieures de vérification. Dans ce cas, après l'analyse des contre-exemples générés par model-checking symbolique, les concepteurs du système pourront détecter facilement les erreurs provenant d'une construction inexacte au niveau des tables de décision.

En somme, chacune de ces tables représentera une seule action avec les différentes conditions et leurs valeurs respectives. Les colonnes de gauche représentent chacune des conditions et la colonne de droite, l'action associée à la table. Chacune des lignes représente un état du système : les valeurs des conditions ainsi que la valeur de l'action qui y est associée.

En recourant au même format de tables pour toutes les autres variables de sortie, nous aurons modélisé le système au complet.

Avant d'aborder le processus adopté dans le traitement de cet exemple de tables de décision, nous développons dans la partie suivante quelques avantages de la notation tabulaire ainsi que les outils existants pour leur manipulation.

3.1.2 Avantages de la notation tabulaire

Au sein de toute entreprise manipulant des systèmes logiciels exigeant un certain niveau de fiabilité et de sécurité plus ou moins considérable, la récupération des requis représentant le comportement général du système en question consiste en une phase excessivement délicate dans le cycle de vie de ce système. Toute erreur ou manque de précision à ce niveau est susceptible de générer des erreurs dans la modélisation du système pouvant même affecter sa conception.

De ce fait, l'utilisation des notations tabulaires au moyen de tables de décision permet à la fois une représentation concise et complète des requis regroupant ainsi les différentes conditions et les actions résultantes du système.

Les tables de décision, utilisées depuis plusieurs années dans les spécifications des systèmes logiciels, regroupent les requis en utilisant une notation tabulaire bien plus facile à lire que les expressions dans la logique d'attributs. De plus, ces tables permettent une base de communication précise et non-ambiguë entre les développeurs ce qui facilite les constructions indépendantes, les révisions, les modifications et l'analyse de sous-parties d'une large spécification.

A noter aussi que les notations tabulaires sont bien adaptables et extensibles : un ensemble de tables peut bel et bien remplacer, en termes de spécifications, un programme d'environ 250 000 lignes de codes Ada comme pour le "C-130J Flight Program" [31].

3.1.3 Traitement des tables de décision

Il existe actuellement une multitude d'outils de modélisation de systèmes sous formats tabulaires. Nous citons : SCR, Tablewise et RSML. Indépendamment du format choisi, plusieurs techniques ont été développées afin de traiter ces différentes tables de décision. Bien que la plupart se concentre sur la génération de codes, quelques-unes ont été connues pour des raisons de vérification de consistance ou d'uniformité "Consistency Checking" comme DETRAN [32]. Ces outils vérifient si les spécifications sont bien formées : correctes du point de vue syntaxe et type, sans dépendances circulaires, complètes (pas de manque dans le comportement désiré) et consistantes (pas d'ambiguïté dans le comportement).

Un autre exemple à très grande importance est le SCR* qui comporte un éditeur de spécifications pour la création et la modification des spécifications, un "Consistency Checker" pour la vérification de la bonne construction de ces spécifications comme mentionnée ci-haut et un simulateur pour l'exécution symbolique de spécifications afin de s'assurer que ça répond bien aux intentions des développeurs. Un model-checker interne a été ajouté par la suite et ce dans le but d'analyser les spécifications. Cette analyse a comme but de vérifier si une propriété critique donnée a été respectée ou non ("property violation"). Ceci est réalisable à l'aide d'un model-checker qui, donnant un contre-exemple dans le cas d'une violation de propriétés, permet au développeur d'identifier l'imperfection au niveau des spécifications.

Toutefois, la vérification réalisée à ce niveau sert uniquement à valider les propriétés indépendantes de l'application en question. Pour une vérification complète des propriétés du système tenant compte de l'application, un appel ultérieur à un model-checker devrait prendre place.

Par exemple, après l'utilisation des outils et des techniques de SCR*, l'utilisateur devrait appeler le model-checker explicite SPIN [33] ou le model-checker symbolique SMV pour compléter la vérification au niveau de l'application. Dans ce cas, une traduction entre les spécifications en notation tabulaire et le langage restreint des deux différents model-checkers est produite automatiquement par les outils de SCR*. Cette traduction automatique est assez primordiale vu qu'elle évite énormément les proba-

bilités de tomber dans une erreur, un phénomène assez courant si la traduction se fait à la main.

Cependant, les grandes limitations au niveau de SCR* découlent des limitations des langages inhérents aux deux model-checkers, Promela de SPIN et le langage restreint de SMV. En effet, bien que ces deux langages présentent leur propres avantages et inconvénients l'un vis-à-vis de l'autre, ils sont néanmoins limités en ce qui concerne les types de variables. Le langage Promela ne supporte que les variables de type booléen, byte ou entier alors que le langage de SMV ne supporte que celles de type booléen, énumération ou entier.

Par la suite, nous allons supposer que le simulateur de vol, modélisé sous forme de tables de décision, est préalablement passé (ou pourrait éventuellement l'être) à un outil comme SCR* afin de vérifier des propriétés indépendantes de l'application comme la complétude, la consistance et la non-redondance. De ce fait, toute vérification de propriétés indépendantes du système sort du cadre de notre travail. Par conséquent, notre travail sera concentré sur la vérification de propriétés dépendantes de l'application comme des propriétés d'accessibilité "reachability" ou de vivacité "liveness". L'implémentation de ces propriétés est discutée plus loin.

3.2 Modèle de notre étude de cas

Comment connecter les tables de décision au model-checker afin de vérifier toutes les propriétés d'un système quelle que soit la nature des variables en question ?

Cette question constitue la raison principale pour laquelle la conception d'un analyseur syntaxique a été prévue comme indispensable. En effet, ce dernier représentera une sorte de passerelle permettant d'interfacer entre deux types de représentations données.

Par conséquent, l'élaboration d'un analyseur syntaxique a été perçue comme un intermédiaire d'interconnexion efficace entre la notation tabulaire, procurée comme entrée d'une part, et le langage du model-checker considéré d'autre part. D'ailleurs, cet analyseur syntaxique récupère comme entrée des tables de décision avec un format prédéfini et servira pour générer un fichier d'entrée compatible avec le model-checker

en question. La démarche à suivre est illustrée dans la figure 3.3.

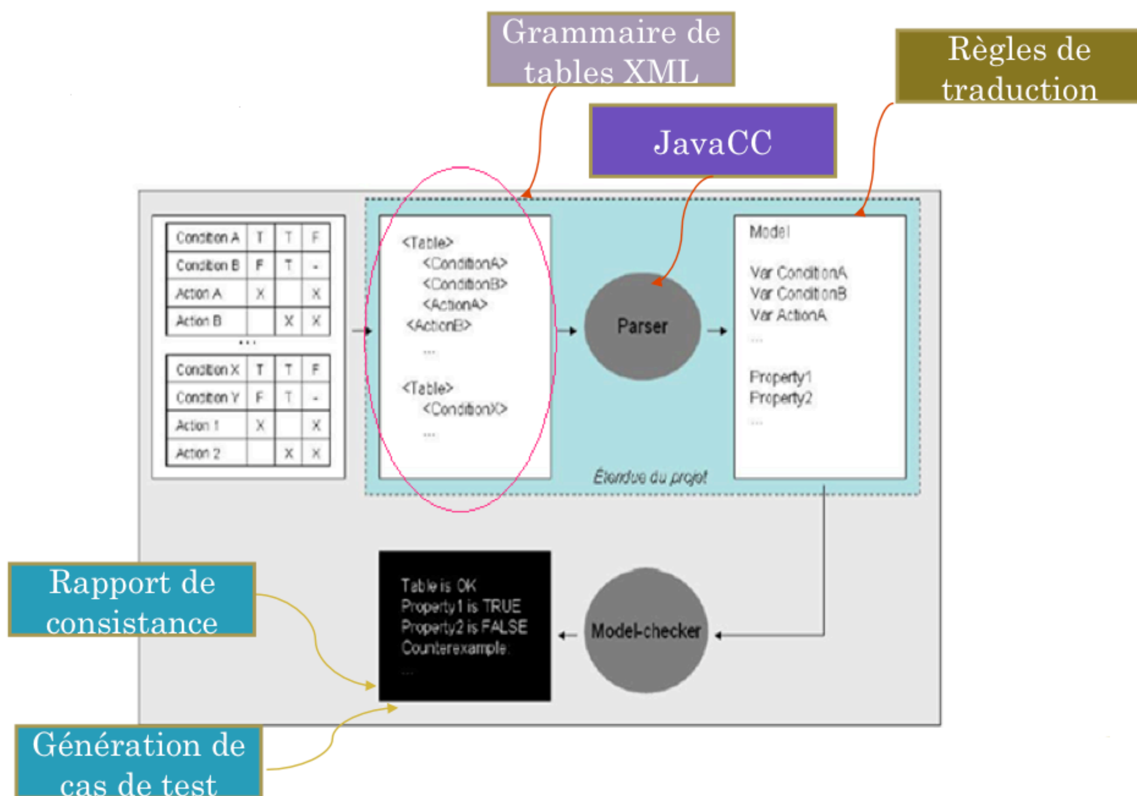

FIGURE 3.3 Processus de vérification des tables de décision

En première étape, nous procéderons à une traduction des tables en un format XML équivalent. Cette étape est forcément nécessaire dans la mesure où il est impossible d'analyser syntaxiquement des formats graphiques. En revanche, dans l'étendue d'application de notre étude de cas, nous considérons directement le fichier XML des tables de décision vu que la démarche de traduction du format graphique en format XML est simple et automatique.

Ensuite, nous construirons un analyseur syntaxique et lexical à l'aide de l'outil JAVACC, qui analysera notre fichier de tables et le traduira en un fichier d'entrée spécifique au model-checker utilisé. Dans le cadre de notre travail, nous opterons pour trois model-checkers différents, et conséquemment trois fichiers d'entrée différents.

Chacun d'entre eux pourra alors finalement valider notre modèle et les propriétés

qui y sont associées. Il faut noter que, dans cette étape finale, une simple instruction peut être ajoutée au code transmis au model-checker pour déclencher ultérieurement une génération "personnalisée" de cas de test comme nous verrons plus loin.

La section suivante sera consacrée à des fins d'illustration de notre démarche appliquée sur la modélisation d'un danger de dépressurisation dans un avion, présentée dans la section précédente.

3.3 Exemple illustratif dans le contexte d'industrie aéronautique

Nous abordons dans cette section un exemple illustratif à aspect théorique simplifié dans le contexte d'industrie aéronautique : "Depressurization Hazard". De ce fait, nous nous limitons à ce niveau à des variables booléennes et entières. Plus spécifiquement, nous considérons en total quatre variables d'entrée sous l'entête "Hazard Condition" et sept variables de sortie sous l'entête "Resulting Actions" suivant la figure 3.1. La démarche de traitement automatisé des tables de décision depuis le fichier des tables jusqu'à la vérification des propriétés sera détaillée dans un ordre logique dans ce qui suit.

3.3.1 De l'analyse syntaxique à la génération de code

Nous utilisons l'outil JavaCC comme outil de génération d'analyseurs lexicaux et de parseurs basé sur une grammaire prédéterminée. Notre choix d'utilisation de l'outil JavaCC est justifié par plusieurs critères notamment son support de grammaires déjà établies pour une majorité d'expressions arithmétiques (+, -, *, mod, etc.), sa manipulation de variables aux valeurs alphanumériques, entières et réelles et sa capacité d'analyse syntaxique granulaire.

Analyse syntaxique et lexicale suivie par une exploration de l'arbre généré

Après avoir conçu la grammaire correspondant au format équivalent XML des tables, l'étape d'analyse lexicale suit : Notre analyseur *lexical* généré par JavaCC traversera la chaîne de caractères en entrée—correspondant au fichier des tables— et la répartira en des sous-chaînes nommées "jetons" (Tokens) définis dans la grammaire. Ces jetons seront ainsi catégorisés selon leur type (nombres, identificateurs, etc.).

La séquence de ces jetons sera par la suite traitée par l'analyseur *syntaxique* pour la construction d'un arbre syntaxique abstrait AST des spécifications (Abstract Syntax Tree) représentant la structure du programme.

En résumé, lors de l'analyse syntaxique, on construit l'arbre syntaxique, puis une fois l'analyse syntaxique achevée, l'extraction de l'information de cet arbre s'effectue par des parcours de ce dernier. Cette méthode est très coûteuse en mémoire dû au stockage de l'arbre. Toutefois, l'avantage réside dans le fait que l'on n'est pas dépendant de l'ordre de visite des sommets de l'arbre syntaxique imposé par l'analyse syntaxique.

La procédure de l'analyse syntaxique et lexicale s'étendant depuis les spécifications sous forme de fichier XML de tables jusqu'à l'arbre syntaxique est illustrée dans la figure 3.4. Ce dernier sera analysé et parcouru par notre visiteur développé dans JavaCC.

Il reste à noter que, pour chaque nœud que nous désirons inclure dans l'arbre syntaxique, le symbole '# ' suivi du nom du nœud en question devraient être rajoutés à la règle de production dans notre grammaire.

L'extraction de l'information basée sur un *visiteur* "personnalisé" de l'arbre permettra de générer du code compatible avec le model-checker **NuSMV** dans une première étape, puis avec le model-checker **Alloy Analyzer** dans une étape suivante. C'est la *traduction dirigée par la syntaxe*.

Ces deux démarches seront détaillées séparément dans des sections consécutives plus loin dans ce chapitre.

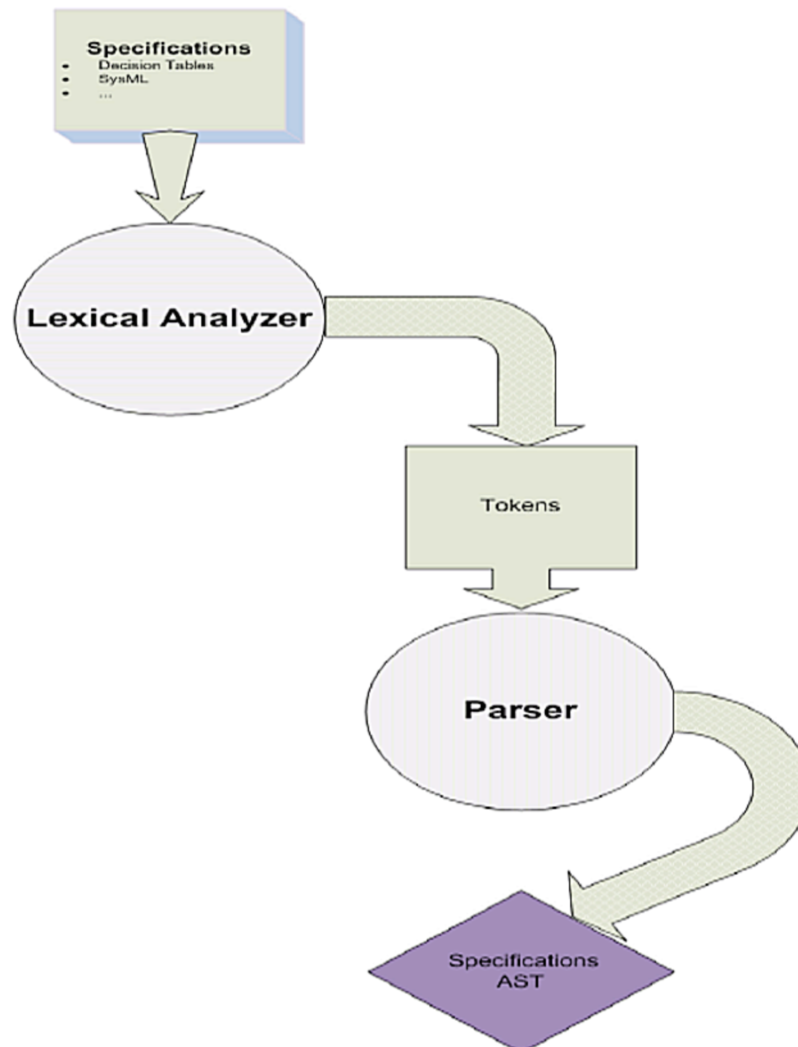

FIGURE 3.4 Analyse syntaxique et lexicale des tables de décision avec JAVACC

Extraction de l'information à partir de l'arbre

A ce niveau, notre analyseur syntaxique a parcouru le fichier de tables de décision prédéfini, et a construit une structure contenant les ensembles de variables visitées. Ceci résulte dans l'arbre syntaxique et lexical de la figure 3.5. On remarque que la hiérarchie "parent-fils" désirée entre les nœuds est conservée depuis la grammaire jusqu'à l'arbre syntaxique généré.

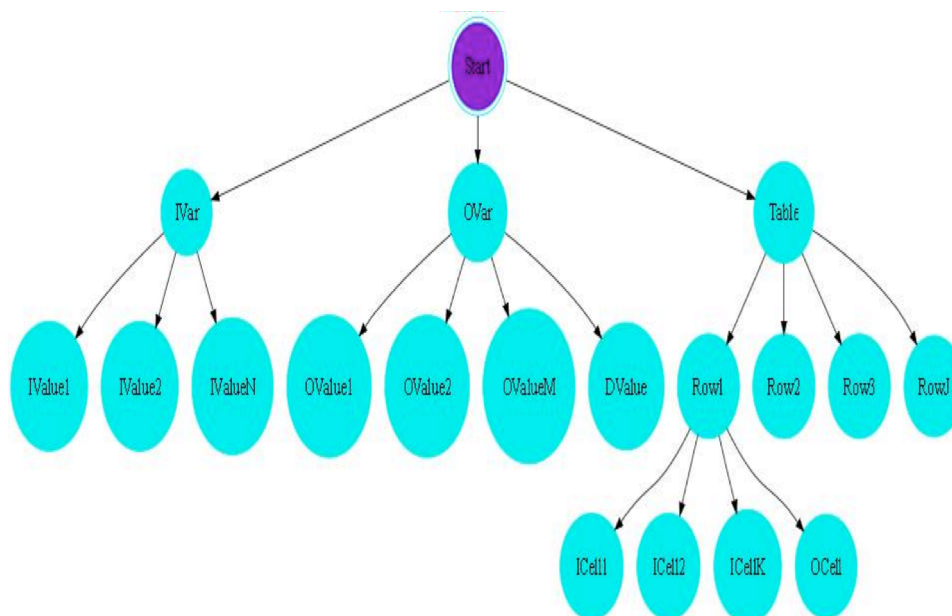

FIGURE 3.5 Arbre syntaxique et lexical généré

Nous dressons, dans la table 3.1, l'association entre les composantes des tables de décision et les nœuds construits dans l'arbre syntaxique. Nous notons que le nœud *ASTStart* est le nœud racine de l'arbre syntaxique généré quelle que soit la table de décision. De ce fait, aucune composante de la table ne correspond à ce dernier. De plus, nous notons qu'à des fins de simplication dans l'illustration de la figure 3.5, nous avons considéré une seule variable d'entrée, une seule variable de sortie et une seule table de décision.

L'extraction de l'information de l'arbre ainsi généré s'appuie sur la technique suivante : Par exemple, lorsqu'un nœud *ASTIVar* est identifié par le nœud défini *"# IVar"* dans la règle de production lors du parcours de la grammaire, le visiteur créé par JAVACC et accepté par le nœud de départ *ASTStart* et consécutivement par tous les enfants de ce nœud (incluant le nœud *ASTIVar*), appelle la méthode *public Object visit (ASTIVar node, Object data)* dans la classe du visiteur personnalisé étendant le visiteur générique créé par JAVACC.

La méthode *visit()* appelle à son tour la méthode *collect()* de la classe du nœud correspondant, ici la classe *ASTIVar*, qui a comme rôle de stocker les valeurs des jetons récupérés dans des vecteurs appropriés comme *IVarBuffer* dans ce cas. Ces

Tableau 3.1 Association entre les composantes des tables de décision et l'arbre syntaxique généré

Composante dans la table de décision	Nœud correspondant
Variable d'entrée	*ASTIVar*
Variable de sortie	*ASTOVar*
Valeur de la variable d'entrée	*ASTIValue*
Valeur de la variable de sortie	*ASTOValue*
Valeur initiale par défaut de la variable de sortie	*ASTDValue*
Nom d'une table de décision	*ASTTable*
Un état de la table de décision	*ASTRow*
Condition dans un état	*ASTICell*
Action résultante dans un état	*ASTOCell*

vecteurs seront par la suite manipulés dans des classes spécifiques pour la génération du code compatible avec le langage SMV du model-checker **NuSMV** d'une part et le langage Alloy du model-checker **Alloy Analyzer** d'autre part.

La figure 3.6 visualise un extrait de la classe correspondant à notre visiteur qui permet de récupérer les variables d'entrée *IVar* ainsi que leurs valeurs *IValue* respectivement dans les vecteurs *IVarBuffer* et *IValueBuffer*.

Cette démarche est identique pour les neuf autres nœuds définis dans la grammaire : *ASTStart*, *ASTIValue*, *ASTOVar*, *ASTOValue*, *ASTDValue*, *ASTTable*, *ASTRow*, *ASTICell* et *ASTOCell*. Ces nœuds ainsi collectés ne sont autres que les variables dans les tables de décision indispensables pour la génération de code à vérifier.

Par le biais de JJTree—un préprocesseur associé à JavaCC pour la construction de l'arbre—, l'ensemble de ces nœuds sont représentés par des classes correspondantes en Java qui étendent la classe de base *SimpleNode*. La classe *SimpleNode* permet de représenter le mode simple d'un visiteur, discuté ci-dessous.

Le choix du mode à utiliser dans un visiteur est possible grâce aux options ajoutées dans l'entête du fichier de la grammaire où $MULTI = true$ et *NODE_DEFAULT_VOID=true*. Ces deux options servent à générer un arbre syntaxique dépendamment du nœud défini dans la grammaire. De ce fait, les nœuds non définis ne seront pas introduits dans cet arbre généré. Par contre, par défaut et pour une valeur "false", elles permettent de générer un nœud au parcours de chacun d'entre eux : C'est le mode simple du visiteur. De plus, $VISITOR = true$ permet d'insérer

```
public Object visit(ASTStart node, Object data)
{
  data = node.acceptChildren(this, data);

  return data;
}

 public Object visit(ASTIVar node, Object data)
{
  IVarBuffer = node.collect(IVarBuffer);

  data = node.acceptChildren(this, data);
  IVarBuffer.set(index1, IValueBuffer);
  index1++;

  return data;
}

 public Object visit(ASTIValue node, Object data)
{
  IValueBuffer = node.collect();
  data = node.acceptChildren(this, data);

  return data;
}
```

FIGURE 3.6 Extrait de notre visiteur pour la construction de *IVarBuffer* et *IValueBuffer*

la méthode *jjtAccept()* dans les classes des nœuds et de générer une implémentation du visiteur avec une entrée pour chaque type de nœud utilisé dans la grammaire.

A ce niveau, l'étape d'extraction de l'information précédant la dernière phase de génération de code est achevée.

Génération de code

Pour une génération de code efficace, nous procédons à une collecte des variables dans des vecteurs de taille réglable. Cette récupération de variables est réalisée tout au long de l'extraction de l'information via la méthode *collect()* .

Ces vecteurs, dont une partie est visualisée dans la figure 3.7, permettent ainsi de garder le lien entre les nœuds parents et les nœuds fils lors du parcours de l'arbre généré dans la figure 3.5. Par conséquent, nous aurons comme résultat un vecteur global de tables $\{Table_1, Table_2, ..., Table_N\}$ représentant de la sorte le système au complet. Chaque élément de ce vecteur n'est autre qu'une table en soi—un autre vecteur—comprenant sous son premier indice le nom de la table "TableName" ; les

éléments dans les indices suivants représentent de manière similaire une séquence de vecteurs désignant ainsi les lignes ou états de chaque table $\{Row_1, Row_2, ...\}$. Finalement, chaque Row_i contient les valeurs d'entrée ou conditions de chaque état dans les $ICell_i$ ainsi que la valeur de sortie correspondante dans $OCell$.

Une fois l'information nécessaire récupérée des tables de décision, la génération de code devient une tâche simple dépendante du model-checker en question.

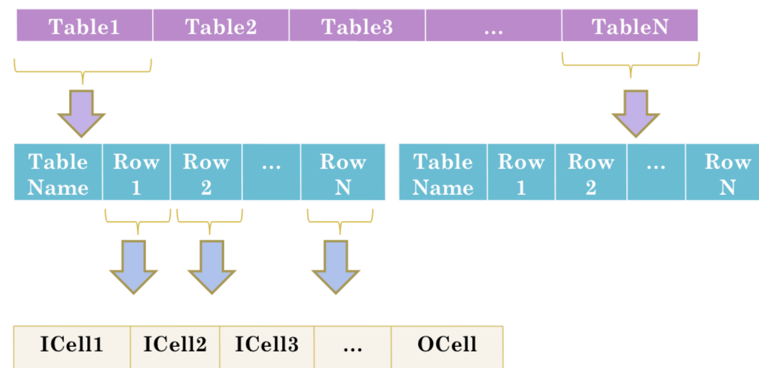

FIGURE 3.7 Récupération des variables après la visite de l'AST

3.3.2 Expérimentations avec NuSMV

Dans le cadre du model-checker **NuSMV**, nous décrivons brièvement le code généré pour la table de décision ControlValveOpened dans la figure 3.8 : Les conditions deviennent des variables d'entrée IVAR, et les actions des variables système VAR. La représentation des états est décrite dans le bloc ASSIGN où chaque variable de sortie est associée à un état initial "init" —la valeur par défaut— suivie par une liste de transitions "next". Chaque transition "next" attribue une valeur à une variable d'action en fonction de l'état courant du système.

Le dernier cas de transition, celui par défaut noté par '1' attribue la valeur UNDEF à la variable d'action. Cette valeur indéfinie est atteinte uniquement lorsqu'aucune transition n'est associée à l'état courant du système.

Particulièrement, dans ce simple exemple, nous procédons à la vérification de la complétude du système de la forme :

```
IVAR

Relieve_Overpressure_Enabled: {Yes, No};
Module_Isolated: {Yes, No};
SensorA_Failed: {Yes, No};
SensorB_Failed: {Yes, No};


VAR

Control_Valve_Opened: {True, False, UNDEF};
Control_Valve_Closed: {True, False, UNDEF};
Isolation_Valve_Opened: {True, False, UNDEF};
Isolation_Valve_Closed: {True, False, UNDEF};
No_Venting: {True, False, UNDEF};
Depressurization: {True, False, UNDEF};
Number_Faults: {0, 1, 2, 3, UNDEF};


ASSIGN

init(Control_Valve_Opened) := False;
init(Control_Valve_Closed) := False;
init(Isolation_Valve_Opened) := False;
init(Isolation_Valve_Closed) := False;
init(No_Venting) := False;
init(Depressurization) := False;
init(Number_Faults) := 0;
```

```
next(Control_Valve_Opened) := case
      Relieve_Overpressure_Enabled = Yes & Module_Isolated = Yes
& SensorA_Failed = Yes & SensorB_Failed = Yes & 1: True;
      Relieve_Overpressure_Enabled = Yes & Module_Isolated = Yes
& SensorA_Failed = Yes & SensorB_Failed = No & 1: True;
      Relieve_Overpressure_Enabled = Yes & Module_Isolated = Yes
& SensorA_Failed = No & 1: False;
      Relieve_Overpressure_Enabled = Yes & Module_Isolated = No
& 1: False;
      Relieve_Overpressure_Enabled = No
& 1: False;
      1: UNDEF;
esac;
        (...)

SPEC
AG(Control_Valve_Opened != UNDEF)

SPEC
AG(Control_Valve_Closed != UNDEF)

        (...)
```

FIGURE 3.8 Code SMV généré pour la table de décision CONTROLVALVEOPENED

AG (variable de sortie) != UNDEF

Cette propriété se catégorise sous le volet de propriétés indépendantes de l'application. Il est important de noter ici que ce genre de propriétés est automatiquement inclus dans le code généré et ce, après le mot-clé *SPEC* comme dans la figure 3.8. Ces propriétés permettent de vérifier que notre système n'entre pas dans des états de blocage, en d'autres termes des variables de sortie du système indéfinies.

L'introduction de propriétés supplémentaires et plus dépendantes de l'application est possible mais devrait être faite à la main dans le fichier généré en ajoutant la macro *SPEC* avant chaque propriété à vérifier.

En somme, une simple commande permet ainsi de traduire le fichier d'entrée de tables de décision en un code SMV dans un fichier de sortie qui servira comme entrée du model-checker après l'avoir lancé par la simple commande "NUSMV". La figure 3.9 démontre une trace d'exécution où toutes les propriétés sont satisfaites vérifiant ainsi la complétude du système. Nous remarquons que les deux dernières propriétés dépendantes de l'application et ajoutées à la main sont vérifiées automatiquement de façon identique à l'ensemble des propriétés de base.

```
[manaja@m4207-02 ]$ NuSMV
*** This is NuSMV 2.4.3 (compiled on Mon Nov 12 18:40:03 UTC 2007)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.

-- specification AG Control_Valve_Opened != UNDEF  is true
-- specification AG Control_Valve_Closed != UNDEF  is true
-- specification AG Isolation_Valve_Opened != UNDEF  is true
-- specification AG Isolation_Valve_Closed != UNDEF  is true
-- specification AG No_Venting != UNDEF  is true
-- specification AG Depressurization != UNDEF  is true
-- specification AG Number_Faults != UNDEF  is true
-- specification AG !(Control_Valve_Opened = True & Control_Valve_Closed = True)  is true
-- specification AG !(Isolation_Valve_Opened = True & Isolation_Valve_Closed = True)  is true
```

FIGURE 3.9 Trace d'exécution NUSMV

```
-- specification AG Number_Faults = 0  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 2.1 <-
  Control_Valve_Opened = False
  Control_Valve_Closed = False
  Isolation_Valve_Opened = False
  Isolation_Valve_Closed = False
  No_Venting = False
  Depressurization = False
  Number_Faults = 0
-> Input: 2.2 <-
  Relieve_Overpressure_Enabled = Yes
  Module_Isolated = No
  SensorA_Failed = No
  SensorB_Failed = No
-> State: 2.2 <-
  Control_Valve_Closed = True
  Isolation_Valve_Closed = True
  No_Venting = True
  Number_Faults = 1
```

FIGURE 3.10 Contre-exemple obtenu dans NUSMV

Une génération de contre-exemples peut être déclenchée dans le cas de violation de propriétés. Par conséquent, les figures 3.10 et 3.11 illustrent d'une part un cas régulier de contre-exemples montrant une trace justificative de la négation de l'assertion "$AG(NumberFaults = 0)$", et d'autre part, un cas de simulation d'erreurs lors de la phase de spécifications. Dans ce dernier contre-exemple, nous avons intentio-

nellement modifié le fichier de tables de décision, plus précisément nous avons retiré une condition donnée dans la table de décision ControlValveOpened pour voir si l'effet désiré est déclenché.

```
-- specification AG Control_Valve_Opened != UNDEF  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  Control_Valve_Opened = False
  Control_Valve_Closed = False
  Isolation_Valve_Opened = False
  Isolation_Valve_Closed = False
  No_Venting = False
  Depressurization = False
  Number_Faults = 0
-> Input: 1.2 <-
  Relieve_Overpressure_Enabled = Yes
  Module_Isolated = Yes
  SensorA_Failed = Yes
  SensorB_Failed = Yes
-> State: 1.2 <-
  Control_Valve_Opened = UNDEF
  Isolation_Valve_Opened = True
  Depressurization = True
  Number_Faults = 3
-- specification AG Control_Valve_Closed != UNDEF  is true
-- specification AG Isolation_Valve_Opened != UNDEF  is true
-- specification AG Isolation_Valve_Closed != UNDEF  is true
-- specification AG No_Venting != UNDEF  is true
-- specification AG Depressurization != UNDEF  is true
-- specification AG Number_Faults != UNDEF  is true
```

Figure 3.11 Contre-exemple simulant une erreur lors de l'élaboration des requis dans NuSMV

Bien que la vérification des tables de décision a été achevée avec succès à l'aide de NuSMV, notre intérêt de passer à un niveau d'abstraction plus élevé pour une vérification éventuelle d'un ensemble étendu de variables dans les tables de décision nous a incité à opter pour un model-checker différent. On passe du langage impératif de NuSMV au langage déclaratif d'Alloy Analyzer comme décrit dans la section suivante.

3.3.3 Expérimentations avec Alloy Analyzer

Alloy est le langage de modélisation de systèmes logiciels utilisé dans le model-checker Alloy Analyzer. Il permet une analyse automatique complète du système

modélisé et procure un outil de visualisation interprétant les solutions ou contre-exemples obtenus en fonction des assertions ou des prédicats définis.

Par ailleurs, il faut noter qu'ALLOY n'est pas déterministe : Lors de l'analyse des mêmes propriétés pour le même modèle, deux ou plusieurs solutions complètement différentes peuvent être générées à chaque exécution.

Dans cette section, nous appliquons la vérification automatisée des tables de décision du même exemple "Depressurization Hazard" rencontré dans la section précédente. L'ensemble des étapes s'étendant de l'analyse syntaxique et lexicale jusqu'à l'extraction de l'information de l'arbre généré reste inchangeable. La seule étape à modifier est dans la génération de code ALLOY au lieu de SMV. Pour ce faire, nous introduisons quelques notions préliminaires du langage déclaratif ALLOY.

```
open util/ordering[State] as ord

abstract sig Answer {}
one sig Yes, No extends Answer {}

abstract sig Bool {}
one sig True, False, UNDEF extends Bool {}

abstract sig InputVariable {}
one sig Relieve_Overpressure_Enabled, Module_Isolated, SensorA_Failed, SensorB_Failed extends InputVariable {}

abstract sig OutputVariable {}
one sig Control_Valve_Opened, Control_Valve_Closed, Isolation_Valve_Opened, Isolation_Valve_Closed,
    No_Venting, Depressurization extends OutputVariable {}

sig Number_Faults {}

abstract sig Value {}
one sig Three, Two, One, Zero, UNDEFINED extends Value {}

sig State {
      isequal: Number_Faults -> one Value,
      isequalto: OutputVariable -> one Bool,
      hasvalue: InputVariable -> one Answer
}
/*
 * In the initial state, all Output variables take their default value.
 */

fact initialState {
      let s0 = ord/first | all t : OutputVariable | ( t.(s0.isequalto) in False && Number_Faults.(s0.isequal) = Zero
}
```

FIGURE 3.12 Entête du code ALLOY généré

Dans ALLOY, on passe à un niveau d'abstraction plus élevé. On manipule des objets "Object", des ensembles d'objets "set", des signatures "sig". Dans notre exemple, les variables d'entrée *InputVariable* de même que les variables de sortie *OutputVariable* ainsi que leurs différentes valeurs possibles sont toutes déclarées comme des signatures dans l'entête du code ALLOY généré comme visualisé dans la figure 3.12.

Des relations existent entre ces différents objets ou signatures. Ces relations peuvent interconnecter plus que deux objets entre eux ; on parle ainsi de relations ternaires ou d'ordre supérieur à 2.

D'ailleurs, *State* est aussi une signature et représente chaque état du système à l'intermédiaire des relations respectives *hasvalue* reliant chaque variable d'entrée *InputVariable* à une et une seule valeur possible *Answer*, *isequalto* reliant chaque variable de sortie *OutputVariable* à une et une seule valeur possible *Bool* et enfin, la relation *isequal* reliant la variable de sortie exceptionnelle *Number_ Faults* à une et une seule valeur possible *Value*. A noter ici que *Number_ Faults* correspond à la variable pouvant prendre des valeurs numériques (pas nécessairement entières) contrairement aux autres variables de sortie de type booléen. De plus, le quantificateur **one** permet d'assurer que chaque variable dans notre système ne peut avoir qu'une seule valeur possible.

En outre, la représentation des variables n'est autre qu'une représentation symbolique d'objets et de relations entre eux. On envisage dans ce cas un niveau d'abstraction plus élevé préférable dans la modélisation de systèmes compliqués à divers états.

Après la définition des variables de notre système par le biais de signatures et de relations entre elles, nous entamons la description du comportement du système, en d'autres termes la représentation des différentes transitions décernant ses états différents. C'est à l'aide des *fact* (voir chapitre 2, section 6) que cette représentation est possible dans ALLOY. La première transition d'états correspondant à une combinaison particulière des conditions d'entrée est illustrée dans la figure 3.13. Par définition, l'utilisation des *fact* impose au système des contraintes définissant son comportement. Toute solution ou contre-exemple ne répondant pas à ces contraintes est de ce fait ignoré(e) vu que ça ne répond pas aux critères visant l'analyse désirée. Par consé-

quent, les *fact* jouent un rôle primordial dans la limitation de l'espace de recherche.

Il est important de noter ici qu'au niveau du modèle ALLOY, il s'avère essentiel de rajouter la notion d'ordre entre les états dans les transitions. Par exemple, dans les états *State* de notre modèle, nous introduisons la notion d'état successeur depuis un état *s* à un état *s' :ord/next[s]*. Cette relation d'ordre a été mise en œuvre en appelant un module déjà existant et importé à l'aide de la commande initiale *open util/ordering[State] as ord*.

```
/*
 * State 1
 */
 fact stateTransition1 {
    all s: State, s': ord/next[s] |
        { Relieve_Overpressure_Enabled.(s.hasvalue) = Yes
          Module_Isolated.(s.hasvalue) = Yes
          SensorA_Failed.(s.hasvalue) = Yes
          SensorB_Failed.(s.hasvalue) = Yes
        } => Control_Valve_Opened.(s'.isequalto) = True &&
             Control_Valve_Closed.(s'.isequalto) = False &&
             Isolation_Valve_Opened.(s'.isequalto) = True &&
             Isolation_Valve_Closed.(s'.isequalto) = False &&
             No_Venting.(s'.isequalto) = False &&
             Depressurization.(s'.isequalto) = True &&
             Number_Faults.(s'.isequalto) = Three
 }
```

FIGURE 3.13 Représentation d'une transition dans ALLOY

La figure 3.13 représente une combinaison spécifique des variables d'entrée (toutes les variables d'entrée sont égales à "Yes") pour laquelle nous obtenons les valeurs des variables de sortie, chacune étant associée à une table de décision déterminée.

En procédant similairement dans la représentation de toutes les combinaisons restantes par des *fact*, nous aurons modélisé le système au complet.

Finalement, nous procédons à la formulation des propriétés à vérifier dans le même langage. Dans le cadre de notre exemple, on se contente de vérifier automatiquement des propriétés indépendantes de l'application comme dans la section précédente. De ce fait, nous souhaitons prouver que notre système n'entre pas dans des états de blocage. Ces états sont représentés par des valeurs indéfinies des variables de sortie, respectivement *UNDEF* pour toutes les variables de sortie booléennes et *UNDEFINED* pour la variable de sortie numérique *Number_ Faults*. Il existe deux moyens possibles de

vérification dans ALLOY : les prédicats *pred* et les assertions *assert* comme dans la figure 3.14.

Par définition, un prédicat *pred* est semblable à une fonction ayant un nom, pouvant prendre des arguments et dont la valeur de retour est une valeur booléenne. Pour vérifier un prédicat, nous appelons *run* suivi par le nom du prédicat et le nombre de signatures qu'on désire considérer dans l'exécution en cours—ce dernier critère limite la portée de l'analyse. Ainsi, un prédicat *invalidState* est envisagé dans notre exemple dans la figure 3.14. Ce prédicat affirme que, pour tous les états du système, il existe au moins une variable de sortie indéfinie. Si une instance du prédicat est trouvée, nous aurons la possibilité de visualiser la trace à l'aide de l'outil de visualisation intégré dans l'outil ALLOY ANALYZER. Dans notre exemple, aucune solution dénotant cette condition indésirable n'a été trouvée telle que annotée dans la figure 3.15.

```
pred invalidState {
 all s: State | some t : OutputVariable | t.(s.isequalto) = UNDEF || Number_Faults.(s.isequal) = UNDEFINED
}
run invalidState for 6 but exactly 1 Number_Faults

assert nonblocking {
      all t: OutputVariable | all s: State | t.(s.isequalto) in (Bool - UNDEF) && Number_Faults.(s.isequal)
in (Value - UNDEFINED)
}
check nonblocking for 6 but exactly 1 Number_Faults
```

FIGURE 3.14 Représentation des prédicats et des assertions dans ALLOY

A la différence des prédicats *pred*, une assertion *assert* suppose que les expressions et les déclarations qui en découlent sont correctes. Chaque assertion possède un nom qui la distingue pour pouvoir l'identifier lors de la vérification avec le mot-clé *check* qui a comme rôle de s'assurer de l'exactitude de l'*assert* correspondant. Dans la figure 3.14, l'assertion *nonblocking* assume que le système considéré n'aboutit jamais à un état de blocage pour toutes les variables de sortie et pour tous les états définis. Dans notre exemple, aucun contre-exemple n'est généré ce qui permet de dire que notre système fonctionne correctement sans aucun état de blocage. Ceci est annoté dans la figure 3.15.

2 commands were executed. The results are:
#1: No instance found. invalidState may be inconsistent.
#2: No counterexample found. nonblocking may be valid.

FIGURE 3.15 Vérification de la complétude de notre système dans ALLOY

Nous pouvons simuler des erreurs dans la phase de modélisation dans le langage ALLOY de façon similaire que dans SMV, et ce en retirant la condition où toutes les variables d'entrée sont égales à "Yes". Dans ce cas, l'assertion *nonblocking* résulte en un contre-exemple dans la figure 3.16. Comme prévu, nous pouvons constater que la condition où toutes les variables d'entrée sont égales à "Yes" aboutit à une négation de l'assertion *nonblocking* dénotant des valeurs indéfinies des variables de sortie.

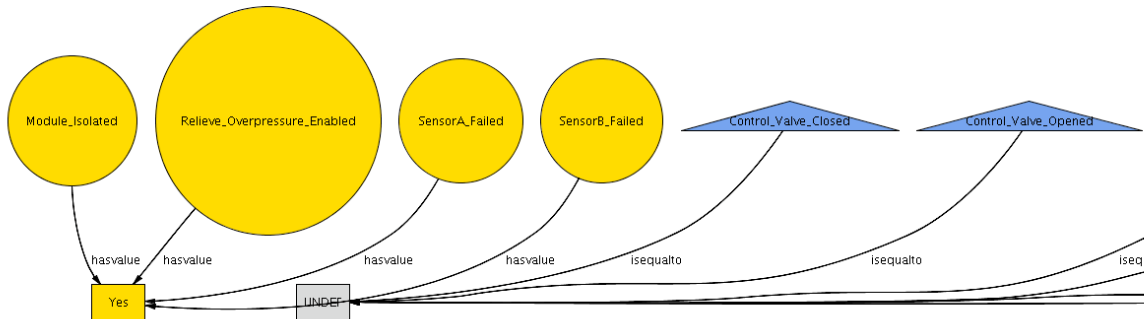

FIGURE 3.16 Contre-exemple dans ALLOY

La vérification automatisée des tables de décision du système "Depressurization Hazard" a été réalisée avec succès au moyen de l'outil ALLOY ANALYZER. Toutefois, notre motivation à privilégier l'abstraction inhérente au langage ALLOY n'a nullement remédié au problème d'analyse des systèmes numériques complexes à domaines infinis comprenant entre autres les transformations trigonométriques.

Par exemple, en utilisant les fonctions *fun* dans ALLOY, nous pouvons définir une fonction *sin*, qui associe le domaine des réels à l'intervalle [-1 ;1]. Cependant, la limitation dans le langage ALLOY réside principalement dans l'absence de la notion

de variables aux domaines infinis comme les variables réelles. En raison de la représentation symbolique dans ALLOY découlant du niveau d'abstraction élevé, les seules variables numériques supportées sont les variables entières. Or, les variables réelles ainsi que les opérations les manipulant sont indispensables dans la modélisation du comportement du simulateur de vol. Ce critère nous amène à l'impossibilité d'utiliser le model-checker ALLOY ANALYZER dans le cadre de notre étude de cas.

3.4 Étude de cas : Simulateur de vol

Au terme de ce travail, nous visons la vérification automatisée de notre étude de cas : Le simulateur de vol. En raison du nombre élevé de conditions et d'états découlant de ce type de système numérique, sa modélisation est très appropriée à l'aide de tables de décision. La figure 3.17 illustre l'ensemble de ces tables à vérifier.

Climb_Dive_Angle	
Inputs	Outputs
Velocity_body_x	Climb_Dive_Angle
> 0	Pitch - (FPM_el * cos(Roll)) + (FPM_az * sin(Roll))
Otherwise	unknown

Ladder_Mode		
Inputs		Outputs
Weight_on_wheels	Velocity_body_x	Ladder_Mode
VRAI	< 60 knots	PITCH_LADDER_TAKEOFF
	otherwise	CLIMB_DIVE_LADDER
FAUX	-	CLIMB_DIVE_LADDER

FPM_az : Flight Path Marker azimuth		
Inputs		Outputs
Ladder_Mode	Velocity_body_x	FPM_az
PITCH_LADDER_TAKEOFF	-	Pitch * sin(Roll)
CLIMB_DIVE_LADDER	> 0	tan(-1) (velocity_body_y / velocity_body_x)
	otherwise	unknown

FPM_el : Flight Path Marker elevation		
Inputs		Outputs
Ladder_Mode	Velocity_body_x	FPM_el
PITCH_LADDER_TAKEOFF	-	Pitch * cos(Roll)
CLIMB_DIVE_LADDER	> 0	tan(-1) (velocity_body_z / velocity_body_x)
	otherwise	unknown

Sig_Al : Signal Alarm			
Inputs			Outputs
FPM_az : Flight Path Marker azimuth	FPM_el : Flight Path Marker elevation	Climb_Dive_Angle	Sig_Al
unknown	-	-	VRAI
-	unknown	-	VRAI
-	-	unknown	VRAI
!=unknown	!=unknown	!=unknown	FAUX

FIGURE 3.17 Tables de décision modélisant le simulateur de vol

3.4.1 Présentation de notre étude de cas

L'ensemble des tables de décision a été annexé à l'ensemble des variables les délimitant dans un même fichier fourni à notre disposition pour des vérifications ultérieures. A ce niveau, nous présentons les variables et les opérations de notre simulateur de vol illustrées dans la figure 3.18 sans toutefois développer l'aspect aéronautique inhérent à ces variables.

Variables d'entrée Notre système se base sur sept variables d'entrée ou conditions d'entrée en total :

1. La variable booléenne *WeightOnWheels* suivie des trois angles :
2. La variable réelle *Pitch*
3. La variable réelle *Roll*
4. La variable réelle *TrueHeading* et finalement les 3 vitesses directionnelles :
5. La variable réelle *DownVelocity*
6. La variable réelle *EastVelocity*
7. La variable réelle *NorthVelocity*

Ces variables dénotent les spécifications relatives aux caractéristiques de vol de l'avion. Dans la figure 3.18, les variables sous l'entête *(Input Variables)* représentent ces variables d'entrée avec les intervalles respectifs de valeurs possibles. Pour les angles, ces intervalles sont [0,360] pour dénoter les angles en degrés Celsius alors que pour les vitesses, les intervalles sont [-270, 270] pour dénoter la vélocité de l'avion en unité de nœud qui est une unité de vitesse utilisée en navigation maritime et aérienne (1 nœud correspond à 1 mille marin par heure, soit exactement 1,852 km/h).

Variables intermédiaires Basé sur les variables d'entrée mentionnées ci-haut, notre système procède au calcul des tables de décision—voire les variables de sortie—tout en utilisant des variables intermédiaires exprimées en fonction de ces variables d'entrée. Ces variables sont illustrées dans la figure 3.18 sous l'entête *(Calculated Variables)* et leur calcul est présenté dans une colonne adjacente à celle de leur définition. Elles sont au nombre de douze, notamment :

- Les neuf cosinus directionnels naturellement dans les intervalles [-1,1] : *CXX*, *CXY*, *CXZ*, *CYX*, *CYY*, *CYZ*, *CZX*, *CZY* et *CZZ*. Ces variables sont ainsi exprimées en fonction des angles en entrée soit *Pitch*, *Roll* et/ou *TrueHeading*.
- Les trois vitesses correspondant aux trois directions de l'espace X, Y et Z : *VelocityBodyX*, *VelocityBodyY* et *VelocityBodyZ*. De même, ces variables sont en fonction des neuf cosinus directionnels ainsi que des trois vitesses directionnelles *DownVelocity*, *EastVelocity* et *NorthVelocity*. Identiquement aux vitesses directionnelles, ces variables doivent appartenir à l'intervalle [-270,270].

Par conséquent, après leur calcul, les variables d'entrée ainsi que les variables intermédiaires serviront à représenter ensemble les conditions d'entrée des tables de décision.

Nous remarquons, dans la figure 3.17, la présence des valeurs aléatoires '-' de ces variables indiquant ainsi l'interdépendance entre ces variables et la variable de sortie en question.

Variables de sortie Finalement, une fois ces variables intermédiaires calculées, le calcul des tables de décision est rendu possible et ce, en respectant un ordre bien déterminé en raison de la corrélation entre les tables. De ce fait, une variable de sortie correspondant à une table de décision spécifique peut représenter l'une des conditions d'entrée d'une autre table de décision. Dans notre système, par exemple, la variable de sortie *LadderMode* est passée comme l'une des conditions dans les deux tables de décision *FPM_ az* et *FPM_ el*.

Sous le principe de la hiérarchie de ces tables, nous décrivons les variables de sortie dans le même ordre de leur calcul. Il s'agit plus spécifiquement des quatre variables illustrées dans la figure 3.18 sous l'entête *(Derived Variables)* :

1. Une énumération *LadderMode* qui peut prendre deux valeurs constantes possibles : *PitchLadderTakeoff* et *ClimbDiveLadder* suivie des 3 angles sous les intervalles [0,360]
2. *FPM_ az*, *FPM_ el* avec le même ordre de calcul et finalement
3. *Climb_ Dive_ Angle*

Il faut noter que nous avons éventuellement rajouté la table de décision *Signal Alarm* dans la figure 3.17 qui représente la variable de sortie booléenne *Sig_ Al* dans le but de vérifier des propriétés de vivacité dans les états d'erreur. Cette table de décision est la dernière dans le calcul vu qu'elle est fonction des 3 variables de sortie *FPM_ az*, *FPM_ el* et *Climb_ Dive_ Angle*.

(Input Variables)	
PITCH : 0..360;	Aircraft flight specifications
ROLL : 0..360;	
TRUE_HEADING : 0..360;	
WEIGHT_ON_WHEELS : {TRUE, FALSE};	
DOWN_VELOCITY : -270..270;	
EAST_VELOCITY : -270..270;	
NORTH_VELOCITY : -270..270;	
(Calculated Variables)	
CXX : -1..1; -- directional cosines	cos(true_heading) * cos(pitch)
CXY : -1..1;	sin(true_heading) * cos(pitch)
CXZ : -1..1;	- sin(pitch)
CYX : -1..1;	cos(true_heading) * sin(pitch) * sin(roll) - sin(true_heading) * cos(roll)
CYY : -1..1;	sin(true_heading) * sin(pitch) * sin(roll) + cos(true_heading) * cos(roll)
CYZ : -1..1;	cos(pitch) * sin(roll)
CZX : -1..1;	cos(true_heading) * sin(pitch) * cos(roll) + sin(true_heading) * sin(roll)
CZY : -1..1;	-cos(true_heading) * sin(roll) + sin(true_heading) * sin(pitch) * cos(roll)
CZZ : -1..1;	cos(pitch) * cos(roll)
VELOCITY_BODY_X : -270..270;	CXX*north_velocity + CXY*east_velocity + CXZ*down_velocity
VELOCITY_BODY_Y : -270..270;	CYX*north_velocity + CYY*east_velocity + CYZ*down_velocity
VELOCITY_BODY_Z : -270..270;	CZX*north_velocity + CZY*east_velocity + CZZ*down_velocity
(Derived Variables) --> see Decision_Tables tab	
CLIMB_DIVE_ANGLE : 0..360;	Climb Dive Angle for placement of climb dive ladder
LADDER_MODE : {PITCH_LADDER_TAKEOFF, CLIMB_DIVE_LADDER};	
FPM_EL : 0..360;	Flight Path Marker elevation
FPM_AZ : 0..360;	Flight Path Marker azimuth
boolean Sig_Al	Signal Alarm

FIGURE 3.18 Variables et opérations caractérisant le simulateur de vol

3.4.2 Implémentation des propriétés à vérifier

Par définition, une table de décision est **complète** s'il existe une description du comportement du système—à travers ses actions—pour toutes les combinaisons possibles de conditions d'entrée.

Une table de décision est **inconsistante** s'il y existe au moins une paire d'états inconsistants. D'ailleurs, deux états d'une même table sont dits inconsistants si et seulement si ces deux états aboutissent à des actions différentes pour les mêmes conditions d'entrée. L'inconsistance peut aisément s'infiltrer dans les tables de décision en raison de la complexité des spécifications.

Une table de décision est **redondante** si et seulement si elle contient deux états totalement identiques en termes de conditions et d'actions correspondantes. L'introduction des valeurs aléatoires '-' des conditions d'entrée dans une table peut augmenter son risque de redondance.

Nous supposons, dans le cadre de notre travail, que les tables de décision dans la figure 3.17 sont complètes, consistantes et non redondantes. Par conséquent, nous supposons que la validation des spécifications en termes de la construction et de la cohérence du système est achevée a priori.

En revanche, nous cherchons à vérifier des propriétés plus intrinsèques au système modélisé préalablement impossibles à générer automatiquement : L'*accessibilité* (Reachability) et la *vivacité* (Liveness).

Accessibilité En premier temps, nous désirons vérifier si lors de l'exécution, notre système atteint un état critique dans son espace d'états. Dans le cadre de notre travail, les états que nous désirons surveiller sont ceux caractérisés par des variables de sortie inconnus *unknown*. Ainsi, en formalisme de logique temporelle CTL, nous pouvons annoter la propriété à vérifier comme suit :

$$EF(\varphi) \equiv AG(\neg\varphi) \tag{3.1}$$

où φ n'est autre que l'assertion suggérée.

Par exemple, φ peut s'exprimer sous les 3 conditions suivantes :

- "$FPM_az == unknown$"
- "$FPM_el == unknown$"
- "$Climb_Dive_Angle == unknown$"

Vivacité En second temps, nous désirons vérifier, que dans un état d'erreur préalablement défini, une alarme est éventuellement déclenchée. Cette propriété de vivacité est formulée en CTL par le suivant :

$$EtatErreur \Rightarrow AF(alarme) \tag{3.2}$$

La table de décision *Sig_ Al* dans la figure 3.17 a été particulièrement rajoutée à l'ensemble des tables de décision afin de pouvoir vérifier cette classe de propriétés.

Nous implémenterons plus loin ces propriétés d'accessibilité et de vivacité dans le code généré.

A ce niveau, le simulateur de vol a été présenté en termes de variables et d'opérations modélisant son comportement. De même, l'ensemble des propriétés pertinentes à la vérification de notre système a été évoqué. Nous développons dans ce qui suit les différentes méthodologies envisagées pour la vérification du système en question.

3.4.3 Expérimentations avec JForge

Basé sur le même principe d'analyse syntaxique et lexicale des tables de décision, d'extraction de l'information de l'AST généré et de la génération de code comme dans l'exemple illustratif de la section précédente, l'interfaçage entre les tables de décision et le model-checker adéquat est rendu possible.

Toutefois, la difficulté de l'analyse du simulateur de vol provient principalement des limitations au niveau du model-checking. Afin de rendre possible cette analyse, différentes abstractions et modifications au niveau du fichier d'entrée du model-checker—le code généré—sont nécessaires. A ce niveau, nous retenons les mêmes étapes que précédemment mais nous y apporterons quelques modifications avec un choix différent de model-checkers.

Dans les phases d'analyse syntaxique et lexicale, la grammaire et les visiteurs générés s'appuyant sur les fichiers des tables de décision peuvent être élaborés afin de permettre l'analyse de systèmes de nature plus complexe : Des expressions de conditions plus complexes sont intégrées dans ces nouvelles tables qui seront ainsi aptes à modéliser le simulateur de vol. En plus des variables booléennes et entières modélisant les tables précédentes, ce modèle comporte notamment des variables réelles ainsi que des transformations arithmétiques servant à modéliser les trajectoires selon les angles, vitesses et autres variables de l'avion. Par ailleurs, ce modèle implique la manipulation d'expressions trigonométriques composées de fonctions *sinus*, *cosinus*,

arctangente, etc.

De ce fait, modéliser un système numérique à contraintes réelles et expressions arithmétiques et trigonométriques complexes nécessite une réadaptation de la grammaire en JavaCC pour intégrer cette vaste gamme d'expressions arbitraires.

Nous supposons dans cette partie que les modifications apportées à la grammaire sont effectuées avec succès résultant en une génération de code dans un langage adéquat au model-checker envisagé. Ces modifications seront détaillées dans la partie suivante.

Comme résultat, nous nous concentrons directement sur l'implémentation des propriétés dans le code ainsi généré pour s'attaquer au problème du model-checking ultérieur.

Afin de tirer profit du langage Java supportant la classe "Math" pour la modélisation des transformations arithmétiques et trigonométriques d'une part et de la nature des spécifications basées sur un langage similaire au langage Alloy exploité antérieurement d'autre part, nous avons opté pour l'outil d'analyse statique JForge.

Cet outil de vérification de code permet d'analyser le modèle Java qui n'est autre que notre code généré contre des spécifications écrites dans un langage relationnel de premier ordre JFSL (JForge Specification Language) similaire à Alloy. Ceci est réalisé par le biais d'une traduction du modèle et des spécifications en une représentation intermédiaire FIR.

Avant d'aborder l'analyse de code Java modélisant notre système contre ses spécifications, nous envisageons une approche particulière d'abstraction du modèle dans l'étape de modélisation : l'analyse des intervalles.

Analyse des intervalles : Abstraction du modèle

Le langage Java représente un langage de modélisation optimal dans notre étude de cas vu l'étendue de sa gamme de variables et d'opérations les manipulant. Cependant, modéliser notre système au moyen de variables concrètes en respectant les intervalles

appropriés des valeurs permises engendre une réduction dans l'espace d'états d'une multitude de cas possibles en un cas particulier. On passe alors du problème de model-checking au problème de simulation non désirable dans le contexte de notre travail.

De plus, l'utilisation de la class "Random" de Java qui permet de générer des variables aléatoires pour chaque exécution, ne résoud pas ce problème vu que ça restreint toujours l'ensemble des cas possibles en un cas bien déterminé quoique aléatoire. La couverture de la totalité du domaine des états n'est pas garantie.

Ici se manifeste le rôle primordial des abstractions dans le modèle. Au lieu de modéliser concrètement notre système, nous procédons à une manipulation d'intervalles suivant l'*arithmétique des intervalles*.

Par définition, l'*arithmétique* ou l'*analyse d'intervalles* est une arithmétique définie sur des ensembles d'intervalles au lieu d'ensembles de nombres réels. Plus spécifiquement, l'*arithmétique des intervalles* est une méthode de calcul permettant de garantir la relation d'*égalité*.

Dû à l'impossibilité de représenter concrètement un nombre réel, même après l'introduction du standard "IEEE 754" pour la représentation des nombres à virgule flottante en binaire au moyen des nombres à simple précision (32 bit) ou à double précision (64 bit), l'*analyse des intervalles* prend de l'ampleur en remédiant à ce problème.

D'ailleurs, cette méthode permet de réaliser des opérations dont le résultat est toujours certain. Ainsi, par exemple, écrire $\sqrt{2} = 1.414$ est faux puisque $(1.414)^2$ vaut exactement 1.999396 et non 2. Cette imprécision résulte des arrondissements ou approximations des nombres réels pour leur représentation dans la machine en nombres flottants. Dans le standard "IEEE 754" mentionné ci-haut, les quatres modes d'arrondissement sont :

- Vers $-\infty$
- Vers $+\infty$
- Vers zéro
- Au plus proche

Par contre, au moyen de l'arithmétique des intervalles, on peut écrire : $2 \in [2,2]$

puis calculer la formule exacte $\sqrt{2} \in [1.414, 1.415]$.

L'intérêt majeur octroyé à l'*analyse des intervalles* réside dans sa puissance de limiter les bornes des fonctions et des erreurs d'arrondissement attribuant un critère de fiabilité plus élevé en termes de résultats de calcul. En effet, ce critère joue un rôle primordial au sein de systèmes non linéaires comme dans notre simulateur de vol.

Dans cette arithmétique, un nombre réel **x** est représenté par une paire de nombres flottants [lo(x),hi(x)] où lo(x) et hi(x) sont respectivement les bornes inférieure et supérieure de **x**.

Cette représentation est équivalente à dire que $\mathbf{x} \in$ [lo(x),hi(x)].

Les assertions lo(x) $\leq \mathbf{x}$ et $\mathbf{x} \leq$ hi(x) sont de ce fait vraies. Par conséquent, la notion d'*égalité* d'un nombre et de sa représentation disparaît, sauf si lo(x)=hi(x).

Dans le cas où on a deux intervalles **x**=[lo(x),hi(x)] et **y**= [lo(y),hi(y)], les quatres opérations élémentaires pour une arithmétique idéale d'intervalles satisfait le suivant :

$$\mathbf{x}\, op\, \mathbf{y} = \{x\; op\; y |\; x \in \mathbf{x}\; et\; y \in \mathbf{y}\} \tag{3.3}$$

pour op $\in \{+, -, *, /\}$

Quelques définitions opérationnelles découlent de la formule 3.3 :

$$\mathbf{x} + \mathbf{y} = [lo(x) + lo(y), hi(x) + hi(y)] \tag{3.4}$$

$$\mathbf{x} - \mathbf{y} = [lo(x) - hi(y), hi(x) - lo(y)] \tag{3.5}$$

Implémentation des intervalles arithmétiques dans le code Java

Les notions préliminaires de l'arithmétique des intervalles étant définies, nous discutons à ce niveau l'implémentation de cette arithmétique en Java.

Il s'agit de quatre classes préalablement existantes en Java et importées dans notre code généré. Plus spécifiquement, nous ajoutons à notre code les classes *IAException*, *IAMath*, *RealInterval* et *RMath*.

IAException représente une classe qui implémente l'environnement d'exécution dans le cadre d'intervalles arithmétiques. Ces exceptions sont déclenchées en cas d'erreurs d'exécution relatives aux méthodes d'intervalles arithmétiques. L'erreur la plus fréquente découle principalement des intervalles vides "Empty Interval".

IAMath est une classe qui contient la majorité des méthodes permettant l'exécution d'opérations arithmétiques de base sur les intervalles comme l'intersection entre deux intervalles, leur union, leur multiplication, division et ce respectivement à l'aide des méthodes "intersect", "union", "mult" et "div". Par exemple, la figure 3.19 illustre l'implémentation des formules 3.4 et 3.5 par les méthodes respectives "add" et "sub" de cette classe. *IAMath* contient aussi des méthodes de transformations trigonométriques "sin", "cos" et "atan". Toutes ces méthodes de calcul retournent le résultat sous forme d'intervalles.

RealInterval* ou *Rint est la classe visée dans notre simulateur de vol pour la représentation de toutes les variables réelles par la construction d'intervalles. *Rint* représente une implémentation des intervalles fermés de nombres réels ayant chacun une borne inférieure de type double "lo" et une borne supérieure du même type "hi". Ses méthodes permettent entre autres de vérifier les relations entre deux intervalles comme l'égalité grâce à la méthode "equals". De plus, la méthode "toString" sert à afficher les intervalles suivant un format défini.

RMath La classe *RoundedMath* contient des méthodes et des constantes pour contrôler l'arrondissement des opérations arithmétiques de base sur les nombres à virgule flottante. Ces arrondissements efficaces sont indispensables lors de l'appel de toute méthode de la classe *IAMath*.

Vérification des propriétés et limitations

Une fois les classes relatives à l'arithmétique d'intervalles intégrées dans le code à vérifier, nous remplaçons toutes les variables réelles dans le code Java par des intervalles. Les autres variables de types différents restent inchangeables notamment les variables booléennes et l'énumération.

Le code est entamé par l'appel de la classe mère "DecisionTable" avec ses arguments qui sont les variables d'entrée de notre système. Tout d'abord, une vérification de la validité des variables d'entrée passées est réalisée avant toute poursuite d'exécution. Ensuite, la déclaration des variables de sortie est réalisée suivie par des méthodes

```java
public static RealInterval add(RealInterval x, RealInterval y) {
  RealInterval z = new RealInterval();
  z.lo = RMath.add_lo(x.lo,y.lo);
  z.hi = RMath.add_hi(x.hi,y.hi);
  return(z);
}

public static RealInterval sub(RealInterval x, RealInterval y) {
  RealInterval z = new RealInterval();
  z.lo = RMath.sub_lo(x.lo,y.hi);
  z.hi = RMath.sub_hi(x.hi,y.lo);
  return(z);
}
```

FIGURE 3.19 Implémentation de l'arithmétique d'intervalles en Java

de calcul respectives à toutes les variables intermédiaires. A la fin de chaque méthode, nous vérifions si la variable calculée appartient à son intervalle permis.

A ce niveau, le calcul des tables de décision suit à travers des méthodes spécifiques en vérifiant justement la validité des variables de sortie quant à leurs intervalles de valeurs.

```java
 @SpecField("realintervals: set Rint ") | this.realintervals - null = true")
public class DecisionTable {

   public enum LadderMode {DEFAULT,PITCH_LADDER_TAKEOFF, CLIMB_DIVE_LADDER};
   static LadderMode lm = LadderMode.DEFAULT;
   boolean weightOnWheels;
   Rint pitch;
   Rint roll;
   Rint trueHeading;
   Rint downVelocity;
   Rint eastVelocity ;
   Rint northVelocity;
```

FIGURE 3.20 Entête du code Java

Vérification des propriétés La modélisation du système étant achevée, nous procédons maintenant à l'implémentation des propriétés à analyser. Ceci est réalisé par une insertion simple des spécifications dans le code sous forme d'annotations dans le langage JFSL. La figure 3.20 montre la spécification générale d'une table de décision

non nulle insérée dans l'entête du code Java généré. De plus, la figure 3.21 illustre une annotation JFSL vérifiant que la variable de sortie FPM_az n'est jamais égale à une valeur inconnue. Ceci est réalisé en définissant une valeur inconnue "unknown" comme étant un intervalle ayant les mêmes bornes supérieure et inférieure égales à la valeur réelle 1000 ("static Rint unknown = new Rint(1000.0) ;").

```
/* Second-Order Table to compute*/
/* FPM_az : Flight Path Marker azimuth in Celsius degrees*/
@Ensures("no x in this.realintervals | x.lo = x.hi = 1000.0")
public Rint calcFPM_az() throws IAException{
 switch(lm){
    case DEFAULT: //This case is feasible whenever calcVelocityBodyX is false
    System.out.println("This case is feasible whenever calcVelocityBodyX() is false");
        break;
    case PITCH_LADDER_TAKEOFF:
        FPM_az=IAMath.mul(pitch, IAMath.sin(roll)); //FPM_az=Pitch * sin(Roll);
        FPM_az=IAMath.mul(FPM_az,RadiansToCelsius);
        FPM_az=IntervalDoubleToFloat(FPM_az);
        //Convert the negative angles by adding 2 PI (2PI pres)
        if (FPM_az.lo <0.0)
            FPM_az.lo=360+FPM_az.lo;
        if (FPM_az.hi<0.0)
            FPM_az.hi=360+FPM_az.hi;
        if (FPM_az.lo >360.0)
            FPM_az.lo=FPM_az.lo-360.0;
        if (FPM_az.hi >360.0)
            FPM_az.hi=FPM_az.hi-360.0;
        expectedFPM_az.intersect(FPM_az);
        if (expectedFPM_az.equals(FPM_az))
            System.out.println("Success in FPM_AZ ENTRE BORNES");
        else {
            System.out.println("Failure in FPM_AZ ENTRE BORNES");
            FPM_az = outofbound;
        }
        break;
```

FIGURE 3.21 Annotation JFSL pour la vérification du code Java

De ce fait, afin de vérifier un système traduit en Java, il suffit de sélectionner la méthode dont on désire vérifier les propriétés et de lancer la commande "Check and Simulate".

Limitation de JForge Bien que cette méthodologie semble parfaitement adéquate dans notre cas, l'outil considéré ne permettait pas d'ajouter des librairies externes telles que les classes implémentant les intervalles arithmétiques en raison de quelques limitations dans la configuration de JForge, cet outil étant toujours en développement.

Par conséquent, et après une revue de littérature, nous avons eu recours au model-checker logiciel de programmes Java, Java PathFinder, qui présente une maturité assez élevée dans le monde de vérification et de test. Comme nous allons voir un peu plus loin dans ce manuscrit, cet outil d'analyse statique et de vérification de code Java, que nous allons noter JPF dans ce qui suit, permet pratiquement la vérification de tout système numérique en raison des démarches suivies pour atténuer le plus efficacement possible le fameux problème d'explosion d'états.

3.4.4 Expérimentations avec JPF et son extension Symbolic JPF

Notre motivation primordiale à recourir au model-checker JPF est justifiée par les avantages de la modélisation en langage Java quant à la qualité numérique complexe de notre système.

Bien que son application était principalement privilégiée dans la détection d'erreurs de programmation au moyen de rapports avec les lignes de code source critiques, le model-checker JPF a été étendu dans plusieurs projets suivant différentes pistes de travail, notamment l'exécution symbolique de code Java : Symbolic JPF.

Cette extension de JPF combine l'exécution symbolique avec le model-checking et la résolution de contraintes pour la génération de cas de test en entrée.

Cette démarche particulière étend même notre objectif principal de sorte qu'elle nous permet de vérifier les propriétés d'accessibilité et de vivacité de notre système mentionnées précédemment tout en générant automatiquement des cas de test en entrée plutôt que des traces d'exécution. Ces cas de test, assurant une grande couverture de chemins d'exécution, peuvent être personnalisés en termes de variables et de chemins d'exécution les plus pertinents comme nous allons voir plus loin dans cette

section.

Davantage, l'exécution symbolique est d'autant plus cruciale dans notre étude de cas dans la mesure où le programme s'exécute symboliquement dans SYMBOLIC JPF : Des variables symboliques sont passées en entrée pour couvrir la totalité des variables d'entrée concrètes possibles. Les valeurs de ces variables sont ainsi représentées par des expressions numériques et des contraintes générées par l'analyse de la structure du code. Ces contraintes sont par la suite résolues afin de générer des cas de test en entrée garantis d'accéder à la partie pertinente du code analysé.

De ce fait, à ce niveau, l'analyse des intervalles auparavant indispensable dans le code Java de JFORGE, n'est plus nécessaire avec SYMBOLIC JPF. Par ailleurs, l'exécution symbolique résoud le problème persistant dans l'analyse d'intervalles et découlant de l'interdépendance entre les variables. Par exemple, les trois angles calculés CXX, CXY et CXZ ne peuvent jamais avoir une valeur commune '-1' même si cette valeur satisfait les trois intervalles permis [-1,1] des trois variables . Le système trigonométrique formé de ces trois variables a comme solution possible CXZ = -1 lorsque la variable d'entrée $pitch = \frac{\Pi}{2}$ résultant en une valeur nulle des deux angles CXX et CXY.

Vu l'impossibilité de gérer même des intervalles pour représenter l'ensemble de tous les états possibles de notre système, l'exécution symbolique semble être une raison supplémentaire nous incitant à son intégration dans le model-checking.

Avant de discuter l'approche adoptée pour l'application de l'exécution symbolique dans notre code généré à l'aide de SYMBOLIC JPF, nous développons les modifications apportées à la grammaire depuis sa conception dans l'exemple illustratif. De plus, l'ajout des visiteurs pour la génération de code approprié est également discuté dans la partie suivante.

Modifications de grammaires et ajout de visiteurs

Dans notre étude de cas, les phases d'analyse syntaxique et lexicale sont abordées à partir de trois fichiers de base : L'un d'entre eux est issu des tables de décision alors que les deux autres le sont de l'ensemble variables-opérations des figures respectives

3.17 et 3.18.

Plus spécifiquement, la manipulation de ces trois fichiers est catégorisée dans trois paragraphes consécutifs délimitant de ce fait les grammaires, les visiteurs et les codes générés respectifs.

Cependant, il reste à noter qu’une étape commune, relative à la modification de la grammaire dans les trois fichiers, est discutée à ce niveau. En effet, afin d’étendre la gamme d’opérateurs et d’opérations arithmétiques indispensables dans notre simulateur de vol, une importation de la grammaire "*Expression*", préalablement conçue, s’est avérée nécessaire dans JavaCC.

Toutefois, des modifications majeures ont été apportées à cette grammaire pour répondre à nos besoins spécifiques dans la collecte des variables pertinentes pour la génération de code.

Tout d’abord, nous avons ajouté trois jetons $< GT >$, $< LP >$ et $< RP >$ avec leurs règles de production et leurs nœuds définis afin de rendre possible le stockage respectif de la relation ’>’ et des parenthèses gauche ’(’ et droite ’)’.

Ensuite, nous avons défini 14 nœuds dans la grammaire correspondant aux 14 jetons existants. De plus, nous avons ajusté leur règle de production respective pour leur extraction ultérieure. Ces nœuds sont catégorisés comme suit :

Opérationnels Ils sont au nombre de huit : Addition ’Add’, Soustraction ’Sub’, Multiplication ’Mult’, Division ’Div’, Modulo ’Rem’, Incrémentation ’Incr’, Décrémentation ’Decr’ et opérateur d’assignation ’AssignmentOp’

Relationnels Ils sont au nombre de quatre : $<$ ’LT’, $\leq$ ’LE’, $\geq$ ’GE’ et $! =$ ’NE’

Logique Un seul nœud ’And’ et sa règle de production respective "ConditionalAndExpression" ont été ajoutés à la grammaire afin de récupérer les conjonctions ’&&’ dans les préconditions des variables d’entrée comme dans la figure 3.22 ainsi que dans les conditions des variables intermédiaires et des variables de sortie

Constante Le nœud ’Literal’ permet de récupérer toutes les valeurs numériques

entières et réelles ainsi que les identificateurs incluant les transformations trigonométriques comme cos, sin et les noms des variables. Également, ce nœud est important puisqu'il sert à indiquer la nécessité de convertir les angles dans les variables d'entrée des degrés Celsius aux degrés Radians nécessaires à tout calcul avec la classe "Math" de Java

Finalement, il faut noter qu'une règle de production spéciale "RandValue" a été ajoutée afin de distinguer, suivant le contexte, le même jeton '-' utilisé à la fois dans les soustractions et la représentation des valeurs aléatoires au sein des tables de décision.

En somme, nous aurons 17 nœuds définis en total—les *nœuds d'expression*—avec leurs règles de production. Les *nœuds d'expression* ainsi obtenus sont intégrés dans les trois grammaires correspondant aux trois fichiers d'entrée. Cette étape est cruciale dans la mesure où le visiteur que nous concevrons par la suite, serait apte à une collecte "personnalisée" des variables depuis ces fichiers.

Génération de préconditions Le premier fichier "DTInputFile", de la figure 3.22, est généré à partir des variables d'entrée et de leurs préconditions telles qu'affichées sous l'entête "*(Input Variables)*" de la figure 3.18.

Après les balises incluant le mot-clé "IVARIABLE", nous définissons une variable d'entrée de notre système suivant le format : [type ; nom de la variable]. Ensuite, sa précondition particulière, si elle existe, est dénotée à travers des expressions bornées par les balises contenant le mot-clé "PREC". Cette représentation est adoptée pour l'ensemble de toutes les variables d'entrée et de leurs préconditions respectives.

Pour la variable d'entrée booléenne *WeightOnWheels*, sa précondition triviale est omise en raison de ses deux valeurs possibles connues Vrai ou Faux.

Une fois le fichier d'entrée défini, nous procédons à la conception de la grammaire pour les étapes d'analyse syntaxique et lexicale. En plus des *nœuds d'expression* mentionnés ci-haut, cette grammaire comprend deux nœuds supplémentaires : le nœud racine "Start" suivi de son enfant "IVar". Ainsi, ce dernier nœud représente, dans

```
<IPREC>
    <IVARIABLE>boolean weightOnWheels
    </IVARIABLE>

    <IVARIABLE>double pitch
        <PREC> >= 0.0 && <=360.0
        </PREC>
    </IVARIABLE>
    (...)
    <IVARIABLE>double downVelocity
        <PREC> >= -270.0 && <=270.0
        </PREC>
    </IVARIABLE>
    (...)
</IPREC>
```

FIGURE 3.22 Fichier d'entrée "DTInputFile"

cette hiérarchie, le parent de tous les *nœuds d'expression*.

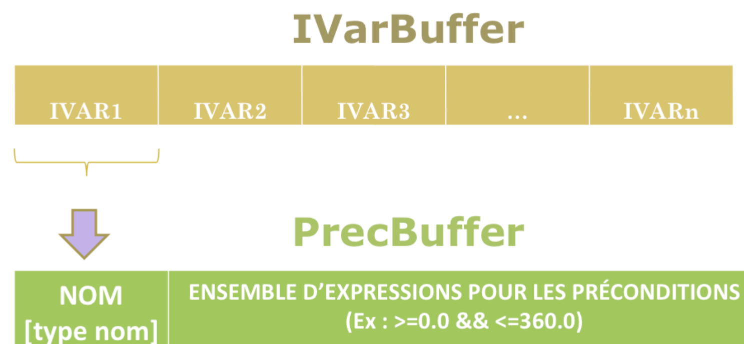

FIGURE 3.23 Extraction de l'information pour la génération des préconditions

Dans le parcours ultérieur de l'AST ainsi généré, notre visiteur crée un vecteur général appelé "IVarBuffer" tel qu'illustré dans la figure 3.23. Chaque indice dans ce vecteur consiste en un autre vecteur en soi. Ce dernier collecte, dans son premier indice, les noms de chaque variable d'entrée sous le format [type ; nom de la variable]. L'indice suivant représente le vecteur "PrecBuffer" tel qu'illustré dans la figure 3.23 contenant l'ensemble des préconditions sous forme d'expressions.

Par conséquent, la génération de code peut ainsi suivre résultant en une classe Java

"MyDriverforDTsample_auto" illustrée dans la figure 3.24. Cette classe représente en soi la classe mère déclenchant l'exécution symbolique au sein de l'outil SYMBOLIC JPF.

Cette classe mère ne permet l'exécution que dans le cas où la totalité des variables d'entrée satisfait les préconditions définies sans aucune exception. Ceci est implémenté grâce aux conjonctions de "if" dans la méthode "imposePreconditions".

```
package coverage;
public class MyDriverforDTsample_auto {
    private static void imposePreconditions(boolean weightOnWheels, double pitch, double roll,
    double trueHeading, double downVelocity, double eastVelocity, double northVelocity)  {
        DTsampleannotated_auto DTannotated = new DTsampleannotated_auto();

        if ( ( pitch >=  0.0  && pitch  <=  360.0  ) && ( roll >=  0.0  && roll  <=  360.0  ) &&
        ( trueHeading >=  0.0  && trueHeading  <=  360.0  ) &&
        ( downVelocity >=  -  270.0  && downVelocity  <=  270.0  ) &&
        ( eastVelocity >=  -  270.0  && eastVelocity  <=  270.0  ) &&
        ( northVelocity >=  -  270.0  && northVelocity  <=  270.0  ) ) {
            DTannotated.calcFPM(weightOnWheels, pitch, roll, trueHeading,
            downVelocity, eastVelocity, northVelocity);
        }
    }
    public static void main(String[] args) {
        imposePreconditions(false, -78.65900123197183, 136.62316026742434,
        4.375005178460285, -51.894691312850746, 74.60071648879666, 169.1746403241376);
    }
}
```

FIGURE 3.24 Génération de la classe mère dans l'exécution symbolique

Génération de l'entête de code Le second fichier "ConditionInputFile" de la figure 3.25 est généré à partir des variables et de leurs opérations affichées dans la figure 3.18.

Dans ce fichier, les variables d'entrée sont bornées par des balises incluant le mot-clé "IVAR" dans le format [type; nom de la variable].

Les variables de sortie suivent après les balises "OVAR" sous le même format que précédemment. Toutefois, ces variables de sortie sont assignées à des valeurs initiales par défaut après les balises "DEFAULT" : Les variables réelles sont initialisées à la valeur réelle définie "_unknown_ ", l'énumération à la valeur constante défi-

```
<VAR>
    <IVAR> boolean weightOnWheels
    </IVAR>
    <IVAR> double pitch_ANGLE
    </IVAR>
    (...)
    <OVAR> double climb_dive_angle
        <DEFAULT> _unknown_
        </DEFAULT>
        <COND> >=0 && <=360
        </COND>
    </OVAR>
    <OVAR> enum LadderMode {DEFAULT,PITCH_LADDER_TAKEOFF,CLIMB_DIVE_LADDER}
        <DEFAULT> DEFAULT
        </DEFAULT>
    </OVAR>
    (...)
    <CVAR> double CXX
        <EXPR> cos(trueHeading) * cos(pitch)
        </EXPR>
        <COND> >=-1 && <=1
        </COND>
    </CVAR>
    (...)
</VAR>
```

FIGURE 3.25 Fichier d'entrée "ConditionInputFile"

nie "DEFAULT" et la variable booléenne à "Faux". En plus, les variables de sortie réelles sont conditionnées par des expressions bornées par des balises avec le mot clé "COND".

Finalement, les variables intermédiaires sont représentées après les balises "CVAR" sous le même format. Ici aussi, la définition de calcul de ces variables ainsi que leurs intervalles permis sont représentés respectivement après les balises "EXPR" et "COND".

Après l'élaboration de ce fichier d'entrée, la grammaire conçue rajoute sept nœuds de base aux *nœuds d'expression* préalablement établis : le nœud racine "Start" suivi des enfants "IVar", "OVar" et "CVar". Les deux derniers ont respectivement les nœuds-fils "DValue" d'une part, et "Value" et "Cond" d'autre part.

L'extraction de l'information suivant l'analyse syntaxique et lexicale aboutit aux vecteurs illustrés dans la figure 3.26. Le vecteur "IVarBuffer" collecte le nom et type des variables d'entrée alors que les vecteurs "OVarBuffer" et "CVarBuffer", non illus-

trés dans la figure pour des raisons de brièveté, collectent les expressions relatives aux variables de sortie et aux variables intermédiaires respectivement. Les vecteurs "OVar" et "CVar" représentent un vecteur parmi d'autres dans chacun de ces vecteurs.

Il est important de noter que la hiérarchie entre les nœuds est assurée par les deux méthodes intégrées dans JavaCC : "jjtGetParent" et "jjtGetNumChildren".

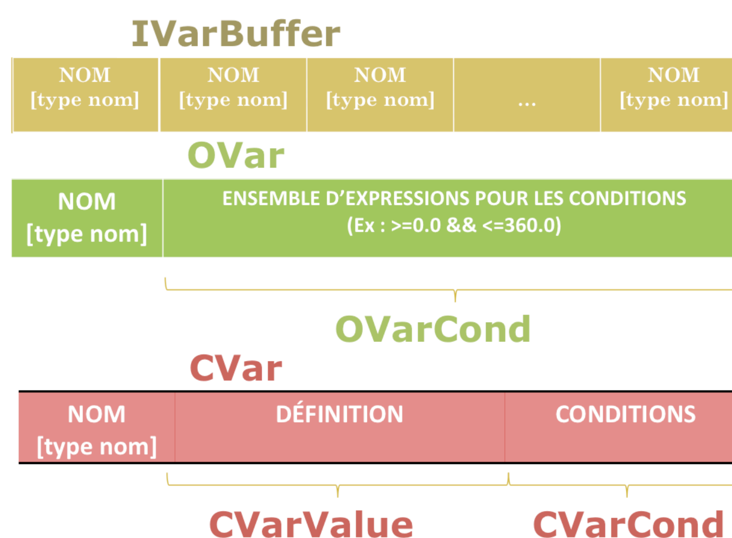

FIGURE 3.26 Extraction de l'information pour la génération de l'entête du code

Ces vecteurs seront nécessaires à la génération de l'entête du code qui, lorsqu'éventuellement complété, servira à la modélisation de notre système. Un extrait de cette entête est illustré dans la figure 3.27. Le code ainsi généré comprend :

- La déclaration des constantes globales du système ainsi que des variables indispensables pour l'exécution symbolique : l'entier "jpfIfCounter" et la chaîne de caractères "path". Ces deux variables seront interprétées plus loin.
- La déclaration des variables de sortie avec leurs valeurs initiales.
- Les conversions d'angles nécessaires pour les variables d'entrée.
- L'appel de la méthode qui assume le calcul des tables de décision. Cette méthode prend comme argument les variables d'entrée ainsi que leur type.

– La déclaration des variables intermédiaires suivies par leurs définitions et leurs conditions d'évaluation. Ces conditions sont implémentées par des "if" dans le code suivis des "path.append" pour leur intégration dans le chemin d'exécution final des cas de test en entrée.

```
public class DTsampleannotated_auto {
/* * Output variables */
    final double _unknown_ = 500.0;
    double climb_dive_angle = _unknown_;

    enum LadderMode {
        DEFAULT, PITCH_LADDER_TAKEOFF, CLIMB_DIVE_LADDER
    };

    static LadderMode enumVar1 = LadderMode.DEFAULT;
    double FPM_az = _unknown_;
    double FPM_el = _unknown_;
    boolean Sig_Al = false;

    public double calcFPM(boolean weightOnWheels, double pitch, double roll,
            double trueHeading, double downVelocity, double eastVelocity,
            double northVelocity) { /* * Constants */
        final double ConvertCelstoRad = (2 * Math.PI) / (360.0);
        final double ConvertRadtoCels = (360.0) / (2 * Math.PI);
        int jpfIfCounter = 0;
        StringBuilder path = new StringBuilder();
        pitch = pitch * ConvertCelstoRad;
        roll = roll * ConvertCelstoRad;
        trueHeading = trueHeading * ConvertCelstoRad; /* * Derived Variables */
        double CXX;
        double CXY;
        double CXZ;
        double CYX;
        double CYY;
        double CYZ;
        double CZX;
        double CZY;
        double CZZ;
        double VelocityBodyX;
        double VelocityBodyY;
        double VelocityBodyZ;

        CXX = Math.cos(trueHeading) * Math.cos(pitch);
        CXY = Math.sin(trueHeading) * Math.cos(pitch);
        CXZ = -Math.sin(pitch);
        (...)
        if (CXX >= -1 && CXX <= 1)
            path.append("<br />**Derived Variables conditions** [CXX >=  -  1  && CXX  <=  1 ]<br />");
        if (CXY >= -1 && CXY <= 1)
            path.append("<br />**Derived Variables conditions** [CXY >=  -  1  && CXY  <=  1 ]<br />");
        (...)
```

FIGURE 3.27 Entête du code modélisant notre système

Génération du code correspondant aux tables de décision Le troisième et dernier fichier "DTFile" dans la figure 3.28 est généré à partir des tables de décision de la figure 3.17.

```
<SPECS>
    <Table>
        <Name> LadderMode
            <InputVariablesHeader>
                <InputVariable> weightOnWheels
                </InputVariable>
                <InputVariable> VelocityBodyX
                </InputVariable>
            </InputVariablesHeader>
            <OutputVariableHeader>
                <OutputVariable> LadderMode
                </OutputVariable>
            </OutputVariableHeader>
            <Rows>
                <SingleRow>
                    <InputValue> true
                    </InputValue>
                    <InputValue>
                        <EXPR> <60
                        </EXPR>
                    </InputValue>
                    <OutputValue> PITCH_LADDER_TAKEOFF
                    </OutputValue>
                </SingleRow>
                <SingleRow>
                    <InputValue> true
                    </InputValue>
                    <InputValue>
                        <EXPR> >=60
                        </EXPR>
                    </InputValue>
                    <OutputValue> CLIMB_DIVE_LADDER
                    </OutputValue>
                </SingleRow>
                <SingleRow>
                    <InputValue> false
                    </InputValue>
                    <InputValue> -
                    </InputValue>
                    <OutputValue> CLIMB_DIVE_LADDER
                    </OutputValue>
                </SingleRow>
            </Rows>
        </Name>
    </Table>
    <Table>
    (...)
</SPECS>
```

FIGURE 3.28 Fichier d'entrée "DTFile" correspondant au calcul des tables de décision

Ce fichier encapsule toutes les spécifications et les calculs des tables de décision après les balises incluant le mot-clé "SPECS". Dans la figure, une seule table de décision, *LadderMode*, traduite en format prédéfini est illustrée. Suivant ce format, les noms des variables d'entrée ainsi que de la variable de sortie de cette table sont représentés. Ensuite, chaque balise "SingleRow" contenant les valeurs correspondant à

ces variables dénote une ligne donnée de la table de décision, en d'autres termes un état particulier.

A ce niveau, la grammaire conçue à partir du fichier d'entrée "DTFile" comprend 10 nœuds de base en plus des *nœuds d'expression*. Comme avant, le nœud "Start" est la racine de l'AST généré suivi des nœuds-fils "Table" correspondant à chaque table rencontrée dans le fichier "DTFile". Chaque nœud "Table" aura comme nœuds-fils "ITable" pour les noms des variables d'entrée, "OTable" pour ceux des variables de sortie et "SingleRow" contenant toutes les lignes constituant notre table. En particulier, le nœud "SingleRow" est le parent des nœuds-fils résultants "IValue", "IValueExpr", "OValue", "OValueExpr" et "RandValue" représentant toutes les valeurs possibles des variables d'entrée et de sortie.

En raison des similarités avec les étapes précédentes, l'extraction de l'information sous forme de vecteurs ne sera pas détaillée dans cette partie.

Comme résultat de visite de cet AST, nous générons le code modélisant les tables de décision tel qu'illustré dans la figure 3.29 pour la table de décision considérée *LadderMode*.

L'entête du code ainsi que ce dernier seront annexés à l'aide d'un simple programme Java résultant ainsi en une classe "DTsampleannotated_auto" appelée dans la classe mère "MyDriverforDTsample_auto" de la figure 3.24.

```
if (weightOnWheels == true && VelocityBodyX < 60) {
    path.append("<br />**Table order 1: LadderMode** [ weightOnWheels ==  true && VelocityBodyX <  60 ]<br />");
    enumVar1 = LadderMode.PITCH_LADDER_TAKEOFF;
}
if (weightOnWheels == true && VelocityBodyX >= 60) {
    path.append("<br />**Table order 1: LadderMode** [ weightOnWheels ==  true && VelocityBodyX >=  60 ]<br />");
    enumVar1 = LadderMode.CLIMB_DIVE_LADDER;
}
if (weightOnWheels == false) {
    path.append("<br />**Table order 1: LadderMode** [ weightOnWheels ==  false ]<br />");
    enumVar1 = LadderMode.CLIMB_DIVE_LADDER;
}
```

FIGURE 3.29 Code modélisant nos tables de décision

Le code Java étant disponible, il est possible à ce niveau d'entamer l'exécution

symbolique à l'aide de Symbolic JPF comme discutée dans la partie suivante.

Exécution symbolique avec Symbolic JPF

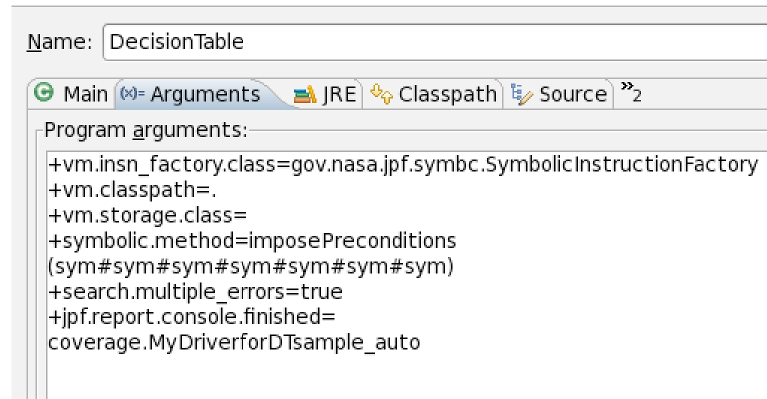

Figure 3.30 Configuration dans Symbolic JPF

La figure 3.30 illustre la configuration d'exécution adoptée pour l'exécution symbolique avec Symbolic JPF.

Afin d'exécuter "symboliquement" notre système, nous précisons à JPF l'utilisation d'un interpréteur non standard de codes objets. Cet interpréteur n'est autre que *SymbolicInstructionFactory* précisé dans la première ligne de cette configuration.

Ensuite, nous lui indiquons le nom de la méthode à exécuter symboliquement, ici "imposePreconditions", avec ses arguments qui sont les sept variables d'entrée de notre système. Cependant, ces arguments ne sont pas passés à travers leur nom comme dans une exécution Java ordinaire. Par contre, avec Symbolic JPF, il suffit de passer le nombre exact d'arguments avec le mot-clé "#sym" ou "#con" pour indiquer respectivement l'aspect symbolique ou concret de ces arguments. Dans notre simulateur de vol, toutes les variables d'entrée sont symboliques comme nous pouvons constater dans la cinquième ligne de la figure 3.30. Dans la figure 3.24, nous précisons que les arguments concrets passés à la méthode exécutée symboliquement "imposePreconditions" ont été générés aléatoirement par la classe "Random" de Java vu l'absence d'impact de ces valeurs sur l'exécution symbolique tant que ces argu-

ments sont déclarés "#sym" lors de la configuration.

La dernière ligne de la configuration comprend le nom de l'application "main" de Java—MyDriverforDTsample_auto—qui appelle la méthode "imposePreconditions" pour son exécution symbolique.

Caractéristiques de base de Symbolic JPF

Générateur de choix Par défaut et en l'absence de précision inverse, l'interface générique de procédures de décision/solutionneurs de contraintes **Choco** est utilisée comme un générateur de choix. Cette interface est compatible avec les contraintes linéaires et non linéaires, entières et réelles. Son rôle est de vérifier la validité des chemins d'exécution rencontrés tout au long de l'exécution symbolique.

Ainsi, pour chaque chemin d'exécution "faisable", Symbolic JPF construit une condition de chemin ou "*Path Condition*". De ce fait, une séquence de chemins d'exécution parcourus résulte en une conjonction de ces conditions de chemin sur les variables d'entrée. Si une condition particulière d'un chemin d'exécution résolue par **Choco** n'est pas satisfiable, le model-checker JPF rebrousse chemin afin d'explorer uniquement les chemins satisfiables. C'est à ce niveau que l'exécution symbolique combinée aux deux points forts du model-checking d'une part et de la résolution de contraintes d'autre part prend de l'envergure dans notre étude de cas.

Les contraintes non linéaires et les opérations arithmétiques complexes, auparavant identifiées comme un obstacle redoutable dans le model-checking surtout du point de vue extensibilité, sont manipulées par **Choco** avec la même simplicité que les contraintes linéaires.

Observateur *Listener* Par défaut, l'observateur standard de JPF— "Symbolic Listener" — sert à imprimer les conditions de chemins **PCs** et les résumés des méthodes exécutées. Cependant, dans notre étude de cas, nous implémentons notre propre observateur "JPF_coverage_CheckCoverage" qui déclenche l'exécution symbolique depuis le premier appel de la méthode à exécuter "imposePreconditions".

Lorsque cette méthode retourne (ou une limite d'exécution fixée par l'utilisateur a été atteinte), le model-checker JPF imprime les cas de test caractérisant l'exécution symbolique. Cet affichage a été personnalisé en utilisant des mots-clés **HTML** pour construire des tableaux, ayant dans chaque ligne : Une colonne représentant le numéro du cas de test, suivie par sept colonnes correspondant aux valeurs concrètes des variables d'entrée soulignant des situations particulières dans les chemins d'exécution visités. La dernière colonne affiche la conjonction résultante des conditions de chemin **PCs**. Ce dernier affichage des **PCs** est rendu possible au moyen des appels "path.append" utilisés au fur et à mesure de l'exécution du code généré, plus spécifiquement après la satisfiabilité d'une condition particulière, comme après la condition "*if (weightOnWheels == true && VelocityBodyX < 60)*" dans la figure 3.29. Conséquemment, la condition précédente sera annexée à la condition de chemin actuelle au moyen de "path.append".

```
if (FPM_az == _unknown_) {
    jpfIfCounter++;
    path.append("<br />**Table order 5: Sig_Al** [ FPM_az ==  _unknown_ ]<br />");
    Sig_Al = true;
}
if (FPM_el == _unknown_) {
    path.append("<br />**Table order 5: Sig_Al** [ FPM_el ==  _unknown_ ]<br />");
    Sig_Al = true;
}
if (climb_dive_angle == _unknown_) {
    path.append("<br />**Table order 5: Sig_Al** [ climb_dive_angle ==  _unknown_ ]<br />");
    Sig_Al = true;
}



if (Sig_Al == false && FPM_az == (_unknown_)) {
    path.append("<br />**Table order 5: Sig_Al** [ Sig_Al == false && FPM_az == (_unknown_)]<br />");
    System.out.println("Impossible Situation");
}

Verify.ignoreIf(jpfIfCounter != 1);
```

FIGURE 3.31 Filtrage des cas de test pertinents

Instrumentation avec "Verify" Le model-checker JPF offre l'opportunité de filtrer les cas de test qui nous sont pertinents et ce, grâce à la classe "Verify" et plus précisément sa méthode "ignoreIf". Ainsi, dans la figure 3.31, nous filtrons les cas de test en entrée pour la vérification de la propriété d'accessibilité relative à FPM_az : [FPM_az==$_unknown_$]. La variable "jpfIfCounter"

initialisée à '0' dans chaque exécution de la méthode est incrémentée si et seulement si la condition/propriété précédente est satisfaite. L'appel de la méthode "*Verify.ignoreIf(jpfIfCounter != 1)*" à la fin du code généré permet ce filtrage. En effet, tant que la condition n'est pas satisfaite, la variable "jpfIfCounter" reste nulle et les cas de test aboutissant à cette condition sont ignorés.

Nous illustrons dans la figure 3.32 une ligne du tableau HTML représentant le premier cas de test en entrée lors de la vérification de la propriété : [*FPM_ az==_ unknown_*]. Nous notons que la dernière colonne "Path" de ce cas de test est représentée sur une nouvelle ligne afin de rendre visible le plus possible la figure.

Résultats expérimentaux

En instrumentant le code pour les différentes propriétés à vérifier, comme dans la partie précédente, nous obtenons les résultas affichés dans le tableau 3.2.

Ces résultats de l'exécution symbolique sont obtenus sur une machine Intel Pentium 4 de fréquence 1.8 GHz. La première colonne dénote les différentes propriétés vérifiées : Les trois premières représentent des propriétés d'accessibilité alors que la dernière démontre la vivacité dans notre système. La seconde colonne représente le nombre de cas de test obtenus en entrée. La durée d'exécution exprimée en secondes est indiquée dans la dernière colonne.

TABLEAU 3.2 Évaluation des cas de test obtenus

Condition du chemin d'exécution	Nombre de cas de test	Durée d'exécution
FPM_az==unknown	729	115
FPM_el==unknown	729	115
Climb_Dive_Angle==unknown	1458	176
Sig_Al==false et FPM_az==unknown	0	43

Nous remarquons que le nombre de cas de test obtenus lors de la vérification des deux propriétés [*FPM_ az==_ unknown_*] et [*FPM_ el==_ unknown_*] est iden-

Testcase #	weight_on_wheels	pitch	roll	trueHeading	downVelocity	eastVelocity	northVelocity
1	false	0.0	0.0	0.0	-270.0	-270.0	-270.0

Path

Derived Variables conditions [CXX >= - 1 && CXX <= 1]

Derived Variables conditions [CXY >= - 1 && CXY <= 1]

Derived Variables conditions [CXZ >= - 1 && CXZ <= 1]

Derived Variables conditions [CYX >= - 1 && CYX <= 1]

Derived Variables conditions [CYY >= - 1 && CYY <= 1]

Derived Variables conditions [CYZ >= - 1 && CYZ <= 1]

Derived Variables conditions [CZX >= - 1 && CZX <= 1]

Derived Variables conditions [CZY >= - 1 && CZY <= 1]

Derived Variables conditions [CZZ >= - 1 && CZZ <= 1]

Derived Variables conditions [VelocityBodyX >= - 270 && VelocityBodyX <= 270]

Derived Variables conditions [VelocityBodyY >= - 270 && VelocityBodyY <= 270]

Derived Variables conditions [VelocityBodyZ >= - 270 && VelocityBodyZ <= 270]

Table order 1: LadderMode [weightOnWheels == false]

Table order 2: FPM_az [enumVar1==LadderMode. CLIMB_DIVE_LADDER && VelocityBodyX <= 0]

Table order 3: FPM_el [enumVar1==LadderMode. CLIMB_DIVE_LADDER && VelocityBodyX <= 0]

Table order 4: climb_dive_angle [VelocityBodyX <= 0]

Table order 5: Sig_Al [FPM_az == _unknown_]

Table order 5: Sig_Al [FPM_el == _unknown_]

Figure 3.32 Un cas de test de la propriété [$FPM_az{=}{=}_unknown_$]

tique. Ceci est logique dû à la structure commune des conditions d'entrée des deux tables de décision correspondantes. Sous le même principe de structure de tables de décision, nous notons que si l'une de ces deux variables de sortie est inconnue, ceci implique que la variable de sortie $Climb_Dive_Angle{=}{=}_unknown_$. Ce cas particulier est illustré dans la figure 3.32. Toutefois, l'inverse n'est pas nécessairement vrai.

De plus, la dernière ligne de la table 3.2 correspond au chemin d'exécution qui suppose que le signal d'alarme n'est pas déclenché *Sig_ Al==false* ; pourtant, une des variables de sortie est égale à la valeur inconnue suscitant l'état d'erreur critique. Ceci est exprimé par la conjonction *(Sig_ Al==false) && (FPM_ az==unknown).*

Dans ce cas, il n'existe aucun cas de test en entrée généré affirmant la bonne fonctionnalité du signal d'alarme vis-à-vis des états d'erreur. Il reste à noter que les mêmes résultats sont obtenus, dans ce cas, si l'on utilise les autres variables de sortie au lieu de *FPM_ az* pour les états critiques.

Ces cas de test en entrée, générés automatiquement dans moins de 2 minutes en moyenne, représenteront de la sorte des traces d'exécution illustrant les scénarios aboutissant à la condition spécifique du chemin d'exécution.

Chapitre 4

SECONDE ÉTUDE DE CAS : SYSTÈME D'ÉDITIONS COLLABORATIVES DISTRIBUÉES

Le modèle **pair-à-pair**, "P2P" (Peer-to-Peer), est un modèle de réseau informatique qui s'oppose strictement au modèle client-serveur. Les systèmes pair-à-pair permettent à plusieurs ordinateurs—nœuds dans le réseau— de communiquer via un réseau en partageant simplement des objets comme des fichiers le plus souvent, mais également des flux multimédias continus, le calcul réparti ou un service comme la téléphonie.

Cependant, le partage de fichiers en pair à pair demeure l'application la plus répandue dans ce type de modèle, surtout au sein des systèmes de fichiers répartis ou distribués.

Par définition, un **système de fichier distribué** ou système de fichier en réseau représente un système qui stocke des informations sur des machines distantes de manière à ce que, du point de vue de l'utilisateur, tout se passe comme si les données étaient stockées localement. Dans ces systèmes, chaque internaute est un pair du réseau et les ressources sont des fichiers.

Basés sur ce principe, les systèmes d'édition collaborative distribuée procèdent à une *réplication de données* où les mêmes données sont dupliquées sur plusieurs périphériques qu'on appellera **sites** par la suite.

La réplication est un processus de partage d'informations assurant à la fois la cohérence de données entre plusieurs sources de données redondantes et l'amélioration de la fiabilité. Au sein de ces systèmes, on s'adresse plus spécifiquement à la réplication *active*, où l'ensemble des calculs effectués par la source est répliqué sur les autres sites.

Notre étude de cas consiste ainsi en un *système d'éditions collaboratives distribuées* dans lequel un site quelconque peut traiter une requête ou une opération. Nous considérons ainsi un schéma de réplication multi-maître. Cette architecture pose des problèmes de contrôle de concurrence : plusieurs processus qui travailleraient de manière incontrôlée sur les mêmes données pourraient remettre en cause la cohérence globale du système.

Afin de résoudre ces problèmes et dans le but de supporter une édition collaborative efficace, des transformées d'opérations non locales— transformées opérationnelles — sont nécessaires dans chaque site.

De ce fait, une multitude d'algorithmes de réplication optimiste basés sur l'approche des transformées opérationnelles ont été proposés dans la littérature. Leur but principal consiste à garantir la propriété primordiale de consistance de ces systèmes : La convergence des copies au niveau de chaque site.

Toutefois, l'obstacle majeur envisagé dans ces algorithmes découle de l'infinité d'états compris dans ces systèmes. En effet, le caractère interactif imprévisible et inhérent à ces systèmes, combiné à la réplication de données, contribuent considérablement dans l'augmentation du nombre d'états.

Par conséquent, assurer la convergence au sein de tels systèmes à l'aide de ces algorithmes semble être une tâche difficile et incertaine. A ce niveau, il s'avère nécessaire de vérifier la bonne fonctionnalité des algorithmes préalablement proposés. Comme résultat, nous démontrons que quelques algorithmes sont erronés contredisant leur affirmation en termes de satisfaction de la propriété de convergence.

Dans ce chapitre, nous présentons, dans une première section, l'approche des transformées opérationnelles ainsi qu'un bref état de l'art des algorithmes les plus pertinents basés sur cette approche. Ensuite, nous proposons deux modèles, écrits en langage de programmation Java, décrivant le comportement des systèmes d'édition collaborative distribuée : Un modèle concret et un modèle symbolique abstrait. L'élaboration de ces deux modèles ainsi que la vérification d'une catégorie d'algorithmes

de transformées opérationnelles—algorithmes OT (Operational Transformation)—qui y sont appliqués seront discutées respectivement dans les sections 2 pour le modèle concret et 3 pour le modèle symbolique.

4.1 Transformées opérationnelles et algorithmes OT

La concurrence dans la manipulation d'objets partagés entre deux ou plusieurs utilisateurs ou sites est un phénomène fréquent quoique critique dans les systèmes d'édition collaborative distribuée.

Que ce soient des textes, des images ou des fichiers, ces objets partagés subissent une réplication dans la mémoire locale de chaque site afin de limiter les contraintes dans l'édition. Par la suite, une opération est dite locale dans un site considéré si elle est générée dans ce même site. En revanche, une opération est dite non locale dans un site considéré si elle est générée dans un autre site différent du site courant.

Basés sur ce principe, nous notons que chaque opération est exécutée localement en premier avant d'être diffusée pour son exécution dans les autres sites. Par conséquent, les opérations sont appliquées dans des ordres différents sur les différentes copies de chaque site. Ceci contribue à une divergence éventuelle probable entre les copies—aspect non désiré dans ce genre de systèmes.

Dans le but de contourner ce problème, l'approche de *transformées opérationnelles* a été proposée. Basée sur cette approche, toute opération non locale dans un site considéré doit subir une transformation suivant un algorithme de transformation donné avant son exécution sur ce même site. Cet algorithme est spécifiquement conçu dans le but d'assurer la propriété de convergence [34].

Cette section présente tout d'abord l'approche des transformées opérationnelles adoptée tout au long de l'édition collaborative suivie par une catégorie d'algorithmes prétendument assurant la synchronisation des documents partagés entre les sites.

4.1.1 Définition du concept de transformées opérationnelles

Dans les systèmes d'édition collaborative distribuée, chaque utilisateur possède sa propre copie "locale" du document sur laquelle diverses mises à jour peuvent être exécutées. Ces dernières seront par la suite transmises aux différents autres sites. C'est au niveau de cette réplication que la divergence entre les copies peut s'infiltrer.

Pour remédier à ce problème, une approche connue sous le nom de transformées opérationnelles permet de garantir la convergence de ces copies tout en transformant comme son nom l'indique les opérations non locales reçues dans chaque site.

Par définition, les transformées opérationnelles représentent une technique de réplication optimiste qui permet à plusieurs sites de mettre à jour le document partagé de façon concurrente. De plus, cette technique permet de synchroniser les différentes copies afin d'obtenir les mêmes données dans le document final quel que soit le site en question.

Avant d'aborder le processus d'interaction entre les différentes opérations pour une mise à jour efficace, nous définissons quelques notions de base des systèmes d'édition collaborative.

Description des composantes des systèmes d'édition collaborative

Objet partagé Nous manipulons dans notre étude de cas, et plus spécifiquement dans le modèle concret, un objet partagé ayant une structure linéaire. Cette structure est modélisée à l'aide d'une liste de la collection Java que nous avons appelée *text*, qui n'est autre qu'une liste d'entiers $ArrayList\langle Integer\rangle$. En effet, l'objet partagé que nous modélisons est un texte illimité d'entiers -1, 0 et 1. L'utilisation de ces entiers spécifiques est interprétée dans la prochaine section relative au modèle concret.

Toutefois, il reste à noter que cette notion d'objet partagé disparaît dans le modèle symbolique. Une abstraction, discutée plus loin dans ce chapitre, est envisagée dans le but de ne plus avoir à gérer des listes ou plus généralement l'objet partagé.

Opérations L'état de la liste mentionnée ci-haut peut être uniquement modifié après l'application de deux opérations de base :

1. L'insertion *Ins* qui permet d'insérer un élément donné dans la liste
2. La suppression *Del* qui permet de supprimer un élément de cette liste

L'implémentation de ces deux opérations sera discutée dans les sections 2 et 3. Cependant, la seule opération à rajouter est l'opération nulle *Nop* qui n'a aucun effet sur l'état de la liste.

Initialement, il existe un état de listes commun à tous les sites considérés. De plus, vu qu'il existe une copie répliquée dans tous les sites, chacun d'entre eux possèdera un état local de la liste, qui sera modifié par les opérations locales.

Un état *stable* est atteint dans le système lorsque toutes les opérations générées sont exécutées sur tous les sites [34].

Une fois les notions de base d'un tel système introduites, nous procédons dans ce qui suit à la définition des relations entre les opérations, nécessaires pour les transformées opérationnelles, telles que figurées dans [34].

Relations entre les opérations

Causalité Une relation de **causalité** peut exister entre deux opérations o_1 et o_2 respectivement générées dans les sites i et j. Ainsi, on dit qu'une opération o_2 dépend *causalement* de l'opération o_1, relation dénotée par $o_1 \rightarrow o_2$, si et seulement si l'une des deux conditions suivantes est satisfaite :

- $i = j$ (même site) et o_1 est générée avant o_2
- $i \neq j$ (sites différents) et l'exécution de o_1 dans le site j précède la génération de o_2

Concurrence Deux opérations o_1 et o_2 sont dites *concurrentes*, relation dénotée par $o_1 \parallel o_2$, si et seulement si les deux opérations sont indépendantes causalement ; en d'autres termes, on n'a ni $o_1 \rightarrow o_2$ ni $o_2 \rightarrow o_1$.

Il faut noter que les algorithmes OT ultérieurement appliqués sur ces opérations ne transforment que les paires d'opérations concurrentes comme développé dans la prochaine partie.

A ce niveau, nous pouvons résumer les avantages de l'approche de transformées opérationnelles au suivant :

- Elle est indépendante de l'état de la copie—état de la liste—et dépend uniquement des opérations concurrentes.
- Elle permet une concurrence sans contraintes au niveau des sites. En d'autres termes, il n'existe pas un ordre global précis sur les opérations.

Après avoir défini les opérations et les relations entre elles, nous évoquons le principe de transformation suivi, si nécessaire, tout au long de notre étude de cas.

Principe de transformation

Dans le but de maintenir la consistance au fur et à mesure des éditions concurrentes sur les différentes copies de chaque site/utilisateur, l'approche de transformées opérationnelles se base sur le principe suivant : Si un site A reçoit une opération o déjà exécutée par un autre site B sur sa propre copie de l'objet partagé, le site A ne doit pas nécessairement intégrer o en l'exécutant telle quelle sur sa copie. Par contre, ce dernier exécutera une variante de o, dénotée par $o^{'}$, la *transformée* de o, qui a comme intention de réaliser le même effet que o. Moyennant les algorithmes OT, cette approche est pratiquement mise en œuvre comme suit : Une opération déjà exécutée sur un autre site—opération non locale—est transformée selon l'ensemble d'opérations locales concurrentes. Cette démarche est discutée avec plus de détails un peu plus loin.

Critère de consistance

Un système d'éditions collaboratives basé sur les transformées opérationnelles est consistant si et seulement si :

- La relation de causalité est préservée sur tous les sites. Il est ainsi inacceptable d'exécuter o_1 après o_2 dans le cas où $o_1 \rightarrow o_2$ par exemple.

- La convergence est assurée : Après que l'ensemble commun des mises à jour est effectué dans les différents sites, les copies résultantes sont identiques.

D'une part, la préservation de la causalité entre les opérations est réalisée au moyen de deux techniques différentes relatives aux deux modèles concret et symbolique. D'autre part, la convergence est pratiquement assurée à l'aide d'une fonction de transformation IT intégrée dans les algorithmes OT discutés ci-bas.

4.1.2 État de l'art des algorithmes OT

Les algorithmes OT permettent aux différents sites correspondant aux utilisateurs d'échanger leurs copies modifiées dans un ordre quelconque vu que la convergence doit être satisfaite à l'aide de l'approche mentionnée ci-haut.

Pour ce faire, ces algorithmes s'appuient sur des propriétés bien établies de transformation basées sur une fonction de transformation IT (Integration). Ainsi, les algorithmes OT pourront être interchangeablement notés par algorithmes IT dans ce qui suit.

Propriétés de transformation

Nous présentons ici les propriétés de transformation telles que définies dans [34].

Soit *seq* une séquence d'opérations exécutées dans l'ordre o_1 , puis o_2 jusqu'à o_n. Cette séquence est notée par $[o_1; o_2; ...; o_n]$.

Transformer une opération o selon la séquence *seq* est exprimé à l'aide d'une fonction de transformation récursive *IT*(o,seq)* définie comme suit :

$$IT^*(o, []) = o \text{ où } [] \text{ est la séquence vide}$$

$$IT^*(o, [o_1; o_2; ...; o_n]) = IT^*(IT(o, o_1), [o_2; ...; o_n])$$

On dit ainsi que l'opération o a été générée de manière *concurrente* suivant toutes les opérations de *seq*.

Il reste à noter que la définition de la fonction IT est complétée par les deux égalités suivantes :

$$IT(Nop, o) = Nop$$

$$IT(o, Nop) = o$$

A ce niveau et avant de développer l'application de ces transformées opérationnelles moyennant les algorithmes OT, nous présentons un exemple illustratif dans les deux figures 4.1 et 4.2 décrivant le principe de transformation adopté dans un système d'éditions collaboratives. Cet exemple sert à démontrer l'importance des transformées opérationnelles dans l'exécution des opérations non locales pour assurer la convergence.

Exemple illustratif du principe de transformation Dans la signature d'une insertion, nous considérons uniquement la position suivie par le caractère à insérer : *Ins(p,c)* ; la signature d'une suppression comprend simplement la position dans le texte : *Del(p)*.

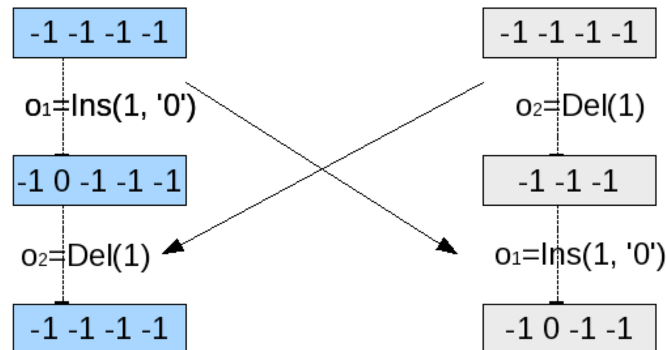

FIGURE 4.1 Divergence des copies due à l'absence de transformées opérationnelles

Dans cet exemple, nous considérons une taille limitée du texte qui consistera en quatre entiers contrairement à notre texte dans le modèle concret n'ayant aucune taille préfixée. Cette limitation est réalisée simplement à des fins d'illustration. De plus, il est à noter que l'état initial des textes est identique dans tous les sites : Les

quatre entiers sont initialisés à '-1'.

Il est à noter aussi que chaque insertion à la position p implique un décalage à droite de tous les entiers à une position égale ou supérieure à p. Il s'agit également d'une incrémentation de la taille du texte.

Par contre, la suppression à la position p implique un décalage à gauche de tous les entiers à une position strictement supérieure à p. L'entier à la position p étant totalement supprimé, la taille du texte est alors décrémenté.

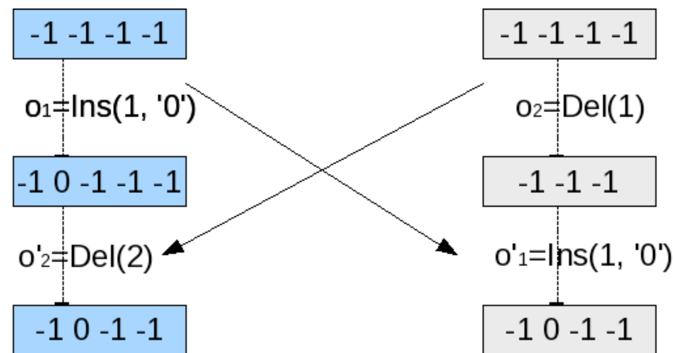

FIGURE 4.2 Convergence des copies après les transformations avec intégration

Dans la figure 4.1, deux utilisateurs ou sites modifient un document partagé représenté par le texte—une séquence d'entiers. Ces entiers sont indexés à partir de 0 jusqu'à la fin du texte.

L'utilisateur$_1$ (Site$_1$) exécute l'opération $o_1 = Ins(1,'0')$ pour insérer l'entier '0' dans la position 1 du texte. De manière concurrente, l'utilisateur$_2$ (Site$_2$) exécute l'opération $o_2 = Del(1)$ afin de supprimer l'entier à la position 1.

Quand o_1 est reçue et est exécutée dans le site$_2$, elle produit la séquence d'entiers désirée *"-1 0 -1 -1"*. Par contre, lorsque o_2 est reçue sur le site$_1$, cette dernière ne prend pas en considération que l'opération o_1 a été exécutée avant elle et elle produit alors la séquence *"-1 -1 -1 -1"*.

Comme résultat, le texte final dans le site$_1$ est différent de celui du site$_2$: Divergence des copies. L'intention dans o_2 de supprimer le second entier dans le texte

(position=1) devrait être mise à jour après la dernière insertion effectuée par o_1. Il ne s'agit plus de la position 1 dans ce cas mais de la position 2.

Cette incrémentation de la position, nécessaire pour la convergence des copies, est réalisée dans la figure 4.2. En effet, la transformation de l'opération o_2 dans le site$_1$, de *Del(1)* à *Del(2)*, s'appuie sur le principe de transformées opérationnelles et est effectuée à l'aide d'un algorithme IT quelconque vu leur équivalence dans la manipulation des situations (Ins ;Del) et (Del ;Ins) comme nous verrons plus loin dans cette section. Ces algorithmes se basent sur la fonction de transformation IT : $o'_2 = IT(o_2, o_1) = Del(2)$.

Il est à noter que l'opération o_1 est également transformée après sa réception dans le site$_2$. Toutefois, sa transformée aboutit à la même opération ($o'_1 = IT(o_1, o_2) = o_1$).

Cet exemple montre la nécessité d'utiliser les transformées opérationnelles lors de l'exécution des opérations non locales dans un site quelconque pour, au moins, respecter l'intention originale des utilisateurs. On aurait ainsi évité la violation de l'intention—un des problèmes d'inconsistance de ces systèmes— où l'effet actuel d'une opération est différent de l'effet désiré.

Afin d'assurer la convergence, les algorithmes OT, s'appuyant sur les transformées opérationnelles, doivent satisfaire deux conditions TP (Transformation Property), dites *propriétés de transformation*, notées **TP1** et **TP2** [34]. Pour toute opération o et toute paire d'opérations concurrentes o_1 et o_2 :

- Condition **TP1** : $[o_1; IT(o_2, o_1)] \equiv [o_2; IT(o_1, o_2)]$.
- Condition **TP2** : $IT^*(o, [o_1; IT(o_2, o_1)]) = IT^*(o, [o_2; IT(o_1, o_2)])$.

La condition **TP1** définit deux états identiques et garantit que si les opérations o_1 et o_2 sont concurrentes, l'ordre d'exécution de o_1 et o_2 n'affecte en rien le résultat final. Cette propriété est nécessaire mais non suffisante quand le nombre d'opérations concurrentes dépasse 2.

La condition **TP2**, quant à elle, garantit que la transformation de o selon des séquences d'opérations différentes mais équivalentes donne le même résultat.

La combinaison des deux conditions, **TP1** et **TP2** ensemble, est suffisante pour assurer la convergence et ce, quel que soit le nombre d'opérations concurrentes qui peuvent être exécutées dans un ordre *arbitraire*.

Par conséquent, tant que l'état final est un état stable et que ces deux propriétés sont satisfaites par les algorithmes OT utilisés, il n'est nullement nécessaire de préciser un ordre global entre les opérations concurrentes vu que la divergence des données est finalement rectifiée par l'algorithme OT en question.

Toutefois, concevoir un tel algorithme nécessite une analyse minutieuse des différents scénarios et situations possibles qui peuvent résulter en une convergence totale. Ce qui est désavantageux dans un environnement dominé par une infinité d'états possibles.

En outre, si les algorithmes se basant sur cette approche ne sont pas corrects alors la consistance entre ces copies n'est pas garantie. De ce fait, vérifier de tels algorithmes s'avère crucial dans le but d'éviter une perte d'information lors de la diffusion des opérations entre les différents sites.

Dans notre étude de cas, nous modélisons un système d'éditions collaboratives distribuées dans le but de vérifier une catégorie de cinq algorithmes OT au total. La partie suivante couvre la présentation de ces différents algorithmes avec leurs spécifités respectives.

Présentation des algorithmes IT

Avant d'aborder la présentation des algorithmes IT à vérifier, nous présentons la relation de **conflit** rencontrée dans l'ensemble des algorithmes considérés telle que définie dans [34]. Considérons deux opérations $o_1 = Ins(p_1, e_1)$ et $o_2 = Ins(p_2, e_2)$ où, d'un côté, p_1 et p_2 représentent les positions de l'insertion dans la liste et, d'un autre côté, e_1 et e_2 représentent les éléments à insérer dans cette liste. Les éléments considérés dans les algorithmes envisagés représentent des caractères. Ces deux opérations sont générées sur des sites différents.

On dit que o_1 et o_2 sont en *conflit* si et seulement si :

- $o_1 \parallel o_2$ et

- o_1 et o_2 sont générées sur le même état de liste et
- $p_1 = p_2$ (même position)

La notion de la relation de conflit étant définie, nous considérons ici, dans le cadre de nos travaux, les algorithmes IT les plus pertinents dans la littérature. Ces algorithmes sont au nombre de cinq et sont consécutivement présentés dans ce qui suit. On cite notamment les algorithmes d'Ellis, Ressel, Sun, Suleiman et Imine. Chaque algorithme considère les quatre cas ou situations de transformation à envisager relativement aux deux opérations de base : (Ins ;Ins), (Ins ;Del), (Del ;Ins) et (Del ;Del).

A noter que, par exemple, l'expression *IT(Ins(p_1,c_1), Ins(p_2,c_2))* représente la transformation de l'opération $o_1 = Ins(p_1, c_1)$ exécutée après l'opération $o_2 = Ins(p_2, c_2)$.

Algorithme d'Ellis Ellis et Gibbs sont les pionniers de l'approche OT. Dans leur algorithme illustré dans la figure 4.3, ils définissent les deux opérations d'édition de base comme suit :
- *Ins(p,c,pr)* : Insertion d'un caractère c à la position p avec la priorité pr
- *Del(p,pr)* : Suppression d'un caractère à la position p avec la priorité pr

Le paramètre pr dénote la priorité de l'opération calculée dans le site la générant— le site original. Chaque site a sa propre priorité qui est affectée aux opérations générées sur ce site. Ainsi, deux opérations générées par deux sites différents auront toujours deux priorités différentes.

Ce paramètre permet alors de résoudre la situation de conflit définie plus haut entre deux insertions de caractères différents à la même position comme indiquée dans les deux dernières conditions de la situation (Ins ;Ins) de la figure 4.3. Le résultat d'un tel cas est alors un décalage à droite de la position de l'insertion o_1 traduite en une incrémentation de la position (p_1+1) quand o_1 a une priorité plus élevée. Dans le cas contraire, l'opération o_1 reste intacte et est exécutée telle quelle sans aucune transformation.

Un autre cas spécial de la situation (Ins ;Ins) est lors d'une insertion du même caractère à la même position que la dernière opération concurrente dans la séquence

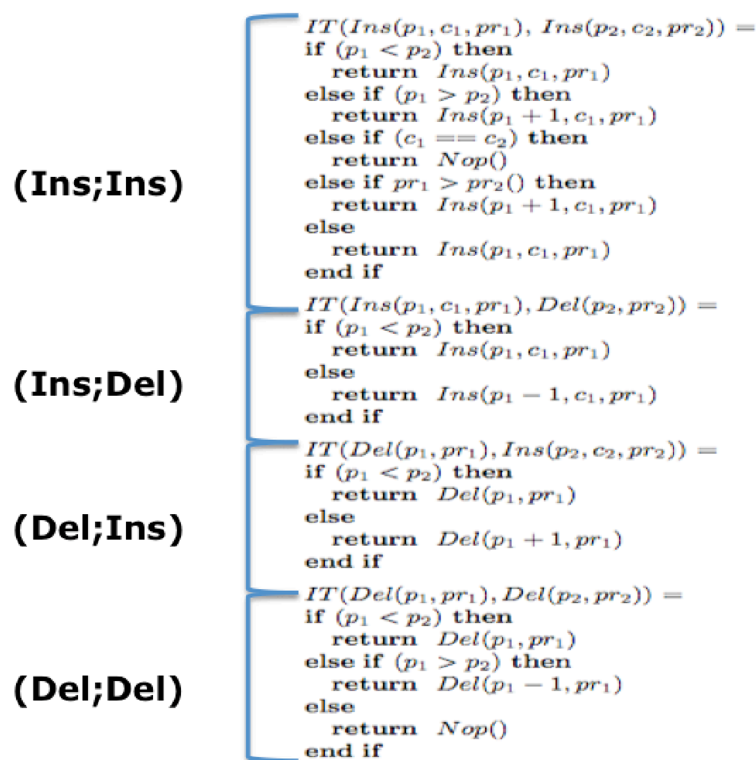

FIGURE 4.3 Algorithme IT d'Ellis (Source [34])

d'opérations. Ce cas résulte en une génération de l'opération *Nop* qui n'aura éventuellement aucun effet sur l'état de la liste.

Ce même résultat—opération *Nop*—est envisagé dans la situation (Del ;Del) dans le cas de deux suppressions concurrentes à la même position. La situation (Del ;Del) est identique pour tous les algorithmes IT considérés ici.

Les autres situations dans cet algorithmes sont simples et se limitent à une décrémentation ou une incrémentation de la position de l'opération envisagée correspondant respectivement à un décalage à gauche et à droite de cette position. A noter que les situations (Del ;Del), (Del ;Ins) et (Ins ;Del) sont identiques pour tous les algorithmes considérés dans notre étude de cas à l'exception de la dernière situation (Ins ;Del) pour l'algorithme de Suleiman discuté plus loin.

Algorithme de Ressel D'après divers travaux de vérification, il a été conclu que l'algorithme d'Ellis satisfait la propriété **TP1** mais non la propriété **TP2**. Ressel *et*

al. ont ainsi proposé deux modifications principales de l'algorithme d'Ellis.

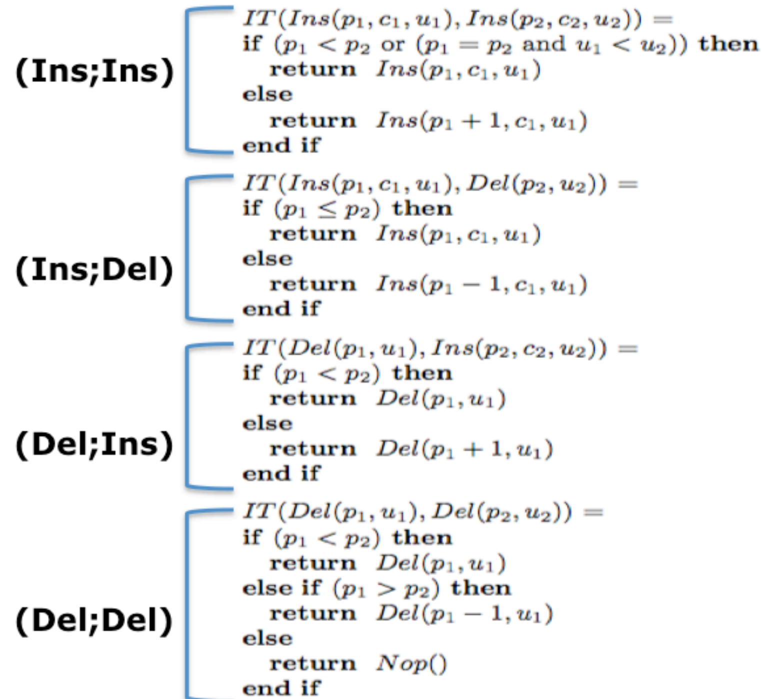

FIGURE 4.4 Algorithme IT de Ressel (Source [34])

Dans l'algorithme de Ressel illustré dans la figure 4.4, nous remarquons que le paramètre pr utilisé dans l'algorithme précédent est remplacé par un autre paramètre u. Ce dernier paramètre est simplement un identificateur du site original utilisé dans le but de résoudre la situation de conflit entre deux opérations concurrentes.

La seconde modification réside au niveau de la situation (Ins ;Ins) lors de deux insertions concurrentes à la même position. Que ce soit le même caractère ou pas, l'insertion est toujours exécutée à l'opposition de l'algorithme d'Ellis où l'opération résultante est l'opération nulle *Nop* s'il s'agit du même caractère inséré à la même position. Cette insertion peut être exécutée telle quelle si son identificateur u est strictement inférieur à la dernière opération concurrente de la séquence d'opérations. Dans le cas contraire, cette dernière opération est transformée en incrémentant sa position.

A l'exception de cette paire de modifications, les autres cas de transformation sont similaires à l'algorithme d'Ellis.

Algorithme de Sun L'algorithme de Sun illustré dans la figure 4.5 est pratiquement identique à celui de Ressel.

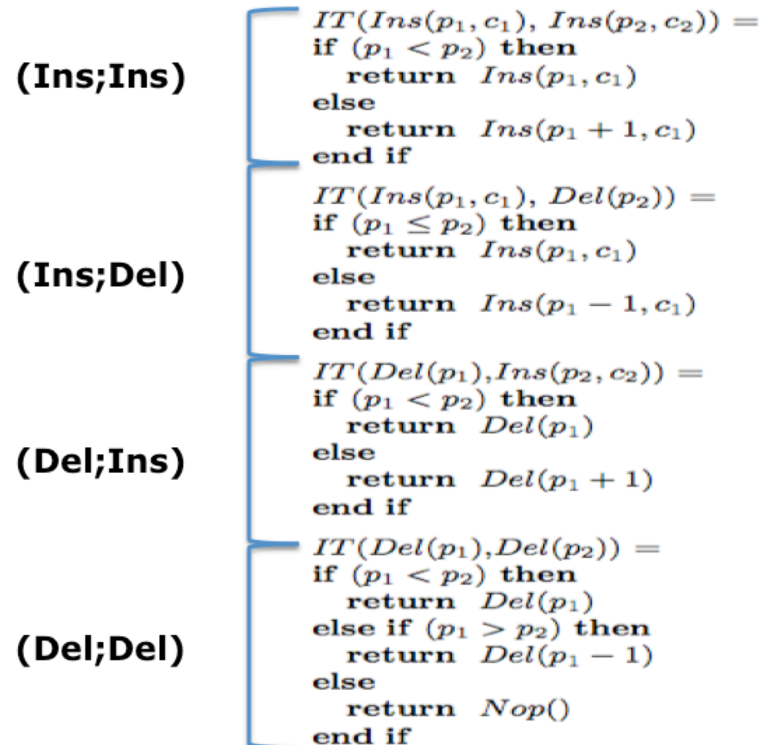

FIGURE 4.5 Algorithme IT de Sun (Source [34])

La seule différence réside dans le traitement de la situation de conflit entre deux opérations concurrentes. En effet, que ce soient deux insertions concurrentes à la même position (caractère identique ou pas), ou à une position de l'opération courante plus élevée, le résultat est le même : Décalage à droite de cette position. Si la position de l'opération courante est plus petite, l'opération est exécutée telle quelle sans aucune transformation.

Il n'y a plus, à ce niveau, une notion d'identificateurs de site comme paramètre de résolution de la situation de conflit vu que c'est à partir de la position que la transformation est achevée.

Algorithme de Suleiman Suleiman *et al.* suggèrent un algorithme de transformation illustré dans la figure 4.6 où la signature de l'opération d'insertion est modifiée en ajoutant deux paramètres *av* et *ap*. Ces derniers collectent l'ensemble des opérations concurrentes de suppression.

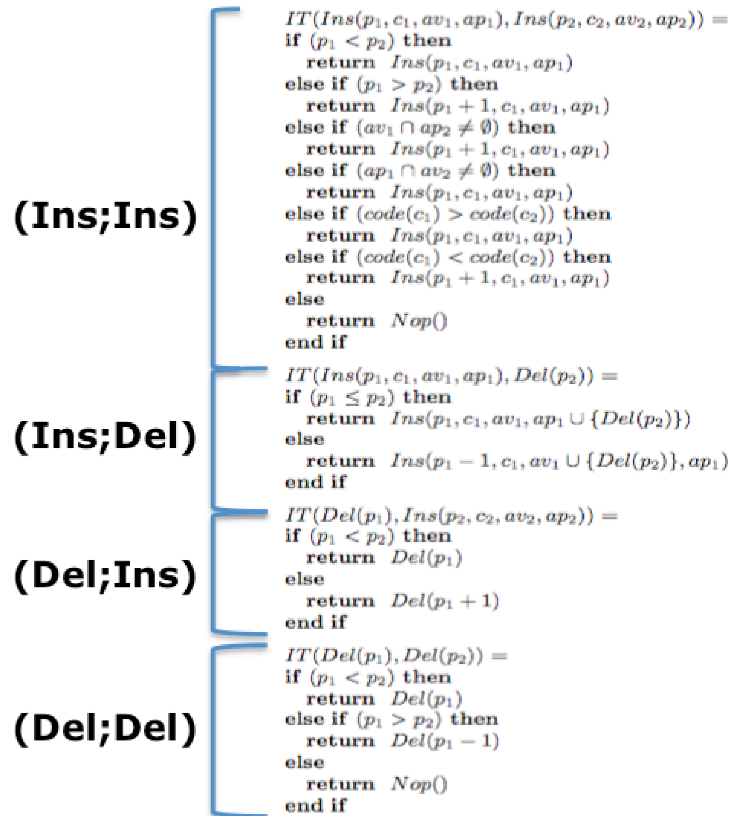

FIGURE 4.6 Algorithme IT de Suleiman (Source [34])

Pour toute opération *Ins(p,c,av,ap)*, *av* est une structure de données qui contient les opérations de suppressions de caractère exécutées *avant* la position de l'insertion *p* ; en d'autres termes, *av* contient les suppressions exécutées avant l'insertion avec des positions strictement inférieures à celle de l'insertion.

Similairement, la structure de données *ap* contient les suppressions exécutées avant l'insertion avec des positions supérieures ou égales à celle de l'insertion.

La génération d'une insertion crée automatiquement deux structures av et ap initialement vides. Ces dernières seront remplies au fur et à mesure des transformations exécutées dans la situation (Ins ;Del) de la figure 4.6, à l'aide des relations d'union '$\cup$'.

Basés sur la nouvelle signature des insertions, Suleiman *et al.* procèdent à la résolution du conflit résultant des deux insertions concurrentes $Ins(p,c_1,av_1,ap_1)$ et $Ins(p,c_2,av_2,ap_2)$ comme suit :

- $(av_1 \cap ap_2) \neq \emptyset$: Le caractère c_2 est inséré avant le caractère c_1
- $(ap_1 \cap av_2) \neq \emptyset$: Le caractère c_2 est inséré après le caractère c_1
- $(av_1 \cap ap_2) = (ap_1 \cap av_2) = \emptyset$. L'ordre d'insertion des caractères c_1 et c_2 est déterminé par la fonction *code(c)* qui se base sur un ordre lexicographique. C'est à l'aide de cette fonction que la situation de conflit entre deux insertions concurrentes est résolue.

A noter que, dans l'algorithme de Suleiman, deux insertions concurrentes identiques ($code(c_1) = code(c_2)$) résultent en une opération nulle *Nop* comme dans le cas d'Ellis.

De plus, l'algorithme de Suleiman est le seul algorithme différent par rapport aux autres algorithmes considérés relativement à la situation de transformation (Ins ;Del).

Algorithme d'Imine Dans [35], Imine *et al.* proposent une autre signature de l'opération d'insertion. Dans leur algorithme illustré dans la figure 4.7, toute opération d'insertion est définie par *Ins(p,o,c)* où p et o dénotent respectivement la position courante et la position originale de l'insertion telle que précisée lors de sa génération ; c n'est autre que le caractère à insérer.

Lors de la génération, les deux positions courante et originale sont identiques (p=o). C'est uniquement la position courante qui change dépendamment des transformations exécutées ; la position originale est une constante depuis la génération de l'insertion.

La position originale o de l'insertion sert comme paramètre de résolution de la situation de conflit entre deux opérations concurrentes d'insertion avec la même position. Dans le cas d'une position originale identique ($o_1 = o_2$), le paramètre utilisé est de nouveau la fonction *code(c)* de l'algorithme de Suleiman.

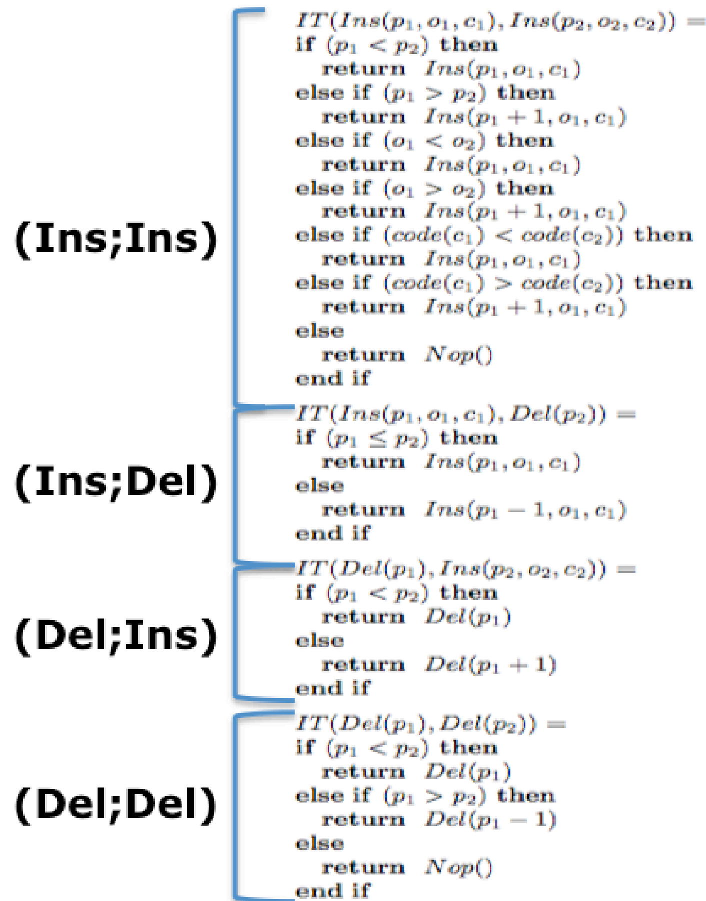

FIGURE 4.7 Algorithme IT d'Imine (Source [34])

Comme dans le cas de Suleiman et d'Ellis, deux insertions concurrentes identiques (positions courantes, positions originales et caractères identiques) résultent en une opération nulle *Nop*.

A ce niveau, les algorithmes à vérifier dans notre étude de cas sont concisément présentés dans cette section. En raison des différents scénarios à gérer, trouver puis prouver qu'un algorithme satisfait les deux conditions **TP1** et **TP2** nécessaires pour assurer la consistance du sytème d'édition collaborative considéré est une tâche compliquée voire impossible à produire à la main. D'où la nécessité de formaliser des modèles décrivant ces systèmes et d'automatiser de la sorte la vérification de la pro-

priété de consistance.

Dans les sections 2 et 3, nous présentons respectivement les deux modèles concret et abstrait suggérés pour la description de notre étude de cas.

4.2 Élaboration du modèle concret

Nous entamons cette section par une description informelle du comportement de notre étude de cas. Un système d'éditions collaboratives distribuées basé sur la réplication est typiquement composé de deux ou plusieurs sites ou utilisateurs. Ces derniers communiquent entre eux à travers un réseau et manipulent un objet partagé—un texte—au moyen de copies sous le principe de la réplication.

Initialement, chaque utilisateur possède une copie de l'objet partagé. Nous considérons un état *initial* de la copie **identique** dans tous les sites. Tout au long de l'édition collaborative, chaque utilisateur peut modifier sa copie en exécutant des opérations générées localement et également des opérations reçues par d'autres utilisateurs.

Notre modèle concret, élaboré dans le cadre de la présente section, traduit la description informelle de ce type de systèmes évoquée ci-haut en une description concrète pratique en langage de programmation Java : L'implémentation du comportement de ces systèmes ainsi que l'environnement d'exécution des algorithmes IT seront présentés dans ce qui suit.

Il est à noter que dans cette modélisation, il y a abstraction totale de la notion de réseau. C'est à partir des variables globales "static" en Java que la communication entre les sites via le réseau est modélisée.

Toutefois, avant de développer la modélisation de notre système en termes de variables, de méthodes et de classes Java, nous définissons quelques notions de base spécifiques à notre modèle concret. Ensuite, une fois le modèle concret implémenté pour simuler le comportement des sites et des transformations d'opérations à l'aide des algorithmes IT, la vérification de la propriété de consistance est rendue possible et ce, en utilisant l'exécution symbolique dans le model-checker SYMBOLIC JPF.

Comme résultat, nous affichons puis interprétons des extraits de cas de test en entrée, générés automatiquement, précisant les situations particulières de violation de la propriété de convergence pour chacun des cinq algorithmes considérés. Finalement, quelques limitations découlant du modèle concret seront présentées afin de justifier notre motivation à intégrer des abstractions dans ce modèle.

4.2.1 Notions de base

Notre modèle concret consiste en un objet partagé entre les sites d'une part, et en trois opérations de base d'autre part, dont deux sont utiles à des fins d'édition de l'objet partagé :

Objet partagé : L'objet partagé dans notre système représente un texte, plus précisément, un texte d'entiers. Il est modélisé par une structure linéaire à taille réglable non limitée à l'aide de la collection Java *text* déclarée comme suit : *"ArrayList⟨ArrayList⟨Integer⟩⟩ text"*.

En effet, le texte est à deux dimensions : La première dimension précise le site possédant la copie du texte tandis que la seconde dimension n'est autre que la position exacte dans cette structure linéaire. A noter que chaque site i peut exécuter les opérations sur sa propre copie du texte en appelant *"text.get(i)"* qui retourne la liste d'entiers *"ArrayList⟨Integer⟩"*.

L'état initial de cette copie doit être commun à tous les sites. Par conséquent, une initialisation particulière de l'objet partagé *text*, développée plus loin dans cette section, est nécessaire. Cette initialisation intègre des '-1' tout au long du texte afin de mettre en relief les modifications apportées ultérieurement par les opérations d'édition.

Il est important de noter que notre alphabet est restreinte aux *entiers* au lieu des chaînes de caractère. Vu que le caractère manipulé dans une opération d'insertion ou de suppression est intégré dans l'exécution symbolique et plus spécifiquement dans la génération des cas de test, il s'est avéré qu'il est forcément nécessaire de limiter l'alphabet aux entiers. Ces types de variables symboliques sont imposés par le model-checker SYMBOLIC JPF.

Cependant, nous imposons, sous forme de préconditions, une restriction supplémentaire aux deux entiers '0' et '1' au lieu de l'intervalle borné d'entiers positifs, vu qu'il suffit de manipuler dans ce cas deux choix différents pour discerner la divergence entre les copies. Cette technique est discutée dans une section prochaine relative à la vérification des propriétés par exécution symbolique.

Dans ce qui suit, les termes *caractère* et *entier* seront utilisés de façon interchangeable lors d'une insertion ou d'une suppression dans le texte.

Opérations Les opérations d'édition dans notre modèle concret sont représentées par la classe Java *"operation_type"* développée plus loin. Toutefois, dans notre système, il existe trois signatures globales d'opérations comprises dans une constante d'énumération *"enum operation {Del,Ins,Nop}"*. L'opération nulle, *Nop*, est générée après deux suppressions concurrentes identiques et/ou deux insertions concurrentes identiques dépendamment de l'algorithme IT considéré. Par ailleurs, dans tous les cas, cette opération n'a aucun effet sur l'état du texte partagé.

Il est à noter que, lors d'une insertion, nous pouvons manipuler les deux caractères '0' et '1' de l'alphabet. En revanche, lors d'une suppression, le caractère manipulé est non pertinent. Par conséquent, nous modifions le caractère associé à toute suppression au caractère '0'.

Dans la prochaine partie, nous mettons en relief les travaux relatifs à notre étude de cas, qui nous ont incités à la modélisation concrète du système en question.

4.2.2 Motivations

La revue de littérature, couvrant les systèmes d'édition collaborative basé sur la réplication, ne semble pas être étendue au niveau de la vérification formelle des algorithmes OT. D'après nos recherches élaborées avant tout intérêt dans le travail mené, nous avons pu discerner trois travaux principaux [34], [36] et [35] s'attaquant directement à ce problème.

Les autres travaux, dont on cite [37], s'appuient plutôt sur la gestion de la consistance dans ces systèmes en prônant la réplication *optimiste* au sein des réseaux non fiables : La notion de transformées opérationnelles n'est nullement intégrée, dans ce contexte, pour assurer la consistance de ces systèmes.

Par ailleurs, dans [38], Quiroga *et al.* modélisent une application, dans un système pair-à-pair, basée sur les transformées opérationnelles OT. Plus précisément, ils utilisent l'algorithme de Ressel *et al.* afin d'assurer une propagation transparente des modifications entre les pairs ainsi qu'une maintenance de la consistance. Cependant, aucune vérification de la convergence de cet algorithme n'est achevée dans le cadre de leur travail.

Dans [35], Imine *et al.* vérifient formellement l'exactitude des cinq algorithmes considérés dans notre étude de cas. Leur approche se base sur la traduction des fonctions de transformation IT de chaque algorithme en une logique du premier ordre. Ensuite, à l'aide du démonstrateur de théorèmes SPIKE basé sur des techniques de déduction automatiques avancées, des contre-exemples sont fournis dans le cas de violation de la propriété de consistance découlant de l'algorithme en question.

Similairement, dans [36], Imine *et al.* proposent une modélisation et une vérification formelles des algorithmes IT avec des spécifications algébriques. Afin de vérifier les propriétés **TP1** et **TP2**, ils optent pour le même démonstrateur de théorèmes, SPIKE. L'objectif principal de ce travail, basé sur [35], est à la fois de rendre simple la rédaction des spécifications formelles et d'ajouter un niveau d'automatisation plus élevé dans les procédures de preuves mathématiques inhérentes au démonstrateur de théorèmes qui nécessite, dans les conditions normales, un haut degré d'expertise.

Par conséquent, d'après leur travail, les concepteurs des algorithmes IT devraient être en mesure d'utiliser le démonstrateur de théorèmes comme un outil assez simple pour vérifier les conditions de convergence. Comme résultat, plusieurs contre-exemples de violation des deux conditions ont été fournis au niveau des cinq algorithmes IT.

Il est évident que toute approche basée sur un démonstrateur de théorèmes est adéquate pour la détection d'erreurs. Toutefois, cette technique reste limitée quant à la justification des contre-exemples fournis dans le cas d'une violation. En d'autres termes, ces derniers n'illustrent pas explicitement un scénario complet de divergence.

Par définition, un *scénario*, dans le contexte de notre système, comprend :

- Le nombre de sites ainsi que les opérations générées dans chacun de ces sites.
- Les ordres d'exécution montrant l'intégration des opérations sur chaque site.

La génération de ces scénarios est surtout pertinente dans la correction ultérieure des erreurs découlant de l'algorithme IT vérifié. De ce fait, un concepteur d'algorithmes IT pourrait vérifier facilement son propre algorithme avant d'affirmer son exactitude.

Cependant, bien que la technique de simplification dans la vérification de l'exactitude des algorithmes IT moyennant un démonstrateur de théorèmes a été achevée avec succès dans [36], ses contre-exemples fournis en sortie ne contribuent pas nécessairement à l'identification des erreurs dans l'algorithme considéré.

Ici se manifeste le rôle des model-checkers qui permettent de générer des scénarios détaillés tout en procédant à une vérification formelle automatique des algorithmes IT.

Dans [34], Boucheneb et Imine proposent une technique de model-checking, basée sur des formalismes utilisés dans l'outil UPPAAL, dans le but de modéliser le comportement des systèmes d'édition collaborative distribuée basés sur la réplication. Plusieurs abstractions et réductions ont été impliquées dans leur modèle afin d'atténuer le plus possible le problème d'explosion d'états. La vérification du modèle ainsi conçu a été achevée moyennant le model-checker intégré dans UPPAAL.

Dans le cas de violation d'une propriété, une trace d'exécution pour laquelle la propriété n'est pas satisfaite est fournie en sortie. A partir de ces résultats, Boucheneb *et al.* ont conclu qu'il existe une limite supérieure pour la convergence des données dans ce type de systèmes. En effet, d'aprés [34], pour un nombre de sites supérieur à 2, la propriété de convergence n'est pas satisfaite pour tous les algorithmes IT considérés dans notre étude de cas.

Cependant, une limitation dans le travail dirigé dans [34] réside dans le choix de l'outil UPPAAL. En effet, le model-checker intégré dans cet outil permet la génération non déterministe d'un seul contre-exemple pour chaque violation de propriétés. En d'autres termes, un scénario unique est généré pour chaque erreur dans l'algorithme IT en question. Bien que pertinent pour la correction de l'erreur rencontrée,

ce scénario peut être insuffisant pour la vérification de tout un système.

Notre travail, basé principalement sur [34], s'appuie sur le model-checking **combiné** à l'exécution symbolique moyennant l'outil SYMBOLIC JPF. Ainsi, la modélisation du système est achevée à l'aide du langage de programmation Java assez étendu pour exprimer le comportement de notre système. Cette étape a été basée sur celle de [34] quoique étendue sur plusieurs niveaux. La modélisation concrète est discutée en détail dans la partie suivante.

Comme résultat de la vérification de ce modèle, la génération "personnalisée" des cas de test en entrée indique tous les scénarios menant à une inconsistance du système relative à la divergence des copies dans chaque site : On parle plus spécifiquement d'*exécution symbolique exhaustive*.

Ce critère d'exécution justifie notre intérêt à opter pour l'exécution symbolique en plus du model-checking. De ce fait, nous gérons des variables symboliques comme les signatures des opérations et les caractères manipulés dans chacune d'entre elles. Cependant, à partir d'un nombre de sites supérieur à 4, notre modèle est limité—une conséquence de l'explosion de l'espace d'états. Cette limitation est discutée plus loin dans cette section.

Dans les prochains paragraphes, nous discutons d'un côté de la modélisation concrète du système en termes de constantes, de variables, de méthodes et de classes Java puis d'un autre côté de la vérification, par model-checking symbolique, des cinq algorithmes IT.

4.2.3 Description des constantes et variables d'entrée

Notre étude de cas comporte 7 constantes et 10 variables d'entrée au total. Leur définition ainsi que leur rôle dans la description du comportement de notre étude de cas sont discutés dans ce qui suit.

Constantes

Les constantes dans notre modèle représentent des paramètres indispensables à la description du système d'éditions collaboratives distribuées, notamment :

1. *NbSites* : nombre de sites représenté sous forme d'entiers "int"
2. *Iter* : nombre d'opérations locales dans chaque site. C'est un tableau d'entiers "int[]" dans lequel chaque indice désigne un site. Ainsi, *Iter[i]* n'est autre que le nombre d'opérations locales dans le site i
3. *MaxIter* : nombre total d'opérations dans tous les sites. C'est un entier "int" qui n'est autre que la somme des *Iters* de chaque site : $\sum_{i=0}^{NbSites-1} Iter[i]$
4. *algorithm* : liste des cinq algorithmes IT considérés dans le cadre de notre travail. C'est une énumération *enum algorithm* {*Ellis,Ressel,Sun,Suleiman,Imine*}
5. *algo* : algorithme IT à vérifier. C'est une constante faisant partie de l'énumération *algorithm* regroupant les cinq algorithmes IT . Ainsi, dans notre modèle, chaque exécution symbolique vérifie les propriétés de convergence relativement à un seul algorithme parmi les cinq.
6. *operation* : identificateur de l'opération. C'est une énumération "enum" pouvant prendre les valeurs suivantes *Del* pour une suppression, *Ins* pour une insertion et *Nop* pour une opération nulle. Ces identificateurs sont surtout utiles lors des considérations des différents cas de transformations IT dans chaque algorithme.
7. *L* : limite supérieure des positions possibles dans le texte. C'est un entier "int" exprimée par *2*MaxIter*. Bien que la taille du texte est illimitée dans notre modèle à l'opposition de sa taille fixe dans [34], l'attribut *position* de la signature d'opérations est restreint à l'intervalle *[0,L-1]*. Ce choix de borne supérieure est aléatoire mais est nécessaire afin d'atténuer le problème d'explosion d'états dans ces systèmes. Cet obstacle est inévitable en raison de la nature exhaustive de l'exécution symbolique dans la vérification de toutes les situations possibles lors des transformées opérationnelles.

Variables d'entrée

Notre modèle concret comporte deux catégories de variables d'entrée : 8 variables à type primitif et 2 à type prédéfini.

Tout d'abord, nous entamons cette partie par une définition ainsi qu'une description des variables à type primitif puis de celles à type prédéfini pour ensuite conclure avec la relation de causalité exprimée en fonction d'une ce ces variables d'entrée.

Variables à type primitif Il est à noter que les 5 premières variables d'entrée à type primitif représentent des attributs de la signature d'une opération alors que les autres décrivent quelques caractéristiques propres aux sites.

1. *alphabet* : caractère ou entier manipulé dans les opérations d'insertion ou de suppression. C'est un entier "int" dans l'intervalle [-1,1]
2. *operator* : identificateur spécifique de l'opération en question. Elle correspond à une constante dans l'énumération *operation* définie précédemment et est initialisée à l'opération *Del*
3. *position* : position courante de l'opération en question dans le texte. C'est un entier "int" entre *[0,L-1]*
4. *priority_user* : paramètre de résolution de la relation de conflit entre deux insertions concurrentes à la même position dans les deux algorithmes d'Ellis et de Ressel. C'est un entier "int" qui représente une priorité assignée à l'opération durant sa génération et l'identificateur du site la générant *pid* dans les cas d'Ellis et Ressel respectivement
5. *initial_position* : position originale de l'opération avant toute transformation. C'est un entier "int" qui représente un autre paramètre de résolution de la relation de conflit pour la même situation citée plus haut mais dans le cas d'Imine
6. *pid* : identificateur d'un site. Chaque site possède son propre identificateur *pid*, qui est un entier "int", dans l'intervalle *[0,NbSites-1]*. Par initialisation, *pid* est égal à -1. Ultérieurement, chaque construction d'un site incrémente de 1 l'identificateur *pid*
7. *text* : texte partagé entre les sites. Cette structure linéaire de type *ArrayList⟨ArrayList⟨Integer⟩⟩* a été préalablement définie dans les notions de base de cette section.

8. V : vecteur d'estampilles "timestamp". C'est un tableau d'entiers à deux dimensions *int[NbSites][MaxIter]* relatif au site considéré et à l'opération en question. Sa modélisation est justifiée par la nécessité de préserver la relation de causalité entre les opérations. Les relations de causalité et de concurrence entre les opérations relativement à ces vecteurs d'estampilles sont discutées un peu plus loin.

Variables à type prédéfini Les deux variables à type prédéfini *Operations* et *List* se basent sur des instanciations des classes respectives *operation_type* et *trace_type*.

Opérations C'est un tableau de taille constante égale au nombre d'opérations total *MaxIter* : *operation_type[MaxIter]*. Ce tableau collecte toutes les opérations originales sélectionnées par les différents sites, en d'autres termes, avant toute transformation. D'ailleurs, chaque élément de ce tableau n'est autre qu'une instanciation de la classe *operation_type* qui collecte tous les attributs d'une opération, notamment : les cinq variables de type primitifs du paragraphe précédent, un identificateur du site la générant *Owner* et finalement le vecteur d'estampilles de l'opération pour tous les sites considérés (tableau d'entiers *int[NbSites]*).

List C'est un tableau à deux dimensions, *trace_type[NbSites][MaxIter]*, servant à collecter les traces et les signatures des opérations telles qu'exécutées par chaque site. Chaque élément de ce tableau n'est autre qu'une instanciation de la classe *trace_type* qui comprend en plus des signatures d'opérations les deux vecteurs a pour "*after*" et b pour "*before*" (*Vector⟨trace_type⟩*) utilisés comme paramètres de résolution de la relation de conflit dans le cas de Suleiman. Ces deux vecteurs permettent de collecter les traces des suppressions exécutées respectivement après et avant l'opération en cours d'exécution. Il est à noter que cette classe comprend également une méthode "*isEqual*" indispensable dans la comparaison des suppressions en termes d'opérateurs et de positions lors des intersections des vecteurs a et b.

Relations de causalité dans notre système Les relations de causalité et conséquemment celles de concurrence, retenues de [34], sont définies en termes du vecteur

d'estampilles V comme suit :

Les systèmes distribués basés sur les OTs s'appuient sur ces vecteurs d'estampilles pour déterminer la causalité ou la concurrence entre deux opérations. Pour un site donné j, chaque entrée *V[i]* retourne le nombre d'opérations générées dans le site i et déjà exécutées dans le site j.

Lorsqu'une opération o est générée dans le site i, *V[i]* est incrémenté de 1. Une copie V_o de V est alors associée à l'opération o avant sa diffusion sur les autres sites. Une fois reçue par le site j, si la valeur du vecteur local V_{site} est supérieure ou égale à V_o, alors o est prête à être exécutée dans le site j. Dans ce cas, $V_{site}[i]$ est incrémenté de 1 après l'exécution de o. Sinon, l'exécution de o est retardée.

Basés sur ce principe, nous définissons, d'après [34], les relations de causalité et de concurrence dans les formules 4.1 et 4.2 repectivement.

Soient o_1 et o_2 deux opérations générées respectivement dans les sites s_{o_1} et s_{o_2} et dotées de leurs vecteurs d'estampilles respectifs V_{o_1} et V_{o_2}. Nous définissons dans 4.1 la relation de causalité et dans 4.2 celle de la concurrence. Cette dernière est implémentée dans le modèle Java à l'aide de la méthode "*Concurrent*" visualisée dans la figure 4.8.

$o_1 \rightarrow o_2$ ssi :

$$V_{o1[s_{o1}]} > V_{o2[s_{o1}]} \tag{4.1}$$

$o_1 \| o_2$ ssi :

$$V_{o1[s_{o1}]} \leq V_{o2[s_{o1}]} \; et \; V_{o1[s_{o2}]} \geq V_{o2[s_{o2}]} \tag{4.2}$$

A ce niveau, la présentation des constantes et des variables d'entrée de notre modèle concret est achevée. Dans la prochaine partie, nous menons une description des comportements des différents sites composant notre système.

```
public boolean Concurrent (trace_type op1, trace_type op2) {
        // o1 || o2 iff Vo1[So1] >=Vo2[So1] and Vo2[So2] >=Vo1[So2]
        if ( (Operations[op1.numOp].V[Operations[op1.numOp].Owner] >=
                Operations[op2.numOp].V[Operations[op1.numOp].Owner]) &&
                (Operations[op2.numOp].V[Operations[op2.numOp].Owner] >=
                    Operations[op1.numOp].V[Operations[op2.numOp].Owner]) )
            return true;

        return false;
    }
```

FIGURE 4.8 Méthode "*Concurrent*" implémentant la relation de concurrence

4.2.4 Comportements des sites

Les comportements des sites sont symétriques et représentés par des méthodes en Java au sein d'une classe nommée *Site*. La seule variable dans cette classe est l'identificateur du site *Sitepid*, qui est incrémenté de 1 à chaque construction d'un nouveau site comme déjà mentionné.

Chaque utilisateur ou site exécute toutes les opérations locales et non locales, une à la fois, sur sa propre copie du texte partagé. L'ordre d'exécution des opérations doit néanmoins satisfaire le principe de causalité assuré par les vecteurs d'estampilles V.

Présentation des méthodes Java

Afin d'aborder la description de l'exécution des opérations locales et non locales, nous présentons les 7 méthodes implémentant le comportement de chaque site :

– La méthode *"garde"*, illustrée dans la figure 4.9, prend comme paramètre l'identificateur d'un site k et retourne une valeur booléenne. Cette méthode sert à tester, par un résultat vrai ou faux, si un site considéré avec un identificateur *Sitepid* peut exécuter une opération d'un site k. Une opération locale, dénotée par $(Sitepid == k)$, peut être exécutée tant que le nombre actuel d'opérations locales exécutées ne dépasse pas le nombre maximal d'opérations locales à exécuter. Ceci est exprimé dans l'inégalité $V[Sitepid][Sitepid] < Iter[Sitepid]$. En revanche, une opération non locale peut être exécutée sous deux conditions : La première condition, exprimée par l'inégalité $V[Sitepid][k] < V[k][k]$, affirme que

le nombre actuel d'opérations non locales générées dans le site k et exécutées dans le site *Sitepid* ne dépasse pas le nombre maximal d'opérations locales au site k. La seconde condition, exprimée par *V[Sitepid][j]* < *Operations[i-1].V[j]*, reflète une violation du principe de causalité pour laquelle la réponse à l'exécution de l'opération est "false".

```
public boolean garde(int k) {
        if (Sitepid == k) //local site
             return V[Sitepid][Sitepid] < Iter[Sitepid];
          // case pid != k (remote site)
          if (V[Sitepid][k] < V[k][k]) {
             for (i=0,j=0; i<MaxIter && j<=V[Sitepid][k];i++ )
                 if (Operations[i].Owner == k)
                     j++;
             for (j=0;j<NbSites;j++)
                 if (V[Sitepid][j] < Operations[i-1].V[j])
                     return false;
             return true;
          }
          else //(V[Sitepid][k] >= V[k][k])
             return false;
       }
```

FIGURE 4.9 Méthode *"garde"* testant la possibilité d'exécution d'une opération

- La méthode *"Execution"* prend comme paramètre l'identificateur du site k générant l'opération à exécuter. L'appel de cette méthode est toutefois conditionné par une valeur "True" retournée par la méthode *"garde"*. Le rôle principal de cette méthode consiste à gérer la construction des traces et l'exécution des opérations en collectant l'identificateur et la position de l'opération à exécuter dans une variable de type *trace_ type*. Dans le cas d'une opération locale, le vecteur d'estampilles de l'opération en question est assigné à celui du site : *Operations[ns-1].V[i]* = *V[Sitepid][i]*.
  Ensuite, La méthode *"Execution"* appelle la méthode responsable des transformations *"Transformation"* avant de mettre à jour le vecteur d'estampilles propre au site : *V[Sitepid][k]++*. Cette dernière incrémentation permet de simuler la diffusion dans d'autres sites de l'opération qui vient d'être exécutée dans le site identifié par *Sitepid*.
- La méthode *"Transformation"*, illustrée dans la figure 4.10, déclenche la procédure de transformation IT dans le cas où l'opération est non locale et ce, en

appelant la méthode *"TransformR"*. Dans les deux cas, l'opération résultante est concrètement exécutée en appelant la méthode *"Operation"*.

```
public void Transformation(trace_type op, trace_type List[], ArrayList<Integer> t) {
        int i,len;

        for (i=0,len=0;i<NbSites;i++)
            len = len + V[Sitepid][i];
        if (len>0 && Sitepid != Operations[op.numOp].Owner) //not a local operation
            op=TransformR(op,List,len);

        Operation(op,List,len,t);
    }
```

FIGURE 4.10 Méthode "*Transformation*" déclenchant le processus de transformation

– La méthode *"Operation"*, illustrée dans la figure 4.11, implémente l'effet des opérations d'édition : Insertion *Ins* et suppression *Del*. Tant que la position de l'opération est comprise dans la taille du texte à modifier, l'effet d'une insertion via *"add"* déclenche implicitement un décalage à droite de toutes les positions égales et supérieures à la position courante de l'opération dans le texte. Une suppression est achevée via *"remove"* et résulte implicitement en un décalage à gauche de toutes les positions supérieures à la position courante dans le texte. La suppression dans notre cas se distingue de celle réalisée dans [34] dans la mesure où il n'y pas d'ajout de -1 à la fin du texte dans une suppression. Ceci a été imposé dans le but de garder une taille fixe du texte après toute opération, aspect non retrouvé dans notre cas vu que la taille du texte n'est pas limitée. Nous notons aussi que l'opération nulle *Nop*, n'ayant aucun effet sur le texte, est simplement ajouté après son exécution comme les deux autres opérations dans la trace des opérations *List*.
– La méthode *"TransformR"* est responsable du processus d'intégration des opérations non locales. L'intégration débute par un réordonnement des opérations déjà exécutées dans la trace *List*. Ce réordonnement, séparant les opérations concurrentes des opérations dépendantes, est achevé de façon récursive par la méthode *"TransformR"* en appelant la méthode *"ReOrder"* discutée plus bas. De ce fait, la méthode *"TransformR"* transforme en utilisant l'un des cinq algorithmes IT considérés l'opération en question relativement à la liste d'opérations

```
public void Operation(trace_type op, trace_type List[], int len, ArrayList<Integer> t) {
            if (op.oper != operation.Nop) {
                if (op.posC >=0 && op.posC <= t.size()) {
                        if (Operations[op.numOp].operator == operation.Ins) { //Insert operation
                            if (Operations[op.numOp].x==1)
                                t.add(op.posC,1);
                            else {
                                if (Operations[op.numOp].x==0)
                                    t.add(op.posC,0);
                            }
                        }
                        else {  //Delete operation
                            if (Operations[op.numOp].operator == operation.Del) {
                                if (op.posC!=t.size())
                                    t.remove(op.posC);
                            }
                        }
                }
            }

            List[len]=op;
            //else do nothing in case the position is not in the interval [0;L]
    }
```

FIGURE 4.11 Méthode "*Operation*" exécutant l'opération sur le texte partagé

concurrentes obtenue en respectant le principe de concurrence partielle définie plus loin.

- La méthode *"ReOrder"* permet de réordonner la liste d'opérations *List* en mettant les traces des opérations dépendantes au début de *List* pour ensuite y inclure les opérations concurrentes. Cette distinction entre les opérations s'appuie sur la méthode *"Concurrent"*. Lors du réordonnement des opérations, il s'avère nécessaire de créer deux listes *List2* et *List3* : La première permet de collecter les traces des opérations telles que générées alors que la seconde collecte les traces des opérations transformées. *List2* est surtout nécessaire pour la récupération des opérations originales dans le cas d'opérations résultantes *Nop*.
- La méthode "Concurrent", illustrée dans la figure 4.8, permet de déterminer la concurrence entre deux opérations o_1 et o_2 en se basant sur les vecteurs d'estampilles associés à chaque site les générant.

Problème de concurrence partielle

Le problème de concurrence partielle peut se manifester au sein d'un système d'éditions collaboratives dans la présence d'une séquence combinée d'opérations locales et non locales.

En effet, la définition du problème de concurrence partielle retenue de [34], se formule dans le suivant :

Deux opérations concurrentes o_1 et o_2 sont dites opérations partiellement concurrentes ssi o_1 est générée sur un état de liste l_1 dans le site 1 et o_2 est générée sur un état de liste l_2 dans le site 2 avec $l_1 \neq l_2$.

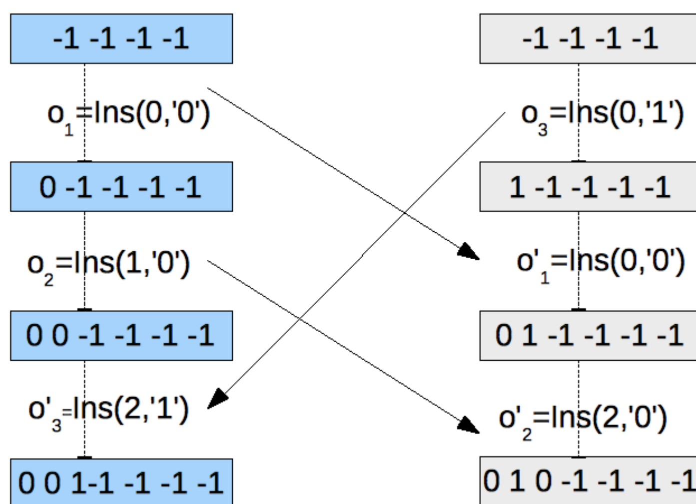

FIGURE 4.12 Situation de divergence due au problème de concurrence partielle

Afin d'interpréter plus concrètement la définition ci-haut, nous illustrons par un exemple une situation dominée par la concurrence partielle aboutissant à une divergence de données après les transformations IT dans la figure 4.12. Afin de remédier à ce problème, des appels récursifs de *"TransformR"* et implicitement de *"ReOrder"* sont réalisés aboutissant à une convergence finale des données. Cette technique est illustrée dans la figure 4.13.

Dans les deux figures, nous supposons que la taille initiale du texte est fixée à 4 afin de simplifier l'illustration. De plus, nous considérons l'algorithme d'Imine comme

algorithme des transfomées IT. Nous rappelons que la fonction de transformation IT ne manipule que les transformées entre des opérations concurrentes.

Dans la figure 4.12, nous considérons deux sites essayant de modifier le texte initialement commun *-1 -1 -1 -1*. Le site 1 génère deux opérations o_1 et o_2. De manière concurrente, le site 2 génère l'opération o_3.

Ainsi, nous avons dans cette situation, $o_1 \rightarrow o_2$ et $o_1 \| o_3$, mais o_2 et o_3 sont partiellement concurrentes vu qu'elles sont générées sur des listes à différents états. Dans le site 1, o_3 doit être transformée relativement à la séquence d'opérations locales $[o_1; o_2]$. En d'autres termes, $o'_3 = IT^*(o_3, [o_1; o_2]) = (Ins(2,' 1'))$. L'exécution de o'_3 donne comme résultat la séquence *0 0 1 -1 -1 -1 -1*.

Dans le site 2, transformer o_1 relativement à o_3 donne $o'_1 = IT(o_1, o_3) = o_1$ et transformer o_2 relativement à o_3 résulte en $o'_2 = IT(o_2, o_3) = Ins(2,' 0')$. L'exécution de o'_2 donne comme résultat la séquence *0 1 0 -1 -1 -1 -1* différente de la séquence obtenue dans le site 1. Cette situation de divergence est une conséquence d'une mauvaise application de la transformation IT des opérations o_2 et o_3 dans le site 2. En effet, la transformation IT nécessite que la paire d'opérations soit concurrente et définie sur le même état de liste. Toutefois, o_3 est générée sur la séquence *-1 -1 -1 -1* alors que o_2 est générée sur la séquence *0 -1 -1 -1 -1*.

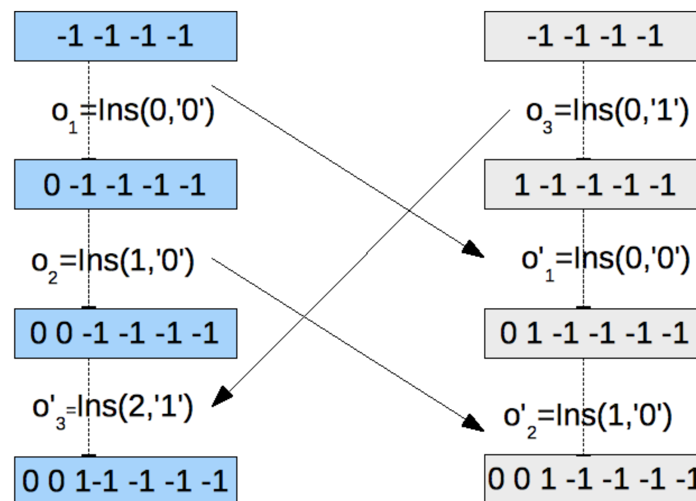

FIGURE 4.13 Solution au problème de concurrence partielle

Dans le but de remédier au problème de concurrence partielle, o_2 ne doit pas être

directement transformée relativement à o_3 vu que o_2 dépend causalement de o_1. Plutôt, o_3 doit être transformée relativement à o_1 et ensuite o_2 est transformée comme suit : $o_2' = IT(o_2, IT(o_3, o_1)) = Ins(1,'0')$. C'est à travers les méthodes *"TransformR"* et *"ReOrder"* que cette démarche est réalisée. La convergence des copies finales dans chaque site est ainsi obtenue comme illustrée dans la figure 4.13.

4.2.5 Intégration du modèle concret pour son exécution symbolique

Le modèle concret du système ainsi établi, comme illustré dans la figure 4.14, décrit le comportement symétrique de chaque site le composant—Site i avec $i \in [0, NbSites-1]$— et ce, vis-à-vis de la gestion des opérations locales et non locales et des effets de leur exécution sur le texte partagé.

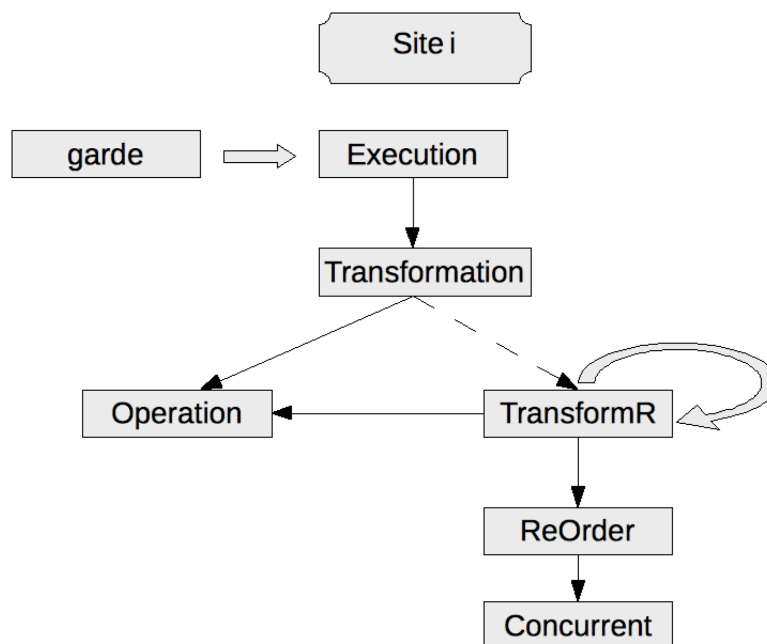

FIGURE 4.14 Modèle des méthodes internes à la classe *Site*

Nous présentons dans ce qui suit l'ensemble des classes Java intégrées dans le contexte d'exécution symbolique : Trois d'entre elles sont nécessaires pour le model-

checking et la génération de cas de test en entrée, alors que la quatrième sert à tester concrètement un cas de test obtenu pour illustrer les scénarios complets de divergence.

Élaboration de l'environnement pour l'exécution symbolique

La construction des sites dans le modèle concret est dûment réalisée à l'aide d'une méthode principale *"Execute"* qui prend comme paramètres les triplets suivants : les positions, les identificateurs ainsi que les caractères de l'ensemble des opérations dans tous les sites.

Après la construction d'un nombre constant et prédéfini de sites, *NbSites*, cette méthode principale déclare les opérations locales de chacun d'entre eux à l'aide des vecteurs de la forme $LocalOpsInSite = new\ ArrayList\ \langle Trio \rangle ()$, associant de la sorte chaque opération considérée dans le système au site la générant. Chaque opération est de ce fait représentée sous forme d'un triplet à l'aide de la classe *Trio* comprenant les paramètres de la méthode *"Execute"*, notamment la position d'édition, l'identificateur et le caractère de l'opération en question.

Après la déclaration des opérations, le texte partagé est initialisé à -1 dans l'intervalle *[0 ;L-1]* au niveau de chaque site. Comme déjà mentionné, cette initialisation est réalisée dans le but de mettre en relief les modifications ultérieures dans le texte dues aux éditions collaboratives. Il est à noter que l'intervalle d'initialisation est identique à celui des positions originales des opérations d'édition dans le texte. La restriction des positions originales des opérations générées est expliquée plus loin dans cette partie.

Une fois les opérations associées à chaque site, la méthode *"Execute"* procède à plusieurs démarches consécutives dans l'ordre suivant :

1. Exécution des opérations *locales* dans chaque site via la méthode *"Execution"* propre à chaque site et qui prend en paramètre l'identificateur des sites un à la fois. Cette exécution est conditionnée par la valeur "True" retournée de la méthode *"garde"* dans la classe *"Site"*.

2. Exécution des opérations *non locales* dans chaque site via la même méthode *"Execution"*. Les conditions de causalité s'appliquent en plus dans ce cas lors du conditionnement de leur exécution par la méthode *"garde"*.
3. Vérification de la propriété de convergence. Cette démarche finale, illustrée dans la figure 4.15, prend en considération à la fois l'état stable du système et la convergence des copies dans chaque site. Nous définissons la propriété de convergence dans le paragraphe suivant. Toutefois, nous notons que c'est la vérification de cette propriété qui suscite l'utilisation du model-checking combiné à l'exécution symbolique.

```
boolean convergent=false;
boolean finalresult=true;
    for (int i1: result) {
        for (int j1: result) {
            if (i1!=j1) {
                for (int k: result) {
                    if ( (V[i1][k] == V[k][k]) && (V[j1][k] == V[k][k]) ) {
                        convergent=true;
                        //Divergent texts: not the same size
                        if (text.get(i1).size() != text.get(j1).size()) {
                            convergent=false;
                        }
                        else {//same size
                            for (int l=0; l<=text.get(i1).size()-1; l++) {
                                if (text.get(i1).get(l) == text.get(j1).get(l))
                                    convergent=convergent&true;
                                else
                                    convergent=false;
                            }
                        }
                        if (!convergent )
                            finalresult= false;
                    }
                }
            }
        }
    }

//The convergence property is not satisfied
if (!finalresult) {
    jpfIfCounter++;
    path.append("The property convergence is not satisfied for Position1= " + pos1 +
            " and Position2=" + pos2 +" and Position3=" + pos3 /*+ " and Position4=" + pos4 */+ "\n");
}
Verify.ignoreIf(jpfIfCounter > 1);
CheckCoverage.processTestCase(path.toString());
```

FIGURE 4.15 Propriété de convergence implémentée pour l'exécution symbolique

Propriété de convergence La propriété principale à vérifier dans les systèmes d'édition collaborative est la *propriété de convergence*. Sous le principe de convergence, cette propriété affirme que, lorsque deux sites quelconques **achèvent** l'exécution d'un ensemble commun d'opérations, leurs textes résultants **doivent** être identiques.

Évidemment, cette formule intègre implicitement deux sous-formules : la stabilité du système et la convergence des copies.

Par définition, un site i est dit dans un état *stable* si toutes les opérations envoyées au site i sont reçues et exécutées par i. Relativement aux vecteurs d'estampilles simulant l'exécution des opérations dans un site, la stabilité du site i peut s'exprimer sous la forme : *V[i][k]==V[k][k]* pour tout $k \in$ *[0,NbSites-1]* [34].

Par conséquent, la propriété de convergence peut être reformulée au suivant : Lorsque deux sites i et j sont dans un état stable, ils possèdent des textes identiques.

Les deux notions de stabilité entre chaque paire de sites et de convergence des textes dans cette dernière sont implémentées en Java à la fin de la méthode *"Execute"* telles qu'illustrées dans la figure 4.15.

A ce niveau, il faut noter que la méthode principale *"Execute"* est une méthode appelée pour infiltrer ses paramètres dans l'exécution symbolique. De ce fait, dans le cas de violation de la propriété de convergence, exprimée par une valeur "false" de la valeur booléenne *finalresult*, la variable entière *jpfIfCounter*, qui est initialisée à 0 pour chaque exécution dans le model-checking symbolique, est incrémentée de 1. Ensuite, une phrase décrivant la violation de cette propriété en termes de positions concrètes des opérations en question est annexée à la chaîne de caractères *path* servant à la génération de cas de test. Nous filtrons cette génération aux scénarios de divergence, uniquement grâce à la méthode *Verify.ignoreIf(jpfIfCounter > 1)* vu que la variable *jpfIfCounter* reste égale à 0 si la propriété de convergence est satisfaite. Cette technique de filtrage des cas de test en entrée est également illustrée dans la figure 4.15.

Par la suite, nous intégrons le comportement des sites et le modèle concret du système respectivement implémentés dans les méthodes internes à la classe *Site* et

dans la méthode principale *"Execute"*.

Cette intégration de l'ensemble de ces méthodes résulte en une classe globale sous le nom
ConcreteModel_ OTReplicationAlgos. C'est la première classe indispensable pour l'exécution symbolique.

La seconde classe, *MyDriverforConcreteModel*, est la classe mère dans l'exécution symbolique comme nous verrons dans la configuration de l'exécution de la partie suivante. C'est cette classe qui, après avoir instancié un objet de la classe
ConcreteModel_ OTReplicationAlgos, déclenche l'exécution symbolique à partir des variables symboliques *oper* et *alpha* représentant respectivement les identificateurs des opérations ainsi que leurs caractères. Les valeurs de ces variables symboliques sont limitées à 0 et 1 pour les deux variables grâce à la méthode *imposePreconditions* exécutée symboliquement. En effet, c'est cette dernière méthode qui appelle la méthode *"Execute"* en lui passant les variables symboliques limitées suivant les préconditions qui se résument au suivant :

- Les identificateurs des opérations : $Ins \longrightarrow 1$ et $Del \longrightarrow 0$.
- Les caractères manipulés dans les insertions et suppressions : 0 et 1.

Nous notons ici que ces variables symboliques n'incluent pas les positions originales des opérations. En effet, en raison des transformées opérationnelles OT, il se peut que la position passée symboliquement soit incrémentée ou décrémentée. Or, dans un environnement d'exécution symbolique, les variables symboliques ne peuvent pas être manipulées arithmétiquement tout en leur assignant de nouvelles valeurs. Par exemple, si une variable symbolique est incrémentée, sa nouvelle valeur devient directement égale à 0.

Une telle limitation dans l'exécution symbolique nous a forcément dirigés vers une utilisation de variables concrètes des positions, consécutivement choisies dans des boucles de l'intervalle *[0,L-1]*. Cette démarche est identique au fait de passer à la méthode *imposePreconditions* les positions originales en termes de variables symboliques avec des préconditions entre 0 et *L-1*.

La troisième classe, nécessaire dans l'exécution symbolique, représente notre obser-

vateur *listener*, *JPF_ coverage_ CheckCoverage*, implémenté dans le but de surveiller les variables symboliques pour une génération "personnalisée" de cas de test en entrée. En effet, les cas de test obtenus dans le cas de violation de la propriété de convergence pour chaque algorithme IT, seront représentés sous un format **HTML** comme dans l'extrait affiché dans la figure 4.16. Chaque ligne désigne 1 cas de test pour une combinaison différente d'opérations. Les colonnes représentent de gauche à droite le numéro du cas du test suivi des identificateurs et des caractères des opérations alors que la dernière colonne comprend le chemin d'exécution, en d'autres termes la phrase annexée à *path* dans la méthode *"Execute"* décrivant la violation de la propriété pour les positions de chaque opération.

Testcase #	Operation1	Alphabet1	Operation2	Alphabet2	Operation3	Alphabet3	Path
1	Delete	0	Insert	1	Insert	0	The property convergence is not satisfied for Position1= 0 and Position2=0 and Position3=0

FIGURE 4.16 Extrait d'un cas de test en entrée

Finalement, une dernière classe supplémentaire, *ConcreteSimulation*, a été rajoutée à l'ensemble des trois classes précédentes afin de tester, si désiré, les contre-exemples préalablement obtenus dans les cas de test en entrée. De plus, cette classe est essentielle dans la rectification des erreurs provenant des algorithmes IT dans la mesure où l'affichage des scénarios complets de divergence est rendu possible. Nous rappelons qu'un scénario comprend d'une part le nombre de sites ainsi que les opérations générées dans chacun de ces sites et d'autre part les ordres d'exécution montrant l'intégration de chaque opération dans chaque site.

L'environnement d'exécution symbolique étant globalement présenté, nous procédons à la vérification de la propriété de convergence de notre modèle concret pour ensuite discuter des résultats obtenus pour les cinq algorithmes considérés dans le cadre de notre travail.

4.2.6 Vérification de la propriété de convergence

Nous afficherons, dans cette section, les résultats obtenus pour chacun des cinq algorithmes IT lors de la vérification des deux propriétés **TP1** et **TP2** définies dans une section antérieure. Nous notons que les expériences de vérification ont été menées sur une machine Intel Core2Duo de fréquence 2.0 GHz avec une mémoire vive de 2GB. De plus, le temps d'exécution (temps CPU), exprimé en secondes, est le temps moyen de cinq exécutions au total.

Le tableau 4.1 affiche les résultats de la vérification de ces deux propriétés pour chacun des cinq algorithmes IT dans le cadre d'une approche basée sur un démonstrateur de théorèmes dans [36].

Tableau 4.1 Résultats obtenus avec les démonstrateurs de théorèmes

Algorithmes IT	TP1	TP2
Ellis *et al.*	violée	violée
Ressel *et al.*	violée	violée
Sun *et al.*	violée	violée
Suleiman *et al.*	satisfaite	violée
Imine *et al.*	satisfaite	violée

Afin de comparer les résultats du tableau 4.1 avec nos résultats basés sur l'approche de model-checking symbolique, nous menons tout d'abord l'exécution symbolique du modèle concret pour un nombre de sites égal à 2, soit $NbSites = 2$ et un nombre total d'opérations égal à 2, soit $MaxIter = 2$. Plus précisément, nous vérifions dans ce cas la propriété **TP1** du système sous des conditions spéciales. Les résultats ainsi obtenus sont affichés dans le tableau 4.2.

Tableau 4.2 Résultats obtenus pour $NbSites=2$ et $MaxIter=2$

Algorithmes IT	Temps d'exécution	TP1
Ellis *et al.*	5.74	satisfaite
Ressel *et al.*	5.723	satisfaite
Sun *et al.*	5.80	violée
Suleiman *et al.*	5.86	satisfaite
Imine *et al.*	5.77	satisfaite

Nous distinguons alors un seul cas, celui de l'algorithme de Sun *et al.*, où il existe une violation de la propriété **TP1**. En effet, il y a génération de 8 contre-exemples ou cas de test violant cette propriété. Ces contre-exemples sont caractérisés par deux insertions de caractères différents à la même position. Vu qu'il s'agit de deux opérations au total ($MaxIter = 2$), nous aurons 4 positions possibles pour chaque opération ($L \in [0,(2*MaxIter)-1]$), d'où les 8 contre-exemples au total.

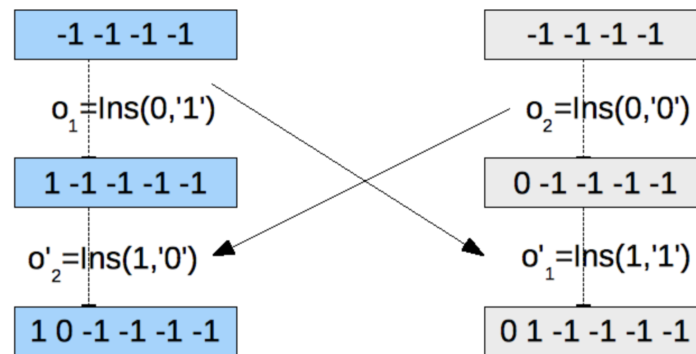

FIGURE 4.17 Scénario de divergence avec l'algorithme de Sun *et al.* pour $MaxIter = 2$

Dans la figure 4.17, nous illustrons un scénario de divergence obtenu dans l'exécution concrète de la classe *ConcreteSimulation*. Dans les scénarios de divergence, nous supposons que la signature des opérations comprend uniquement leur identificateur et leur position pour simplifier l'illustration. Nous notons que l'algorithme de Sun est le seul parmi les cinq autres algorithmes à ne pas traiter spécialement le cas de conflit entre deux insertions concurrentes à la même position.

En prenant toujours le même nombre de sites $NbSites = 2$ et le même nombre d'opérations concurrentes mais en ajoutant une autre opération dépendante dans l'un des deux sites, soit $MaxIter = 3$, nous obtenons les résultats tels qu'affichés dans le tableau 4.3. Ici aussi, nous testons la propriété **TP1** vu qu'il s'agit toujours de deux opérations concurrentes au total.

Nous mettons en relief le comportement symétrique des sites vu que les mêmes résultats sont obtenus dans le cas où le site 1 génère deux opérations dépendantes et le site 2 génère une opération concurrente ou bien dans le cas inverse.

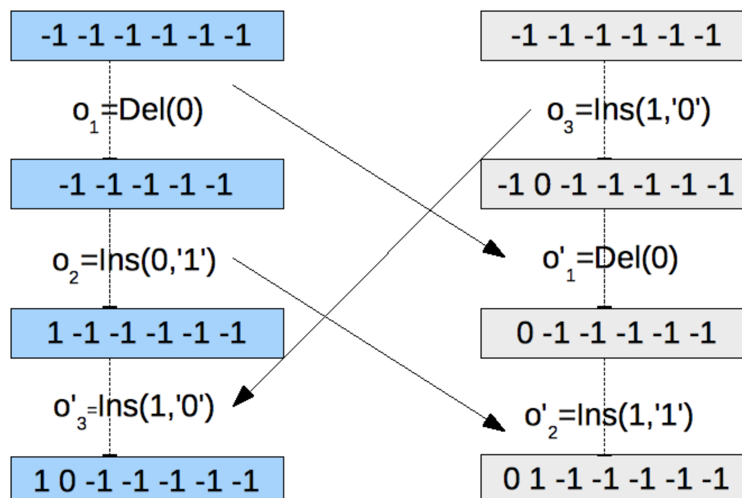

FIGURE 4.18 Scénario de divergence avec l'algorithme de Sun *et al.* pour $MaxIter = 3$

TABLEAU 4.3 Résultats obtenus pour *NbSites=2* et *MaxIter=3*

Algorithmes IT	Temps d'exécution	TP1
Ellis *et al.*	566	violée
Ressel *et al.*	558	violée
Sun *et al.*	602	violée
Suleiman *et al.*	574	satisfaite
Imine *et al.*	615	satisfaite

Nous remarquons que ces résultats concordent avec ceux obtenus dans le tableau 4.1. En prenant l'algorithme de Sun, nous illustrons dans la figure 4.18 un scénario de divergence complet dans le cas d'une suppression $o_1 = Del(0)$ et d'une insertion $o_2 = Ins(0,'1')$ dans le site 1 et d'une insertion $o_3 = Ins(1,'0')$ dans le site 2.

A ce niveau, nous passons à la vérification de la propriété **TP2**. De ce fait, tout d'abord, on passe à un nombre de sites égal à 3, soit $NbSites = 3$ et à un nombre total d'opérations égal à 3 puis à 4, soient respectivement $MaxIter = 3$ et $MaxIter = 4$. Ces résultats sont dressés respectivement dans les tableaux 4.4 et 4.5.

Des scénarios de divergence pour $MaxIter = 3$ avec l'algorithme de Ressel et pour $MaxIter = 4$ avec l'algorithme d'Imine sont affichés respectivement dans les figures 4.19 et 4.20.

TABLEAU 4.4 Résultats obtenus pour *NbSites* = *3* et *MaxIter* = *3*

Algorithmes IT	Temps d'exécution	TP2
Ellis *et al.*	560	violée
Ressel *et al.*	548	violée
Sun *et al.*	590	violée
Suleiman *et al.*	559	violée
Imine *et al.*	573	violée

TABLEAU 4.5 Résultats obtenus pour *NbSites* = *3* et *MaxIter* = *4*

Algorithmes IT	Temps d'exécution	TP2
Ellis *et al.*	602	violée
Ressel *et al.*	594	violée
Sun *et al.*	613	violée
Suleiman *et al.*	572	violée
Imine *et al.*	589	violée

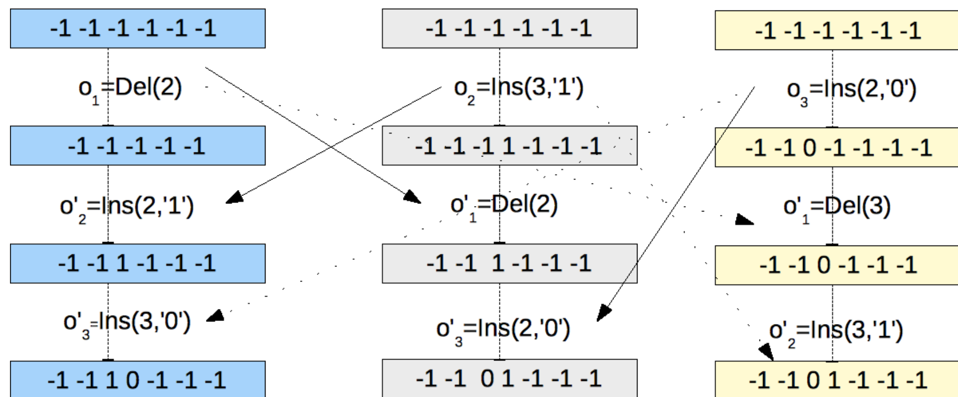

FIGURE 4.19 Scénario de divergence avec l'algorithme de Ressel *et al.*

En somme, nous pouvons conclure que nos résultats concordent avec ceux obtenus dans [34] et [36] : Pour un nombre de sites plus élevé que 2, tous les algorithmes IT considérés ne satisfont pas la propriété de convergence. En effet, quand le nombre de sites dépasse 2, le nombre d'opérations concurrentes dépasse 2 systématiquement. Ce qui signifie que la propriété **TP2** n'est pas satisfaite. Cette condition étant nécessaire pour la convergence du système, nous pouvons conclure que notre système n'est pas consistant pour *NbSites* $>$ *2* et ce, quel que soit l'algorithme IT en question.

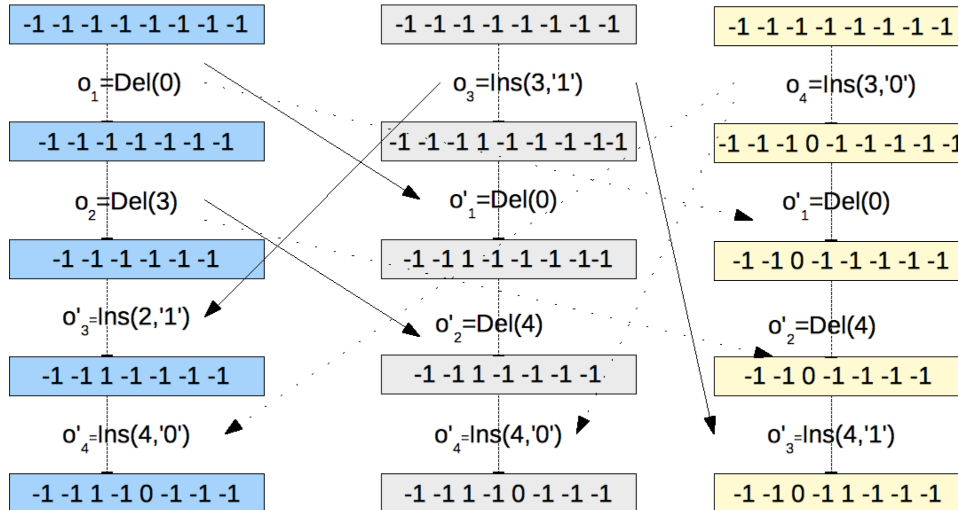

FIGURE 4.20 Scénario de divergence avec l'algorithme d'Imine *et al.*

Toutefois, notre approche est avantageuse relativement aux approches précédentes dans la mesure où elle contribue à retracer le scénario de divergence au complet pour toute violation de propriété, et ce quel que soit l'algorithme considéré. Cette démarche est intéressante dans un contexte de rectification des algorithmes IT. Toutefois, au terme de ce travail, nous n'apportons aucune modification aux algorithmes violant les propriétés **TP1** et **TP2**.

4.2.7 Limitations du modèle concret

Bien que les résultats obtenus répondent à notre objectif principal de vérification de la consistance de notre étude de cas de manière efficace, notre modèle concret demeure plutôt limité. En effet, pour un nombre de sites égal à 4, soit $NbSites = 4$, l'exécution symbolique moyennant le model-checker SYMBOLIC JPF prend plus que 20 minutes comme temps d'exécution. Ceci est principalement dû à l'aspect *exhaustif* de l'exécution qui essaie toutes les combinaisons possibles d'identificateurs d'opérations, de positions et de caractères pour chaque algorithme IT vérifié. Pour un nombre de sites plus élevé que 4, l'exécution souffre d'un manque de mémoire, ce problème est une conséquence de l'explosion d'états dans ce type de systèmes interactifs.

Une seconde limitation du modèle concret réside dans l'ordre global prédéfini des

opérations : Il y a exécution des opérations locales en premier et ensuite des opérations non locales. Afin d'analyser l'impact de l'ordre d'exécution entre les opérations, nous optons à une exécution dans tous les ordres acceptables dans le modèle prochain, octroyant un aspect plus générique et certes plus réaliste à l'édition collaborative.

La vérification de la consistance du système suivant tous les ordres d'exécution possibles acquiert alors un niveau de confiance plus élevé vis-à-vis des résultats obtenus, surtout dans un cas redoutable où une erreur n'est pas trouvée en raison d'un ordre d'exécution préalablement non considéré.

L'implémentation de cette exécution combinée à diverses abstractions au niveau des variables intégrées dans la vérification permet de passer d'un modèle *concret* à un modèle *abstrait*. L'élaboration de ce modèle ainsi que la description de ses caractéristiques sont développées dans la section suivante.

4.3 Abstractions dans le modèle concret : Modèle abstrait

Comme déjà mentionné, l'une des limitations de notre modèle concret est engendrée par la nature exhaustive de l'exécution symbolique avec le model-checker SYMBOLIC JPF. Afin d'atténuer le problème d'explosion d'états rencontré dans ce type d'exécution, nous avons opté, dans le modèle concret, à une restriction du domaine des variables intégrées dans la vérification à l'aide de préconditions sur ces variables, plus spécifiquement sur les positions des opérations et sur les caractères à insérer.

Toutefois, cette restriction des domaines infinis des positions et des caractères peut ignorer des cas de violation des propriétés de convergence lors de la vérification de notre système. De ce fait, nous désirons conserver toutes les valeurs possibles des variables tout en utilisant une représentation intégrant des abstractions dans le modèle concret.

L'aspiration aux abstractions au niveau des variables composant le système d'édi-

tions collaboratives distribuées nous incite à opter pour une représentation du modèle décrivant le système avec des *contraintes libres* : Les positions des opérations ainsi que les caractères à insérer peuvent prendre des valeurs naturelles aléatoires, en d'autres termes, pour tout p et pour tout c, nous aurons :

$$[p \in \mathbb{N}] \equiv [0 \leq p \leq \infty] \text{ et } [c \in \mathbb{N}] \equiv [0 \leq c \leq \infty]$$

De plus, les transformées opérationnelles OT caractérisant notre système d'éditions collaboratives engendrent une nécessité de gérer des paires d'opérations pour tout algorithme IT envisagé. Nous gérons des relations de comparaison $(<, \leq, =, >, \geq)$ entre les attributs d'opérations plutôt que des variables absolues. On passe alors aux contraintes de base suivantes pour toute paire de positions $(\mathrm{p}_i,\mathrm{p}_j)$ et toute paire de caractères $(\mathrm{c}_i,\mathrm{c}_j)$:

$$[-\infty \leq p_i - p_j \leq \infty] \text{ et } [-\infty \leq c_i - c_j \leq \infty]$$

Cette représentation abstraite de notre système à l'aide de contraintes nous suscite à l'intégration des matrices de bornes, notées DBMs (Difference-Bound Matrices), dans la modélisation de notre système. Cette technique adoptée tout au long du modèle abstrait est développée dans ce qui suit.

4.3.1 Intégration des matrices de bornes DBMs dans le modèle

Introduits par Dill [39], les matrices de bornes DBMs sont des structures de données permettant de représenter et manipuler de manière efficace un ensemble de contraintes sur des variables avec des domaines infinis, de la forme $x_i - x_j \leq c$ où $c \in \mathbb{Z}$.

Les DBMs ont été conçus dans le but principal de vérifier formellement des automates temporisés par model-checking. Par conséquent, les variables intégrées dans ces DBMs représentent des contraintes temporelles sur les *horloges* dans un système temporisé.

Toutefois, nous procédons dans notre étude de cas à utiliser les DBMs dans un contexte spécial : A partir des matrices de bornes, nous manipulons un ensemble de

contraintes sur les variables de notre système de la même forme $x_i - x_j \leq c$. Les variables que nous représenterons par ces contraintes, appelées dans ce cas *contraintes potentielles*, comprennent tous les attributs d'une opération considérée, notamment les positions originales et positions transformées, les caractères et les paramètres de résolution du conflit des insertions concurrentes à la même position.

Ici se manifeste l'avantage crucial conféré par les matrices de bornes dans la modélisation de notre système : Les limitations des domaines infinis, rencontrées dans le model-checking symbolique avec l'outil SYMBOLIC JPF dans l'absence d'impositions de préconditions, seront ainsi résolues comme nous verrons dans les étapes prochaines.

Du modèle concret au modèle abstrait

Avant d'élaborer l'implémentation des matrices de bornes dans notre modèle, nous notons qu'en se basant sur les DBMs, les notions de base à partir desquelles nous avons élaboré notre modèle concret disparaissent totalement dans le modèle abstrait. En effet, il n'existe plus de notions d'**objet partagé** sous forme concrète de liste ou de texte ni d'**opérations** avec des attributs symboliques (quoique exécutées "concrètement" de manière exhaustive).

Par conséquent, à la différence du modèle concret, la signature des opérations n'inclut pas la position de l'opération ni son caractère mais uniquement son opérateur qui peut être une insertion *Ins* ou une suppression *Del*.

De plus, vu qu'il n'existe plus de texte concret à modifier dans les différents sites de ce modèle, l'exécution de ces opérations sur le texte devient impossible concrètement. On passe alors à une exécution *abstraite* des opérations, simulée par de simples modifications des positions des opérations exécutées.

La traduction des comportements des sites et des transformées opérationnelles du modèle concret au modèle abstrait ainsi que l'approche d'exécution abstraite seront discutées dans cette section. Toutefois, nous précisons que notre objectif dans le modèle abstrait est identique à celui dans le modèle concret et consiste en une vérification de la consistance de notre étude de cas à partir des cinq algorithmes IT

préalablement définis.

Représentation des comportements des sites Afin de représenter notre système envisagé pour la vérification, nous définissons dans un premier temps des constantes globales utilisées comme paramètres d'exécution du modèle abstrait. Ces constantes de base définies par l'utilisateur sont catégorisées comme suit :

1. Le nombre total de sites, *NbSites*
2. Le nombre total d'opérations, *NbOperations*
3. Les opérations locales dans chaque site. Chaque opération locale générée est définie en termes de son opérateur *op* (qui peut être *Ins* ou *Del*), son identificateur *id* qui est un entier nécessaire pour l'accéder dans la matrice de bornes et finalement son site original *flag* qui est un entier identifiant le site générant l'opération et pouvant représenter dans certains algorithmes le paramètre de résolution de la situation de conflit entre deux insertions concurrentes.
4. L'algorithme IT à vérifier : Ellis, Ressel, Sun, Suleiman ou Imine

Dans un deuxième temps, nous définissons un *état* dans chaque site du système.

Dans notre modèle concret, chaque état particulier d'un site était représenté relativement aux opérations locales, aux transformées d'opérations non locales et aux modifications d'éditions résultantes dans le texte. En revanche, dans notre modèle abstrait, chaque état d'un site est représenté par une matrice de bornes.

Définition des matrices de bornes Nous définissons une matrice de bornes comme étant une matrice carrée de dimension M, M étant la dimension des lignes i et des colonnes j avec $0 \leq (i, j) \leq M - 1$.

Chaque élément d'une matrice de bornes est défini comme suit :

$$m_{ij} \triangleq \begin{cases} (\leq; 0) & \text{si i = j} \\ (\upsilon; c) & \text{si } (i \neq j) \text{ et } (\ x_i - x_j)\ \upsilon \text{ c} \\ (\leq; +\infty) & \text{sinon} \end{cases}$$

En tant que matrice carrée, les lignes et les colonnes de nos matrices de bornes représentent les mêmes variables. Ces variables intégrées dans les *contraintes potentielles*, telles que mentionnées précédemment, dénotent ainsi des relations d'inégalité entre elles. Ce qui explique que tous les éléments de la diagonale de nos matrices de bornes sont égaux à la paire ($\leq$;0) dénotant l'égalité ($x_i = x_j$) correspondant aux deux contraintes $x_i \leq x_j$ et $x_j \leq x_i$ pour $i = j$.

De plus, dans l'absence de contraintes potentielles entre les variables x_i et x_j, chaque élément (m_{ij}) de la matrice est égal à ($\leq$; $+\infty$). Cette valeur a été implémentée par la constante *Integer.MAX_ VALUE* du langage Java. Il est à noter que, pour tout $x \in \mathbb{Z}$:

$$\begin{cases} x + \infty = +\infty \\ x - \infty = -\infty \end{cases}$$

Par contre, dans le cas d'une contrainte potentielle de la forme, $(x_i - x_j)\ \upsilon$ c, l'élément m_{ij} de la matrice de bornes devient une paire $(\upsilon\,;c)$ pour tout $c \in \mathbb{Z}$ avec :

$$\upsilon = \begin{cases} < \\ \leq \end{cases}$$

Une fois les notions de base de nos matrices de bornes définies, nous présentons dans les paragraphes suivants leur implémentation pour les cinq algorithmes IT considérés. Cette catégorisation dans l'implémentation de ces matrices découle du fait que les lignes et les colonnes constituant la matrice en question dépendent de l'algorithme IT choisi. De plus, à des fins d'illustration, nous supposons dans la suite que les constantes globales se réduisent à la liste suivante pour tout algorithme IT précis :

- *NbSites = 2*
- *NbOperations = 3*
- Il existe deux opérations dépendantes o_1 et o_2 dans le site 1 et une opération o_3 dans le site 2. Cette dernière opération est concurrente à o_1 et partiellement concurrente à o_2. Nous considérons que les trois opérations sont des *Ins* et que leur variable *flag* est 1 pour les 2 opérations o_1, o_2 (Site 1) et 2 pour l'opération o_3 (Site 2).

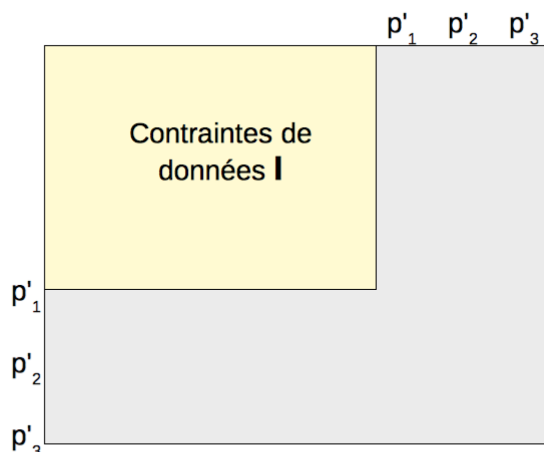

FIGURE 4.21 Format général d'une matrice de bornes

Nous affichons dans la figure 4.21 le format général de toute matrice de bornes quel que soit l'algorithme IT considéré. La sous-matrice I représente les contraintes de données et est relative à l'algorithme IT. Cette sous-matrice à laquelle nous rajouterons les positions transformées, qu'on dénotera par *p'*, représentera à la fois l'ensemble des lignes et des colonnes de la matrice de bornes globale correspondante. Dans tout algorithme IT, la position transformée *p'* est exprimée en fonction de la position courante p sous 3 cas différents :

$$p' = \begin{cases} p & \text{par défaut} \\ p+1 & \\ p-1 & \end{cases}$$

Algorithme d'Ellis Suivant l'algorithme d'Ellis de la figure 4.3, nous remarquons que les variables pertinentes à la détermination des opérations transformées sont les positions courantes p, les caractères c et les paramètres de résolution des situations de conflit pr dénotant la priorité assignée à chaque opération. Ces variables représentent la sous-matrice I de la figure 4.21.

L'implémentation de l'algorithme d'Ellis se traduit ainsi à la matrice de bornes *initiale* de la figure 4.22.

	p_1	p_2	p_3	c_1	c_2	c_3	o_1	o_2	o_3	pr_1	pr_2	pr_3	p'_1	p'_2	p'_3
p_1	(≤;0)												(≤;0)		
p_2		(≤;0)												(≤;0)	
p_3			(≤;0)												(≤;0)
c_1				(≤;0)											
c_2					(≤;0)										
c_3						(≤;0)									
o_1							(≤;0)								
o_2								(≤;0)							
o_3									(≤;0)						
pr_1										(≤;0)	(≤;0)	(<;0)			
pr_2										(≤;0)	(≤;0)	(<;0)			
pr_3												(≤;0)			
p'_1	(≤;0)												(≤;0)		
p'_2		(≤;0)												(≤;0)	
p'_3			(≤;0)												(≤;0)

FIGURE 4.22 Matrice de bornes initiale correspondant à l'algorithme d'Ellis

Nous remarquons que nous avons ajouté à l'ensemble des variables de la sous-matrice I les positions originales o de chaque opération. Ces variables sont utilisées à des fins d'affichage dans les contre-exemples ultérieurement obtenus s'ils existent et figurent également dans la sous-matrice I pour toute implémentation des matrices de bornes quel que soit l'algorithme IT en question.

De plus, nous notons que les cases vides représentent les éléments de la matrice sans contraintes potentielles entre les variables ; en d'autres termes, ce sont les paires $(\leq; +\infty)$.

Dans toute matrice de bornes initiale, nous assignons la paire $(\leq; 0)$ à toutes les positions transformées *p'* relativement à leur position originale vu que nous avons initialement :

- $[p'_1 = p_1] \equiv [p'_1 \leq p_1 \text{ et } p_1 \leq p'_1]$
- $[p'_2 = p_2] \equiv [p'_2 \leq p_2 \text{ et } p_2 \leq p'_2]$
- $[p'_3 = p_3] \equiv [p'_3 \leq p_3 \text{ et } p_3 \leq p'_3]$

Ces éléments pourront être modifiés dans cette matrice au fur at à mesure de l'application des transformées opérationnelles entre chaque paire d'opérations.

Finalement, nous précisons que les contraintes entre les paramètres de résolution de la situation de conflit pr_1, pr_2 et pr_3 sont fixées dans chaque matrice de bornes. Ceci résulte du fait que ces priorités sont des constantes définies lors de la construction des opérations. Il ne s'agit pas dans ce cas de construire ou de mettre à jour des contraintes pour ces constantes. Par contre, il s'agit d'une pure comparaison de leurs valeurs pour chaque transformation d'une paire d'opérations. Nous avons ainsi dans la matrice de bornes initiale deux éléments $(\leq; 0)$ pour dénoter l'égalité entre pr_1 et pr_2 (Site 1) et deux autres éléments $(<; 0)$ pour dénoter les inégalités $pr_1 < pr_3$ et $pr_2 < pr_3$, pr_3 étant associé au site 2.

Algorithme de Ressel L'algorithme de Ressel de la figure 4.4 comprend de grandes similarités avec l'algorithme d'Ellis. En effet, la seule différence entre les deux algorithmes réside au niveau du traitement de la situation de deux insertions concurrentes (Ins ;Ins). Mais, vu que nous considérons à ce niveau simplement la matrice de bornes initiale pour chaque algorithme, nous supposons que la matrice de Ressel est identique à celle d'Ellis de la figure 4.22.

Algorithme de Sun Dans l'algorithme de Sun de la figure 4.5, nous remarquons que les seules variables pertinentes à la détermination des opérations transformées sont les positions courantes p. D'où la conclusion que la matrice de bornes initiale de

Sun illustrée dans la figure 4.23 est une sous-matrice de celles d'Ellis et de Ressel.

	p_1	p_2	p_3	o_1	o_2	o_3	p'_1	p'_2	p'_3
p_1	(≤;0)						(≤;0)		
p_2		(≤;0)						(≤;0)	
p_3			(≤;0)						(≤;0)
o_1				(≤;0)					
o_2					(≤;0)				
o_3						(≤;0)			
p'_1	(≤;0)						(≤;0)		
p'_2		(≤;0)						(≤;0)	
p'_3			(≤;0)						(≤;0)

FIGURE 4.23 Matrice de bornes initiale correspondant à l'algorithme de Sun

Algorithme de Suleiman Dans l'algorithme de Suleiman de la figure 4.6, nous remarquons qu'en plus des positions courantes p et des caractères manipulés c, il y a comparaison des ensembles av et ap de chaque paire d'opérations.

En effet, dans le cas de Suleiman, chaque opération o_1 est construite en définissant en plus des attributs ordinaires deux ensembles av_1 et ap_1 comme suit :

$$av_1 = \{Del(p_i)|p_i < p_1\}$$

$$ap_1 = \{Del(p_i)|p_1 \leq p_i\}$$

De manière similaire, la construction de chaque opération o_2 définit deux ensembles av_2 et ap_2 comme suit :

	p_1	p_2	p_3	c_1	c_2	c_3	o_1	o_2	o_3	av_1	ap_1	av_2	ap_2	av_3	ap_3	p'_1	p'_2	p'_3
p_1	(≤;0)															(≤;0)		
p_2		(≤;0)															(≤;0)	
p_3			(≤;0)															(≤;0)
c_1				(≤;0)														
c_2					(≤;0)													
c_3						(≤;0)												
o_1							(≤;0)											
o_2								(≤;0)										
o_3									(≤;0)									
av_1										(≤;0)								
ap_1											(≤;0)							
av_2												(≤;0)						
ap_2													(≤;0)					
av_3														(≤;0)				
ap_3															(≤;0)			
p'_1	(≤;0)															(≤;0)		
p'_2		(≤;0)															(≤;0)	
p'_3			(≤;0)															(≤;0)

FIGURE 4.24 Matrice de bornes initiale correspondant à l'algorithme de Suleiman

$$av_2 = \{Del(p_j)|p_j < p_2\}$$

$$ap_2 = \{Del(p_j)|p_2 \leq p_j\}$$

Il est à noter que ces ensembles sont remplis dans la situation (o_1 =Ins ;o_2 =Del) entre deux opérations d'après l'algorithme de Suleiman de la façon suivante :

- Si $p_1 \leq p_2$ alors l'élément p_2 est rajouté abstraitement dans ap_1 en ajoutant les deux contraintes dans la matrice : $p_2 \leq ap_1$ et $ap_1 \leq p_2$.
- Si $p_2 < p_1$ alors l'élément p_2 est rajouté abstraitement dans av_1 en ajoutant les

deux contraintes dans la matrice : $p_2 \leq av_1$ et $av_1 \leq p_2$.

Par conséquent, vérifier une intersection non nulle entre les deux ensembles av_1 et ap_2 lors d'une situation possible de deux insertions concurrentes revient à vérifier s'il existe un k satisfaisant les deux contraintes $p_k < p_1$ et $p_2 \leq p_k$ dans la matrice de bornes.

Nous illustrons la matrice initiale de bornes correspondant à l'algorithme de Suleiman dans la figure 4.24.

	p_1	p_2	p_3	c_1	c_2	c_3	o_1	o_2	o_3	p'_1	p'_2	p'_3
p_1	(≤;0)						(≤;0)			(≤;0)		
p_2		(≤;0)						(≤;0)			(≤;0)	
p_3			(≤;0)						(≤;0)			(≤;0)
c_1				(≤;0)								
c_2					(≤;0)							
c_3						(≤;0)						
o_1	(≤;0)						(≤;0)					
o_2		(≤;0)						(≤;0)				
o_3			(≤;0)						(≤;0)			
p'_1	(≤;0)									(≤;0)		
p'_2		(≤;0)									(≤;0)	
p'_3			(≤;0)									(≤;0)

FIGURE 4.25 Matrice de bornes initiale correspondant à l'algorithme d'Imine

Algorithme d'Imine Finalement, l'algorithme d'Imine de la figure 4.7 comprend trois variables pertinentes dans la comparaison des opérations précédant les transformées : Les positions courantes p, les caractères manipulés c et les positions

originales o. Ceci résulte en une matrice de bornes initiale correspondant à l'algorithme d'Imine dans la figure 4.25.

Une fois les matrices de bornes initiales définies pour chaque algorithme IT, nous procédons dans l'étape suivante à une discussion de l'intégration générique des transformées opérationnelles sous forme de contraintes dans les matrices de bornes.

Transformées opérationnelles OT abstraites Dans l'état initial de notre système, une seule matrice de bornes—la matrice de bornes initiale préalablement mentionnée—est allouée à chaque site. Dans notre exemple, on aura 2 matrices au total vu qu'on a $NbSites = 2$ et ce, pour chaque algorithme IT considéré.

Cependant, vu que le comportement des sites est identique, nous développons la procédure de transformées opérationnelles dans le modèle abstrait pour un seul site quelconque comme illustrée dans la figure 4.26.

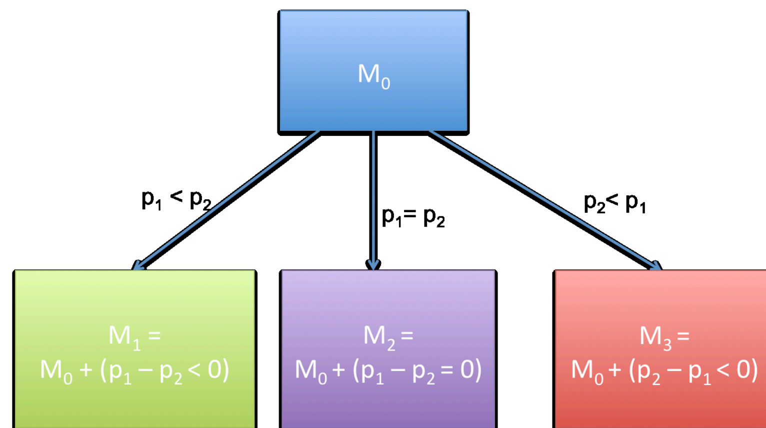

FIGURE 4.26 Procédure générale des transformées opérationnelles à partir de M_0

A partir d'une matrice initiale M_0, nous passons, lors d'une transformation OT d'une opération o_1 après une opération concurrente o_2, à 3 autres matrices subséquentes M_1, M_2 et M_3 et ce, en ajoutant les contraintes considérées respectivement dans chaque cas : $(p_1 - p_2 < 0)$, $(p_1 - p_2 = 0)$ et $(p_2 - p_1 < 0)$. A noter qu'ici on ne considère pas l'un des trois cas mais la combinaison des trois cas résultant en trois

matrices différentes.

Pour illustrer les modifications découlant de la mise à jour des contraintes au niveau de la matrice de bornes initiale M_0, nous prenons un exemple décrivant l'abstraction dans les transformées opérationnelles pour l'algorithme de Sun dans la situation (o_1 =Ins ;o_2 =Ins). Nous distinguons dans cette situation les 3 cas suivants :

1. Si ($p_1 < p_2$) alors ($p'_1 = p_1$)
2. Si ($p_1 = p_2$) alors ($p'_1 = p_1 + 1$)
3. Si ($p_2 < p_1$) alors ($p'_1 = p_1 + 1$)

La matrice initiale de bornes M_0 dans notre exemple est la matrice illustrée dans la figure 4.23.

Dans le premier cas, nous ajoutons une seule contrainte ($p_1 - p_2 < 0$) à M_0 vu que, par défaut, nous avons ($p'_1 = p_1$). Ceci résulte en M_1 illustrée dans la figure 4.27.

	p_1	p_2	p_3	o_1	o_2	o_3	p'_1	p'_2	p'_3
p_1	(≤;0)	**(<;0)**					**(≤;0)**		
p_2		(≤;0)						(≤;0)	
p_3			(≤;0)						(≤;0)
o_1				(≤;0)					
o_2					(≤;0)				
o_3						(≤;0)			
p'_1	**(≤;0)**						(≤;0)		
p'_2		(≤;0)						(≤;0)	
p'_3			(≤;0)						(≤;0)

FIGURE 4.27 Matrice transformée M_1 correspondant au cas ($p_1 < p_2$)

Dans le second cas, nous ajoutons les trois contraintes à M_0 : $(p_1 - p_2 = 0)$ d'une part, $(p'_1 - p_1 \leq 1)$ et $(p_1 - p'_1 \leq -1)$ d'autre part reflétant l'incrémentation de la position p_1 de o_1 sous forme de contraintes. Il faut noter ici que la décrémentation est traitée de la même façon mais en inversant les signes.

Toutefois, vu que la matrice de bornes manipule des domaines, une incrémentation doit se propager dans ce domaine. De ce fait, il ne suffit pas d'incrémenter l'élément correspondant $(\text{p'}_1; p_1)$ mais toute la ligne correspondant à p'_1. De même, la décrémentation de l'élément $(\text{p}_1; p'_1)$ n'est pas suffisante. Nous décrémentons toute la colonne correspondant à p'_1.

La combinaison de toutes ces contraintes, dans M_0, aboutit à la matrice M_2 illustrée dans la figure 4.28. Nous remarquons qu'ici le reste de la ligne et de la colonne correspondant à p'_1 n'est pas affecté vu la dominance de $+\infty$ dans le calcul.

	p_1	p_2	p_3	o_1	o_2	o_3	p'_1	p'_2	p'_3
p_1	(≤;0)	**(≤;0)**					**(≤;-1)**		
p_2	**(≤;0)**	(≤;0)						(≤;0)	
p_3			(≤;0)						(≤;0)
o_1				(≤;0)					
o_2					(≤;0)				
o_3						(≤;0)			
p'_1	**(≤;1)**						**(≤;0)**		
p'_2		(≤;0)						(≤;0)	
p'_3			(≤;0)						(≤;0)

FIGURE 4.28 Matrice transformée M_2 correspondant au cas $(p_1 = p_2)$

Vu que le troisième cas aboutit au même résultat d'incrémentation de la position p'_1, les trois contraintes de base ajoutées à M_0 dans ce cas sont : $(p_2 - p_1 < 0)$, $(p'_1 - p_1 \leq 1)$ et $(p_1 - p'_1 \leq -1)$. Ceci aboutit à la matrice M_3 illustrée dans la figure 4.29.

	p_1	p_2	p_3	o_1	o_2	o_3	p'_1	p'_2	p'_3
p_1	(≤;0)						**(≤;-1)**		
p_2	**(<;0)**	(≤;0)						(≤;0)	
p_3			(≤;0)						(≤;0)
o_1				(≤;0)					
o_2					(≤;0)				
o_3						(≤;0)			
p'_1	**(≤;1)**						**(≤;0)**		
p'_2		(≤;0)						(≤;0)	
p'_3			(≤;0)						(≤;0)

FIGURE 4.29 Matrice transformée M_3 correspondant au cas $(p_2 < p_1)$

Nous continuons de la même façon les transformées opérationnelles entre les opérations concurrentes restantes à partir de chacune des trois matrices M_1, M_2 et M_3. Cette démarche par étapes ressemble à une *exécution symbolique à la volée*.

En se basant sur cette approche tout au long des transformées nécessaires dans le système, nous collectons ainsi un ensemble de matrices résultantes. Nous rappelons que cet ensemble correspond à chaque site considéré. Afin de ne pas collecter des duplicats de matrices dans ces ensembles, nous utilisons les "*LinkedHashSet*" comme structures de données en Java. Dans notre exemple, nous en aurons deux pour les deux sites considérés : E_1 pour le site 1 et E_2 pour le site 2.

Il reste à noter que, dans cette étape, seules les opérations concurrentes sont transformées suivant l'algorithme IT considéré. Deux opérations dépendantes consécutives ne déclenchent ainsi aucun changement au niveau de la matrice de bornes. De plus, la relation de causalité entre ces opérations est respectée en exécutant les opérations d'un même site dans le même ordre de leur construction : o_1 sera construite avant o_2 dans le site 1 si l'on désire qu'elle soit exécutée avant cette dernière.

Exécution abstraite des opérations Les matrices résultantes dans chaque ensemble E_1 et E_2 de l'étape précédente ne sont pas nécessairement des matrices finales

pour chaque site. En effet, en raison de l'absence de la notion de texte dans le modèle abstrait, il s'avère nécessaire de *simuler* l'édition au niveau du texte via des modifications *probables* dans les positions des opérations exécutées.

Plus spécifiquement, ces modifications s'appliquent dans les conditions suivantes :

$\forall o_k$, $\forall o_l$ Si o_l est exécutée après o_k et $p_l \leq p_k$, alors :

- Si ($o_l = Ins$), alors $p'_k = p_k + 1$
- Si ($o_l = Del\ et\ o_k = Ins$), alors $p'_k = p_k$
- Si ($o_l = Del\ et\ o_k = Del$), alors :
  - Si ($p_l < p_k$), alors $p'_k = p_k$
  - Si ($p_l = p_k$), alors $p'_l = p_l + 1$

Nous remarquons que la condition principale pour toute modification probable au niveau des positions concerne toute opération s'exécutant *après* une autre, qu'elle lui soit concurrente ou dépendante, et dont la position d'édition courante lui est inférieure ou égale.

De ce fait, il s'est avéré nécessaire de collecter, dans l'étape précédente, l'ensemble des opérations tout au long de leur exécution dans un vecteur d'opérations spécifique—*ExecutedOps*. Ainsi, l'ordre d'apparition des opérations dans ce vecteur reflètera leur ordre d'exécution. En outre, ce vecteur sera associé à chaque matrice de E_1 et E_2.

Si les modifications s'appliquent au niveau d'une matrice donnée de E_1 par exemple, il y aura mise à jour des nouvelles contraintes au niveau des positions de manière similaire à la phase de transformées opérationnelles de l'étape précédente. Ces matrices modifiées seront replacées dans leur ensemble original, E_1 dans ce cas. Nous pouvons ainsi dire que les ensembles E_1 et E_2 collectent les matrices finales des sites respectifs *Site 1* et *Site 2*.

A ce niveau, toutes les abstractions touchant au modèle ont été discutées, notamment les abstractions dans la représentation des comportements des sites à l'aide des

DBMs, la manipulation de ces matrices pour exécuter abstraitement les transformées opérationnelles et finalement l'exécution abstraite de l'édition. Ces abstractions nous ont permis de manipuler des domaines infinis concernant les positions et les caractères des opérations auparavant restreints à des domaines limités.

Nous discutons dans l'étape prochaine d'un autre avantage de notre modèle abstrait se rapportant à l'ordre d'exécution des opérations.

Ordre générique d'exécutions

Dans le modèle concret, l'ordre d'exécution des opérations dans tout site était limité à l'exécution des opérations locales en premier et ensuite des opérations non locales.

Dans le modèle abstrait, tout ordre d'exécution des opérations respectant le principe de causalité entre les opérations est considéré. Ainsi, en prenant le même exemple envisagé précédemment avec deux opérations dépendantes o_1 puis o_2 dans le site 1, et o_3 concurrente à o_1 dans le site 2, nous aurons les deux ordres d'exécution possibles suivants pour le site 1 :

1. o_3 est exécutée entre o_1 et o_2 dans la séquence $[o_1 ; o_3 ; o_2]$. Dans cet ordre, la transformation IT=(o_3,o_1) est réalisée résultant en o'_3.
2. o_3 est exécutée après o_1 et o_2 dans la séquence $[o_1 ; o_2 ; o_3]$. Dans cet ordre, la transformation IT=$(o_3,[o_1; o_2)]$ est réalisée résultant en o'_3.

De même, nous présentons le seul ordre d'exécution possible pour le site 2 :

1. o_3 est exécutée en premier suivie de o_1 puis o_2 dans la séquence $[o_3 ; o_1 ; o_2]$. Dans cet ordre, la première transformation de l'opération o_1 résulte en une opération o'_1=IT(o_1,o_3). Ensuite, l'opération o_2 sera transformée sur la séquence réordonnée $[o_1 ; o'_3]$ résultant en o'_2. Ce réordonnement est identique à celui rencontré dans le modèle concret et est illustré dans la figure 4.30.

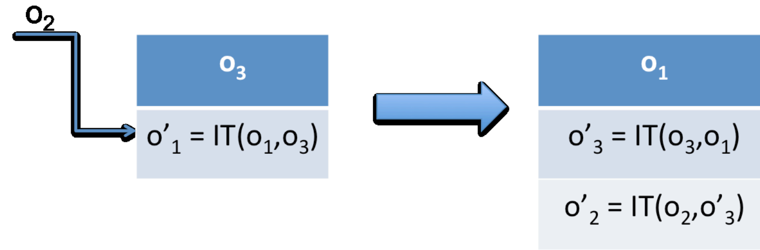

FIGURE 4.30 Réordonnement des opérations pour leur transformation dans *Site 2*

Les ordres d'exécution ainsi présentés seront considérés pour chaque site résultant en une diversité de matrices de bornes dans chaque ensemble associé à un site paticulier.

Ces matrices de bornes ainsi obtenues pour E_1 et pour E_2 sont désormais prêtes à la phase de vérification élaborée dans la partie suivante.

4.3.2 Vérification de la consistance du modèle

Définition de la convergence du modèle abstrait

La vérification de la consistance du modèle manipulant des DBMs revient à comparer chaque paire de matrices finales dans chaque site tout en appliquant les mêmes contraintes de données. Cette démarche remplace la technique de comparaison des textes finaux dans le modèle concret.

Nous optons ainsi pour la comparaison des matrices, résultant de l'application de contraintes de données identiques, pour toute matrice M_1 du *Site 1* ayant la sous-matrice I_1 comme matrice de données et pour toute matrice M_2 du *Site 2* ayant la sous-matrice I_2 comme matrice de données de la façon suivante :

$$(M_1 \wedge I_2) = (M_2 \wedge I_1) \tag{4.3}$$

Par conséquent, la convergence du système est satisfaite si la formule 4.3 est sa-

tisfaite pour toute paire (M_1,M_2) des deux sites respectifs *Site 1* et *Site 2*.

Les intersections '$\wedge$' entre ces matrices résultent de gauche à droite en deux nouvelles matrices M'_1 et M'_2. La comparaison d'égalité entre ces deux matrices résultantes nécessitent une adoption d'une forme canonique associée aux matrices.

La *forme canonique* de toute matrice de bornes se base sur l'algorithme de Floyd-Warshall dont le pseudo-code est illustré dans la figure 4.31. Cet algorithme est un algorithme d'analyse de graphes pour trouver les plus courts chemins dans un graphe dirigé avec des poids sur les arêtes. Pour plus de détails concernant l'algorithme de Floyd-Warshall, se référer à [40].

Il est à noter que toute matrice retournant une matrice nulle après sa transformation en forme canonique est ignorée indiquant une inconsistance dans le domaine représenté.

```
Pour k=1 à n
    Pour i=1 à n
        Pour j=1 à n
        Si M[i][k] + M[k][j] < M[i][j]
            M[i][j] = M[i][k] + M[k][j]
    Si M[i][i] < (<=;0)
        return null
```

FIGURE 4.31 Pseudo-code associé à l'algorithme de Floyd-Warshall

Il est important de mentionner aussi que toutes les matrices obtenues dans le paragraphe 4.3.1 ne sont pas illustrées sous leur forme canonique pour simplifier l'explication.

Après avoir défini la forme canonique, nous pouvons dire qu'avant toute intersection ou comparaison, chaque matrice doit être représentée sous sa forme canonique. En se basant sur ce principe, si deux matrices M_1 et M_2 ont la même forme canonique, nous pourrons considérer ces dernières comme matrices *identiques*, $M_1 \equiv M_2$ ignorant ainsi l'une d'entre elles.

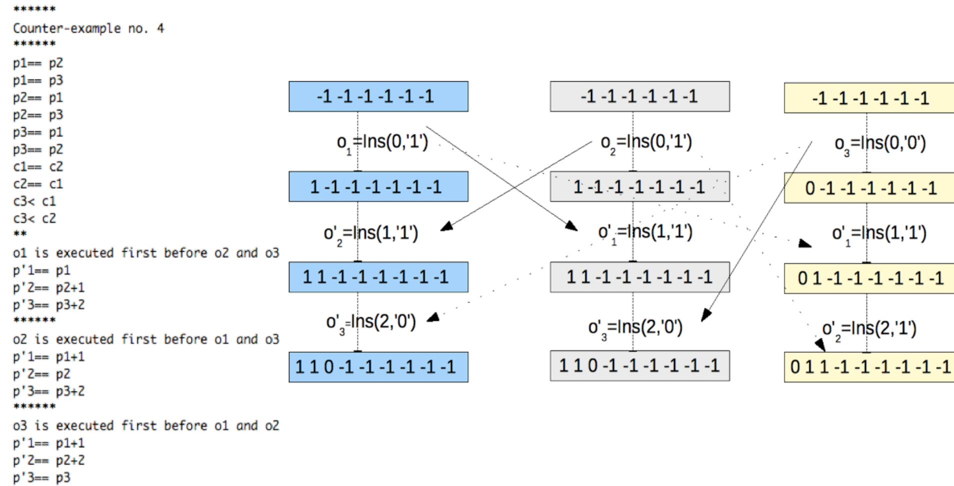

FIGURE 4.32 Contre-exemple abstrait avec l'algorithme de Sun *et al.* pour *3 Ins*

Vérification de la convergence

La notion de convergence étant définie, nous présentons dans la suite un contre-exemple abstrait en appliquant notre approche symbolique abstraite lors de la vérification de la convergence.

Un contre-exemple abstrait affichera le suivant :

- Le numéro du contre-exemple
- L'ordre d'exécution des opérations
- Les contraintes entre les positions originales pour lesquelles nous avons obtenu les contre-exemples
- Les contraintes entre les caractères (si elles existent)
- Les positions transformées en fonction des positions originales

A noter que chaque contre-exemple est affiché par ordre des sites considérés dans la vérification pour y démontrer la divergence.

De plus, ce contre-exemple abstrait englobera un ensemble de contre-exemples concrets. A des fins d'illustration, nous affichons le contre-exemple abstrait obtenu avec l'algorithme de Sun pour 3 sites et 3 opérations 'Ins'. Du côté droit de la figure 4.32, nous considérons une concrétisation possible du contre-exemple illustré.

Nous pouvons alors conclure que cette approche appuie les résultats obtenus avec l'approche de model-checking symbolique tout en nous permettant de gagner à la fois en temps d'exécution et en mémoire relativement au modèle concret antérieur.

Chapitre 5

CONCLUSION

La vérification formelle est une étape cruciale dans le développement de tout système logiciel. Vérifier formellement une vaste classe de propriétés attendues du système contribue à augmenter à la fois la confiance des concepteurs et des utilisateurs vis-à-vis du produit final. Plus spécifiquement, depuis plusieurs années, le model-checking symbolique a pris de l'ampleur dans la vérification de systèmes relativement larges auparavant impossibles à prouver. Dans ce dernier chapitre, nous présenterons d'abord une synthèse de l'application de cette technique sur les deux études de cas envisagés. Les points importants du projet seront ainsi résumés. Par la suite, nous préciserons certaines limitations observées. Finalement, nous conclurons par une discussion qui portera sur les avenues futures de notre démarche.

5.1 Synthèse des travaux

La contribution de cette maîtrise s'articule essentiellement autour de l'application du model-checking jumelé à l'exécution symbolique dans le cadre de systèmes relativement complexes. La vérification de tels systèmes à zone d'états étendue ou même infinie représente un véritable défi. Par conséquent, des abstractions dans le modèle décrivant le système à vérifier s'avèrent indispensables. Nous exploitons dans notre travail ces abstractions au sein du model-checking symbolique de systèmes complexes dans deux champs d'application catégorisant cet ouvrage en deux étapes distinctes.

La première étape de cet ouvrage a été consacrée à la vérification de notre première étude de cas : Le *simulateur de vol*.

Pour ce faire, nous avons modélisé le système en question sous un format spécifique de tables de décision. Cette notation tabulaire est avantageuse dans la représentation d'un tel système à aspect numérique dominé par un ensemble de conditions et d'ac-

tions résultantes.

Après modélisation du système, nous avons procédé à la vérification de ses cas critiques, en d'autres termes les cas où ses variables de sortie sont indéterminées. Toutefois, l'une des difficultés inhérentes à la vérification de ce système émergeait de la qualité de calcul qui s'appuie sur des opérations arithmétiques et trigonométriques arbitraires et complexes. Les domaines infinis des variables réelles dans ces calculs déclenchaient une infinité d'états possibles lors de l'exploration des chemins d'exécution durant le model-checking : C'est le problème d'explosion d'états.

Par ailleurs, adopter des abstractions au niveau de ces calculs ne remédiait nullement à ce problème en raison de la nécessité des valeurs de calcul dans toute exploration ultérieure lors du model-checking. Ceci est dû à l'intégration même des calculs dans les contraintes ou les conditions de chemin.

A ce niveau, afin de contourner ce problème, nous avons proposé de jumeler l'exécution symbolique au model-checking. Ainsi, en imposant les préconditions définies sur les variables d'entrée, nous procédons à une exploration de tous les états possibles par model-checking avec des variables symboliques non initialisées. Cependant, l'intérêt majeur dans cette technique consiste en une possibilité d'explorer uniquement les chemins d'exécution satisfaisant un critère de recherche prédéfini, dans notre cas, un contexte critique du système. Ainsi, lors de l'exploration, si un chemin d'exécution ne répond pas à ce critère, le model-checker considéré rebrousse chemin vers le dernier état le satisfaisant.

Basés sur ce principe, nous avons pu atténuer le problème d'explosion d'états tout en générant des contre-exemples mettant en relief les valeurs des variables d'entrée résultant en une situation critique du système.

Dans la seconde étape de notre travail, nous avons appliqué la même technique d'exécution symbolique jumelée au model-checking dans le cadre d'un *système d'éditions collaboratives distribuées*.

Plus particulièrement, nous nous sommes intéressés à vérifier la consistance de ce système dominé par des opérations d'éditions concurrentes lui octroyant un caractère imprévisible. Ces opérations sont générées puis diffusées sur les différentes copies d'un objet partagé, associées aux utilisateurs du système en réseau. En effet, afin de respecter la convergence entre ces copies, des transformées d'opérations doivent être

appliquées dans des conditions spéciales. Ces transformées ont été proposées dans la littérature et ont été implémentées dans des algorithmes spécifiques.

Par ailleurs, nous avons vu l'intérêt de la vérification par model-checking symbolique de ces algorithmes qui prétendent satisfaire la convergence de ces systèmes. Cependant, la difficulté dans ce type de systèmes réside dans leur caractère interactif augmentant considérablement l'espace d'états dans le model-checking.

En recourant à la même technique envisagée dans la première étude de cas, nous avons pu obtenir un ensemble de contre-exemples mettant l'emphase sur les scénarios de divergence en termes d'opérations sur chaque site. De plus, une dernière étape d'abstractions dans le modèle décrivant le système nous a permis de générer une classe plus étendue de contre-exemples auparavant non trouvés.

Notre approche adoptée reflète son importance non seulement en contredisant les suppositions de convergence des algorithmes considérés mais également en procurant une possibilité de rectification de ces algorithmes après l'analyse des contre-exemples ainsi générés.

5.2 Limitations de la solution proposée

Dans le présent mémoire, nous avons pu vérifier des systèmes relativement complexes en adoptant le model-checking jumelé à l'exécution symbolique.

Cependant, nous avons remarqué que le problème d'explosion d'états peut persister avec cette technique si l'on n'impose pas des préconditions sur les variables d'entrée du système. En effet, l'exécution symbolique envisagée procède à une couverture exhaustive des conditions de chemins à partir de variables symboliques non initialisées : Pour chaque exécution, une combinaison spécifique de variables concrètes, dépendante de la condition de chemin envisagée, est utilisée. Si cette exécution particulière ne satisfait pas la propriété à vérifier, l'ensemble de ces variables concrètes est stocké faisant éventuellement partie des contre-exemples générés.

Par conséquent, si les variables d'entrée ne sont pas prédéfinies dans des intervalles précis, même s'il s'agit d'intervalles de nombres réels, l'exécution symbolique peut en-

gendrer une infinité d'exécutions concrètes résultant en une impossibilité de procéder dans l'exécution en raison du manque de ressources. D'où la nécessité d'imposer des préconditions aux variables d'entrée dans nos deux études de cas.

De plus, nous ajoutons qu'au terme de notre première étude de cas, nous nous sommes limités à une modélisation du système avec des tables de décision ayant un ordre d'exécution séquentiel. Il existe une hiérarchie à respecter dans les tables : Le résultat d'une table de décision est essentiel afin de poursuivre le calcul d'autres tables de décision. Cette limitation découle de la nature même du simulateur de vol à vérifier.

Enfin, concernant la seconde étude de cas, nous nous sommes restreints à la vérification des cinq algorithmes les plus évoqués dans la littérature.

5.3 Avenues futures

Les limitations relatives aux deux études de cas peuvent être des indications pour des recherches futures. En effet, il serait intéressant de vérifier des simulateurs de vol modélisés par des tables de décision concurrentes. A la différence des tables de décision considérées dans notre travail, ces tables comprendront des structures de contrôle [41] comme "Call", "goto", "Repeat", où l'ordre d'exécution dans les tables n'est pas nécessairement séquentiel. Ces tables pourront susciter de plus amples abstractions au niveau du model-checking symbolique.

De plus, dans le cadre des systèmes d'éditions collaboratives distribuées, il serait pertinent d'analyser les contre-exemples que nous avons générés pour chacun des algorithmes dans le but d'essayer de rectifier les erreurs dans chacun d'entre eux.

Il serait peut-être aussi possible de s'appuyer sur cette analyse pour la conception d'un algorithme satisfaisant la convergence de ces systèmes. La conception de tels algorithmes demeure une tâche difficile en raison de l'infinité d'états possibles dans ces systèmes à caractère imprévisible.

Références

[1] Havelund, K., Skou, A., Larsen, K. and Lund, K. : Model Checking Java Programs using Structural Heuristics, *In Proc. of the 18th IEEE Real-Time Systems Symposium*, 1997.

[2] Anderson, R., Beame, P., Burns, S., Chan, W., Modugno, F., Notkin, D. and Reese, J. : Model Checking Large Software Specifications, *IEEE Transactions on Software Engineering*, 24, p. 156, 1996.

[3] Heitmeyer, C., Jeffords, R. and Labaw, B. : Automated Consistency Checking of Requirements Specifications, *ACM Transactions on Software Engineering and Methodology*, 5, p. 231, 1996.

[4] Rakkay, H. : Approches formelles pour la modélisation et la vérification du contrôle d'accès et des contraintes temporelles dans les systèmes d'information, Ph.D. thesis, École Polytechnique de Montréal, 2009.

[5] Huth, M. and Ryan, M. : Logic in Computer Science - Modelling and Reasoning about Systems, 2nd edition, Cambridge, 2004.

[6] Mufti, A. and Tcherukine, C. : Integration of Model-Checking Tools : from Discrete to Hybrid Models, *Multitopic Conference, 2007. INMIC 2007. IEEE International*, 5, p. 1, 2007.

[7] Bharadwaj, R. and Heitmeyer, C. : Model Checking Complete Requirements Specifications Using Abstraction, *Automated Software Engineering*, 6, 1997.

[8] Clarke, E. M., Khaira, M. and Zhao, X. : Word level model checking-avoiding the Pentium FDIV error, *Design Automation Conference Proceedings 1996, 33rd*, p. 645-648, 1996.

[9] Bryant, R., Bryant, A. and Chen,Y. : Verification of Arithmetic Circuits with Binary Moment Diagrams, *In Proceedings of the 32nd ACM/IEEE Design Automation Conference*, p. 535-541, 1995.

[10] Richard, W., Anderson, R., Beame, P. and Notkin, D. : Combining Constraint Solving and Symbolic Model Checking for a Class of Systems with Non-linear Constraints, *In Computer Aided Verification*, p. 316-327, 1997.

[11] Rushby, J. : Ubiquitous abstraction : a new approach for mechanized formal verification, *Formal Engineering Methods, 1998. Proceedings. Second International Conference on*, p. 176-178, 1998.

[12] Biere, A., Cimatti, A., Clarke, E. and Zhu, Y. : Symbolic Model Checking without BDDs, 1999.

[13] Ammann, P. and Black, P. E. : Abstracting formal specifications to generate software tests via model checking, *Digital Avionics Systems Conference, 1999. Proceedings. 18th*, 2, 10.A.6-1-10.A.6-10 vol.2, 1999.

[14] Callahan, J., Schneider, F. and Easterbrook, S. : Automated Software Testing Using Model-Checking, 1996.

[15] King, J. C. : Symbolic execution and program testing, *Commun. ACM*, 19, p. 385-394, 1976.

[16] Khurshid, S., Păsăreanu, C. S. and Visser, W. : Generalized Symbolic Execution for Model Checking and Testing, *In Proceedings of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, p. 553-568, 2003.

[17] Anand, S., Păsăreanu, C. S. and Visser,W. : Symbolic execution with abstraction, *International Journal on Software Tools for Technology Transfer (STTT)*,

p. 53-67, 2008.

[18] Groce, A. and Visser, W. : Model Checking Java Programs using Structural Heuristics, *in International Symposium on Software Testing and Analysis*, 2002.

[19] Giunchiglia, C. C., Cimatti, A., Clarke, E., Giunchiglia, F. and Roveri, M. : NUSMV : a new Symbolic Model Verifier, Springer, p. 495-499, 1999.

[20] Jackson, D. : Software Abstractions : Logic, Language, and Analysis, The MIT Press, 2006.

[21] Yessenov, K. : Report on JForge Specification Language and Tools, MIT Software Design Group, 2009.

[22] Mehlitz, P. C., Visser, W. and Penix, J. : The JPF Runtime Verification System, 2008.

[23] Penix, J., Visser, W., Park, S., Engstrom, E., Larson, A. and Weininger, N. : Verifying Time Partitioning in the DEOS Scheduling Kernel, *In 22nd International Conference on Software Engineering (ICSE00)*, IEEE Press, 2004.

[24] Mehlitz, P. C. : Trust Your Model - Verifying Aerospace System Models with Java Pathfinder, *Aerospace Conference, 2008 IEEE*, p. 1-11, 2008.

[25] Anand, S., Păsăreanu, C. and Visser, W. : JPF–SE : A Symbolic Execution Extension to Java PathFinder, *TACAS*, 4424, p. 134-138, 2007.

[26] Kahkonen, K. : Evaluation of Java PathFinder Symbolic Execution Extension, 2007.

[27] Păsăreanu, C. S., Mehlitz, P. C., Bushnell, D. H. and Gundy-Burlet, K., Lowry, M., Person, S. and Pape, M. : Combining unit-level symbolic execution and

system-level concrete execution for testing nasa software, *ISSTA '08 : Proceedings of the 2008 international symposium on Software testing and analysis*, p. 15-26, 2008.

[28] Visser, W., Păsăreanu, C. S. and Khurshid, S. : Test Input Generation with Java PathFinder.

[29] Visser, W., Păsăreanu, C. S. and Pelánek, R. : Test input generation for red-black trees using abstraction, *ASE '05 : Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, p. 414-417, 2005.

[30] Visser, W., Păsăreanu, C. S. and Pelánek, R. : Test input generation for java containers using state matching, *ISSTA '06 : Proceedings of the 2006 international symposium on Software testing and analysis*, p. 37-48, 2006.

[31] Heitmeyer, C. : Managing Complexity in Software Development with Formally Based Tools, 2004.

[32] Heitmeyer, C. L., Jeffords, R. D. and Labaw, B. G. : Automated Consistency Checking of Requirements Specifications, *ACM Transactions on Software Engineering and Methodology*, 5, p. 231-261, 1996.

[33] Holzmann, G. J. : The Model Checker SPIN, *Software Engineering*, 23, p. 279-295, 1997.

[34] Boucheneb, H. and Imine, A. : On Model-Checking Optimistic Replication Algorithms, *Formal Techniques for Distributed Systems*, 5522, p. 73-89, 2009.

[35] Imine, A., Molli, P., Oster, G. and Rusinowitch, M. : Proving Correctness of Transformation Functions in Real-Time Groupware, *In Proceedings of the 8th European Conference on Computer-Supported Cooperative Work*, 2003.

[36] Imine, A., Rusinowitch, M., Oster, G. and Molli, P. : Formal design and verification of operational transformation algorithms for copies convergence, *Theor. Comput. Sci.*, 351, p. 167-183, 2006.

[37] Saito, Y. : Consistency Management in Optimistic Replication Algorithms, 2001.

[38] Quiroga, L. and Fernandez, A. : A P2P Groupware Framework based on Operational Transformations, *ICDCSW '07 : Proceedings of the 27th International Conference on Distributed Computing Systems Workshops*, p. 70, 2007.

[39] Dill, D. L. : Timing assumptions and verification of finite-state concurrent systems, *Proceedings of the international workshop on Automatic verification methods for finite state systems*, p. 197-212, 1990.

[40] Srinivasan, T., Gangadharan, B. R., Gangadharan, S. A. and Hayawardh, V. : A scalable parallelization of all-pairs shortest path algorithm for a high performance cluster environment, *Parallel and Distributed Systems, 2007 International Conference on*, 2, p. 1-8, 2007.

[41] Lover, R. : Elementary Logic For Software Development, Springer, 2008.

[42] http ://alloy.mit.edu/tutorial3/alloy-tutorial.html

[43] http ://sdg.csail.mit.edu/forge

[44] http ://javapathfinder.sourceforge.net/