THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut Supérieur de l'Aéronautique et de l'Espace (ISAE)

Présentée et soutenue par :
Antoine FERLIN

le mardi 3 septembre 2013

Titre :

Vérification de propriétés
temporelles sur des logiciels avioniques par analyse dynamique formelle

---

École doctorale et discipline ou spécialité :
ED MITT : Sureté de logiciel et calcul de haute performance

Unité de recherche :
Équipe d'accueil ISAE-ONERA MOIS

Directeur(s) de Thèse :

Mme Virginie WIELS (directrice de thèse)

Jury :

M. Yamine AIT AMEUR : Professeur à l'ENSEEIHT - Président du jury
M. Jacques JULLIAND : Professeur à l'Université de Franche-Comté - Rapporteur
M. Ioannis PARISSIS : Professeur à l'INP Grenoble - Rapporteur
M. Philippe DHAUSSY : Enseignant-Chercheur à l'ENSTA Bretagne - Examinateur
M. Julien SIGNOLES : Ingénieur de recherche au CEA - Examinateur
Mme Virginie WIELS : Maître de Recherche à l'Onéra - Directrice de Thèse
M. Famantanantsoa RANDIMBIVOLOLONA : Expert Airbus

Remerciements

Je remercie les membres du jury pour leurs remarques et conseils qui m'ont permis d'améliorer ce manuscrit.

Je remercie ma maître de thèse, Madame Virginie Wiels, pour sa disponibilité envers et contre tout, ses multiples conseils, sa bonne humeur. J'ai été très sensible à ses qualités humaines d'écoute et de compréhension tout au long de son encadrement. J'espère que mes talents culinaires, comme elle l'a dit à la soutenance, ne lui manqueront pas trop.

Mes remerciements vont également à mon co-directeur de thèse, Monsieur Famantanantsoa Randimbivololona, pour ses multiples conseils, sa vision de ce qu'est la recherche tout en travaillant dans un milieu industriel, ses blagues le 1er avril. Je n'ai toujours pas trouvé son amulette anti-bug, mais je ne désespère pas et continue d'arrache-pied à la chercher !

J'ai été accueilli par le département du traitement de l'information et modélisation à l'Onéra et par le département Logiciel (EYYW) à Airbus Operation S.A .S. Je remercie donc Messieurs Bernard Lecussan et Jean-Denis Muller Directeurs successifs du DTIM, et Messieurs Daniel Watson, Olivier Devienne et Philippe Herry Directeurs de département successifs et chef de service, pour m'avoir accueilli dans leurs locaux. Mes remerciements vont également aux membres du labo et au groupe de travail à Airbus pour leur aide que ce soit du point de vue technique, administratif, comme pour les bons moments passés ensemble.

Merci à Pierre-Loïc Garoche, Jacques Cazin, Pierre Roux, Rémy Wyss et Guy Durrieu côté Onéra et Jean Souyris, David Delmas, Philippe Lemeur, Abderrahmane Brahmi, et Gilles Trémolières côté Airbus, pour leur soutien sans faille lorsque je me posais des questions sur Linux, Latex, OCaml, les logiciels avioniques, les problèmes administratifs ou autres...

Merci aux thésards du DTIM pour les parties de Coinche, Hanabi, Mots fléchés, ou autre le midi au RU, c'était un réel plaisir, surtout lorsque j'avais besoin de me changer les idées pendant la rédaction ! Je n'oublierai pas leur bonne humeur et leur blague sur ma théinomanie (oui j'aime bien la bergamote, mais je préfère le thé orange cannelle !).

Merci à Alexandre pour m'avoir aidé à décompresser pendant les moments difficiles.

Mes remerciements vont également à mes parents et ma sœur pour m'avoir soutenu jusqu'au bout de mes études.

Je remercie également mes grands parents pour m'avoir donné le goût des études. Merci également à Maurice et Béatrice pour m'avoir accueilli chez eux au milieu de mes études pour un stage qui m'a fait découvrir et apprécier la recherche.

Table des matières

Introduction

Cette thèse s'organise en quatre parties.

Le contexte industriel présenté dans le chapitre 1, est la vérification de logiciels avioniques soumis à des contraintes de certification. Différentes techniques de vérification sont utilisées, mais certaines propriétés restent difficiles à vérifier. C'est le cas de certaines propriétés temporelles. Le chapitre 2 décrit des travaux existants concernant l'analyse dynamique de ce type de propriétés. Le chapitre 3 détaille l'objectif de cette thèse qui est de mettre en œuvre une approche de vérification formelle de propriétés temporelles sur des traces d'exécution de logiciels.

L'approche proposée fait l'objet de la seconde partie. Le chapitre 4 propose un langage de spécification formelle des propriétés temporelles à vérifier, reposant sur une association de LTL et d'expressions régulières. Le chapitre 5 s'intéresse à la génération de la trace d'exécution à partir de la propriété que l'on cherche à vérifier. Le chapitre 6 se consacre, quant à lui, à la vérification de la propriété et propose une approche statistique pour gérer les problèmes de fin de trace et fournir des informations supplémentaires à l'utilisateur.

La troisième partie concerne la mise en pratique de l'approche proposée. Le chapitre 7 détaille les prototypes développés pendant cette thèse. Le chapitre 8 présente des expérimentations sur différents logiciels Airbus.

La quatrième partie conclut les travaux et propose quelques perspectives.

Première partie

Contexte et objectifs

Chapitre 1

Contexte

Sommaire

1.1 Logiciels embarqués et certification

Les systèmes embarqués sont de plus en plus utilisés, que ce soit dans des domaines peu critiques comme les téléphones, les lecteurs CD, ou dans des systèmes dont la sûreté est capitale, comme les voitures, les trains, ou les avions.

Pour des systèmes critiques, un bug logiciel peut avoir des conséquences graves, voire même funestes. Afin de minimiser le risque d'accident, chaque logiciel embarqué critique est soumis à des processus de certification. En avionique, le standard de certification du logiciel est le DO-178 [RTCA 2011].

Ce standard définit cinq niveaux d'assurance, A, B, C, D, et E. Un système de niveau A peut provoquer la perte de l'avion et la mort de tous les passagers s'il tombe en panne. Un système de niveau E n'a pas d'incidence sur le vol s'il tombe en panne.

Pour chaque niveau, des objectifs de développement et de vérification sont définis. Le standard ne prescrit pas les moyens à mettre en œuvre pour chaque objectif. Cependant, toute méthode employée doit être justifiée et les objectifs du DO-178 satisfaits par la méthode doivent être déterminés.

Une des exigences du standard est de décrire le cycle de vie utilisé pour produire le logiciel. Pour l'avionique, le cycle de vie le plus utilisé est le cycle en V, décrit sur la figure 1.1. Il décrit les différentes étapes de développement du logiciel, de la spécification à la validation.

À chaque élément de la branche gauche du cycle en V correspond un élément de la branche droite. La branche gauche, ou branche descendante regroupe les phases de production du logiciel. La branche droite ou branche ascendante regroupe les phases de contrôle du logiciel.

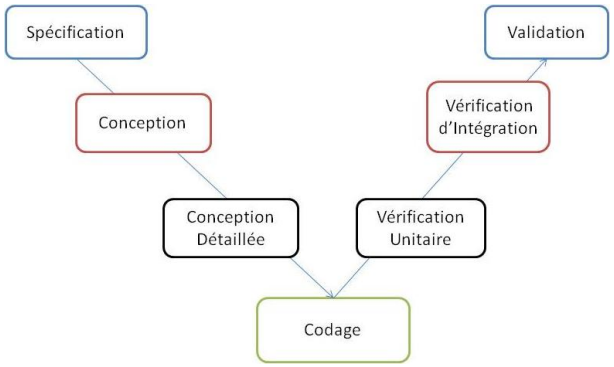

FIGURE 1.1 – Le cycle en V

1.2 Moyens de vérification

L'activité de vérification consiste à s'assurer qu'un programme implémenté est conforme à sa spécification. Elle se décompose en vérification unitaire et vérification d'intégration. La vérification unitaire consiste à vérifier chaque fonction atomique du programme. La vérification d'intégration, quant à elle, vérifie l'enchaînement de plusieurs blocs de programmes, composés de fonctions unitaires.

Les moyens de vérification peuvent être classés suivant deux critères :

- l'analyse est statique ou dynamique,
- l'analyse est formalisée [Ait Ameur 2010] ou non.

1.2.1 Analyse dynamique de programmes

1.2.1.1 Analyse non formelle

Le test est la méthode de vérification la plus répandue. L'activité de test comporte trois étapes successives :

1. définition d'un jeu d'entrées et des résultats attendus,
2. exécution du programme avec le jeu d'entrées défini,
3. comparaison du résultat obtenu avec le résultat attendu appelé oracle, pour s'assurer que le résultat est correct ou non.

Un environnement de test peut être utilisé pour automatiser la vérification du logiciel. Cet environnement est typiquement composé d'un langage qui définit un jeu de tests (jeu des entrées, programme ou partie de programme à vérifier, oracle pour un point donné du programme uniquement). À partir de cette définition, un exécutable de test est généré automatiquement et lié au programme à vérifier, avant d'être lancé. Les résultats sont collectés et comparés automatiquement, de sorte que l'opérateur peut pointer efficacement les cas de tests qui ont échoué.

En avionique, le principal défaut du test est qu'il nécessite des environnements d'exécution lourds, du fait de l'utilisation de plate-formes matérielles dédiées oné-

reuses. La maintenance des logiciels est donc considérablement alourdie par la maintenance conjointe des matériels physiques de tests.

De plus, l'exécution de tests sur un environnement matériel dédié rend cette technique intrusive, ce qui peut engendrer des problèmes pour la certification. Les exécutions sur bancs d'essais diffèrent du matériel embarqué par la fonction d'écoute du comportement du programme testé par les bancs d'essais, et par l'utilisation d'un exécutable permettant de diriger le test.

La simulation consiste à remplacer le matériel et les plate-formes dédiées par un environnement qui simule ce matériel. Un avantage de cette méthode est qu'il n'est pas nécessaire d'instrumenter le code du programme à exécuter, si le simulateur est instrumenté. Il suffit de définir les points d'observation à poser dans le programme, et les variables à collecter pour chaque point d'observation. Le temps d'exécution peut également être simulé et collecté.

1.2.1.2 Analyse formelle

L'analyse dynamique formelle de programme consiste à évaluer des propriétés formalisées sur des exécutions du programme. Il s'agit d'une branche en plein essor ces dernières années, également connue sous le nom de runtime verification. Deux écoles existent ; soit la propriété est vérifiée conjointement à l'exécution, soit elle est vérifiée a postériori sur une trace d'exécution, à savoir une séquence chronologique des différents états du programme. Le chapitre 2 reviendra plus en détail sur ces approches. L'exécution de programme peut se faire dans un environnement de test classique ou dans un environnement qui simule le matériel du logiciel.

1.2.2 Analyse statique de programmes

L'analyse statique est une famille de techniques non intrusives qui ne requièrent pas l'exécution du programme. Lorsque l'analyse est formalisée, elle est basée sur une analyse mathématique, automatique, sûre et exhaustive du programme.

1.2.2.1 Analyse non formelle

Les revues et analyses sont classiquement utilisées en complément du test. D'après le DO-178B, "Les revues et les analyses diffèrent en ce que les analyses fournissent une preuve reproductible de l'exactitude, alors que les revues donnent une évaluation qualitative de cette exactitude." Ces techniques par expertise humaine, procédurées, sont en l'état difficilement automatisables.

1.2.2.2 Analyse formelle

Deux techniques d'analyse statique formelle ont été utilisées pour la certification de logiciels avioniques : l'interprétation abstraite et les méthodes déductives. Le

model checking a également été utilisé sans forcément donner lieu à des crédits de certification.

L'interprétation abstraite [Cousot 1977] offre la possibilité de vérifier des propriétés génériques. Il existe actuellement différents outils permettant chacun de vérifier une classe de propriétés spécifique. Astree [Blanchet 2003],[Souyris 2007] est capable de vérifier l'absence de runtime error sur un code industriel, telle que les divisions par 0, le débordement de tableau, etc... aiT permet de déterminer le WCET (Worst Case Execution Time [Heckmann 2005],[Souyris 2005]), le pire temps d'exécution. L'outil StackAnalyzer [Heckmann 2005] offre la possibilité de vérifier l'absence de débordement de pile en déterminant une borne supérieure de sa taille.

Les méthodes d'interprétation abstraite sont exhaustives, quasiment automatiques et sûres. En outre, elles permettent de vérifier des logiciels de grandes tailles (plusieurs centaines de milliers de lignes de code). Cependant, elles ne permettent de vérifier que certaines propriétés génériques (division par 0, débordement de tableau ...). En outre, puisque l'interprétation abstraite est basée sur une sur-approximation des états possibles d'un programme, des fausses alarmes peuvent être levées et doivent être traitées par les opérateurs, soit en affinant la sur-approximation si c'est possible, soit en utilisant un autre moyen de vérification.

Les méthodes déductives permettent de vérifier des propriétés fonctionnelles spécifiées par l'utilisateur, en logique du premier ordre. Ces méthodes reposent sur le calcul de la plus faible pré-condition [Dijkstra 1976]. Les propriétés peuvent être prouvées automatiquement et dans certains cas, la preuve doit être terminée interactivement par l'utilisateur [Cuoq 2012],[Baudin 2002],[Randimbivololona 1999]. Ces méthodes restent sensibles aux classes de programme. Il existe des utilisations industrielles des preuves unitaires de programme, à travers les outils Caveat et Frama-C[Souyris 2009], pouvant être plus efficaces que les tests unitaires.

Les méthodes déductives sont également exhaustives et sûres dès que la spécification des propriétés et des invariants nécessaires est complète. Cette action de spécification est cependant fastidieuse. En outre, le champ d'application de ces méthodes est restreint. Elles peuvent remplacer les tests unitaires (les fonctions appelées par la fonction vérifiée sont remplacées par des modèles), et dans une moindre mesure les tests d'intégration (les fonctions appelées par la fonction vérifiée sont également spécifiées et vérifiées à l'aide des méthodes déductives). Enfin, la vérification doit être faite par des spécialistes, dans le cas où la preuve d'une propriété doit être interactivement terminée.

Le model checking [Schnoebelen 1999],[Julliand 2009] regroupe un ensemble de méthodes basées sur une modélisation du comportement du programme. Cette modélisation est ensuite transformée dans un formalisme [Miller 2010] (automate) permettant de vérifier des propriétés sur toutes les exécutions du programme. L'étape la plus importante est donc la modélisation, qui doit être en adéquation avec le code

du programme. Des techniques utilisant le slicing de code existent pour construire automatiquement un modèle à partir du code du programme et de la propriété à vérifier [Henzinger 2003],[Corbett 1998]. Le model checking permet de vérifier des propriétés temporelles de façon automatique. Cependant, cette vérification est faite sur un modèle du programme à vérifier.

1.3 Position du problème

Malgré la diversité des méthodes de vérification, certaines propriétés sont trop complexes pour que des méthodes automatiques ou semi automatiques puissent intervenir. C'est notamment le cas des propriétés temporelles, qui sont fréquemment rencontrées en avionique, du fait du caractère temps réel des applications.

En conséquence, seules les analyses et revues peuvent être utilisées pour vérifier ces propriétés complexes, ce qui engendre un coût important (jusqu'à 40% du coût du logiciel) dans la création des logiciels.

Cette thèse va s'intéresser aux techniques d'analyse dynamique formelle qui pourraient être utilisées pour la vérification de ces propriétés.

Chapitre 2

État de l'art

Sommaire

L'objectif de ce chapitre est de donner un aperçu des travaux existants dans le domaine de l'analyse dynamique de propriétés formelles et de positionner mes travaux par rapport à l'état de l'art. Dans un premier temps, plusieurs langages de formalisation de propriétés seront décrits dans 2.1. Dans un deuxième temps, les travaux en Runtime verification seront présentés dans 2.2 en distinguant d'une part l'analyse en ligne, et d'autre part l'analyse a posteriori d'une exécution de programme. Dans un dernier temps, dans la section 2.3, une analyse de la gestion de la fin de l'exécution sera faite pour les travaux étudiés.

2.1 Logique et langages de spécification

Les langages de spécification sont traditionnellement construits au-dessus de la logique. Différents types de logiques sont utiles selon les propriétés qu'il faut formaliser :

- logique propositionnelle et du premier ordre,
- logiques temporelles
  - linéaire (LTL),
  - arborescente (CTL),
  - temporisée (MTL, TLTL).

2.1.1 Langages de spécification

Le langage ACSL [Baudin 2008] (ANSI/ISO C Specification Language) est un langage développé dans le cadre de Frama-C [Cuoq 2012]. Il permet d'exprimer des propriétés sous forme de contrats de fonction avec notamment les pré-requis et les comportements de la fonction. Un comportement est un ensemble de deux types de prédicats, à savoir les prédicats définissant des propriétés sur les entrées de la fonction (pré-condition), et des prédicats sur la sortie attendue de la fonction pour les entrées données (post-condition). Ce langage permet de vérifier des propriétés entre un appel de fonction et le résultat de la fonction. Cependant, ce langage n'est pas adapté pour vérifier une propriété temporelle, qui nécessite de connaître l'état du programme à chaque pas d'exécution du programme.

L'article [d'Amorim 2005b] expose des travaux sur la vérification de propriétés par analyse dynamique en utilisant un ω-langage régulier (langage ajoutant la répétition infinie à un langage régulier). Un langage régulier est transformable en automate. Un ω-langage est, quant à lui, transformable en automate de Büchi. L'approche proposée consiste donc à transformer l'ω-langage en automate de Büchi, puis l'automate de Büchi en BTT-FSM [1]. Ce dernier est alors parcouru pour la vérification de la propriété. L'article précise que la logique temporelle semble plus intuitive et compacte que les ω-langages réguliers. L'expressivité, quant à elle, est équivalente à l'écriture directe d'un automate.

[Barringer 2003] et [Barringer 2004] présentent une approche à la fois en ligne et a posteriori pour vérifier des propriétés temporelles LTL. Utilisé pour vérifier des propriétés sur des logiciels de robots d'exploration planétaire, EAGLE est basé sur l'utilisation de trois primitives temporelles : l'opérateur renvoyant l'état suivant ($\circ$), l'opérateur renvoyant l'état précédent($\odot$) et l'opérateur de concaténation d'événements (séquence d'événement). Tous les autres opérateurs de la logique temporelle classique sont définis à partir de ces opérateurs à l'aide d'équations de point fixe. La section 2.2 reviendra sur ce point. EAGLE nécessite d'appréhender le comportement en fin d'exécution du programme. En effet, chaque opérateur temporel possède deux versions pour gérer la fin de l'exécution. La section 2.3 reviendra sur ce point. Pour chaque opérateur temporel écrit, il faut donc choisir la version adaptée au cas

1. binary transition tree finite state machine.

étudié, ce qui rend la tâche de l'utilisateur ardue. L'expressivité de ce langage est comparable à la logique temporelle linéaire classique.

Le langage Sugar/PSL [Beer 2001], [Foster], [PSL 2004] est un langage de spécification de haut niveau développé pour la conception d'environnements matériels. Sugar a été développé dès 1994 par IBM. Reposant sur la logique temporelle arborescente (CTL), Sugar a été proposé comme standard de spécification en 1998. La seconde version de Sugar est basée sur une version linéaire du temps (LTL), bien que CTL soit toujours intégrée. Sugar a ensuite changé de nom pour devenir PSL (Property Specification Language). PSL intègre un langage d'expressions régulières (SERE) et autorise également l'utilisation de propriétés "répliquées", ou propriété définies avec un opérateur "quel que soit". Par exemple, il est possible d'écrire que "Quel que soit y dans $[\![-5;5]\!]$, toujours, $x+y>0$. L'utilisation de propriétés paramétrées présente un avantage important pour la vérification de propriétés temporelles. En effet, cela permet d'écrire une propriété déclinable pour toutes les valeurs possibles d'une variable donnée, ainsi éviter d'oublier une des valeurs de cette variable.

Le langage SALT [Bauer 2006b] est un langage de spécification basé sur LTL et TLTL (Timed LTL). Il intègre également la notion d'expressions régulières et de propriétés paramétrées. De plus, certains opérateurs temporels sont étendus. Par exemple l'opérateur jusque (until : $\mathcal{U}$), deux versions sont possibles pour la propriété $\phi \,\mathcal{U}\, \psi$:

- ϕ doit être vrai dans l'état où ψ devient vrai, (opérateur $\mathcal{U}_F$)
- ϕ vrai n'est pas requis lorsque ψ devient vrai (interprétation classique de $\mathcal{U}$).

La version étendue des opérateurs n'apporte pas d'expressivité supplémentaire, puisque la formule $\phi \,\mathcal{U}_F\, \psi = (\phi)\,\mathcal{U}\,(\phi \wedge \psi)$ lie les opérateurs $\mathcal{U}_F$ et $\mathcal{U}$.

L'article [Dhaussy 2009] évalue un langage de description de contextes couplé à l'utilisation de patrons de propriétés pré-établis. Les patrons de propriétés permettent d'aider opérateur à écrire efficacement les propriétés les plus fréquemment rencontrées. Ces propriétés ne sont vérifiées que dans le contexte associé. L'avantage de disposer d'un langage de description de contexte est de pouvoir regrouper l'ensemble des propriétés à vérifier pour un élément de programme donné (fonction, ensemble de fonctions ...). En ce qui concerne l'utilisation des patrons de propriétés, l'article conclut qu'environ 70% des propriétés rencontrées peuvent être écrites à l'aide de patrons. L'utilisation de patrons est donc une piste intéressante à étudier pour aider l'utilisateur à la spécification de propriétés.

2.1.2 Positionnement de la thèse

Dans le cadre de cette thèse CIFRE, deux critères sont à prendre en compte pour le choix du langage :

- le pouvoir d'expressivité,
- les éléments syntaxiques permettant de faciliter l'écriture d'une propriété.

Les langages de type SALT et PSL ont un pouvoir d'expressivité permettant de formaliser la majeure partie des propriétés à vérifier. Cependant, ce sont des langages très riches et complexes à appréhender, car ils se veulent les plus larges

possible, pour répondre à tout type de problème rencontré. Par exemple, PSL repose simultanément sur deux visions différentes du temps (linéaire et arborescente), ce qui peut être source d'erreur pour l'utilisateur. Le but de cette thèse CIFRE est de vérifier des propriétés temporelles dans un contexte opérationnel précis, et nécessitant des spécificités que ne fournissent pas SALT et PSL, comme l'utilisation d'opérateurs de comparaison bit à bit, de propriétés numériques ou de propriétés temporelles orientées vers le passé. Ce langage doit permettre à un opérateur de spécifier les propriétés à vérifier sur une exécution d'un programme le plus simplement et efficacement possible. La définition d'un langage restreint aux problèmes rencontrés (confer section 4.1), dérivant de la logique temporelle linéaire a été privilégiée. L'utilisation de patrons pré-définis est également une piste intéressante pour simplifier la spécification de propriétés par un opérateur.

2.2 Runtime verification

La runtime verification consiste à exécuter un programme et à extraire pendant toute la durée de l'exécution des informations permettant de vérifier une propriété donnée. A titre de comparaison, le test ne permet de vérifier une propriété qu'entre le point d'entrée et le point de sortie d'un programme, puisqu'il s'agit de comparer pour un jeu d'entrées donnés la sortie obtenue avec un oracle. Pour vérifier une propriété par runtime verification, il est cependant tout à fait possible d'exécuter le programme à vérifier avec des jeux de tests existants, ou d'utiliser des systèmes de générations de cas de test comme le propose [Blanc 2006].

Les travaux existants peuvent être classés en deux catégories, à savoir vérification en ligne et vérification a posteriori d'une propriété. La vérification en ligne consiste à vérifier la propriété simultanément avec l'exécution du logiciel. Certains auteurs différencient "inline" et "outline". Le premier consiste à fusionner les codes de vérification et du programme, tandis que "outline" consiste à vérifier la propriété avec un processus parallèle communiquant par socket avec le programme vérifié. La distinction ne sera pas effectuée ici. La vérification a posteriori nécessite que lors de l'exécution du logiciel, une trace d'exécution, i.e. une séquence d'états mémoire du programme, soit produite. La propriété est alors vérifiée après l'exécution du logiciel, sur cette trace. Pour la vérification en ligne ou a posteriori, le code du logiciel étudié est instrumenté.

2.2.1 Vérification en ligne

De nombreux travaux existent, sur la vérification en ligne. Les études présentées par [Havelund 2001a] et [Havelund 2002b] présentent une approche de vérification de propriétés temporelles avec l'outil Maude. Ce dernier est exécuté en parallèle du programme. À chaque pas de calcul, la propriété temporelle que l'on cherche à vérifier est réécrite en fonction des événements du programme. Dans l'article, il est également expliqué que la vérification a posteriori n'est pas étudiée car les traces

d'exécution sont trop encombrantes sur le disque dur pour être pratique d'utilisation. La vérification ne s'effectue donc qu'en ligne.

[Stolz 2006] est basé sur le langage Java et propose de vérifier des propriétés LTL par transformation de la formule temporelle en automate fini dit alternant (Alternating Finite Automaton or AFA). Ce sont des automates dont les états sont des formules temporelles. Chaque transition a pour origine un état et pour destination une nouvelle formule temporelle définie à partir des états existants, des opérateurs $\wedge$ et $\vee$ et des constantes $true$ et $false$. L'automate déterministe est généré en même temps que l'exécution et permet de déterminer pour l'état suivant quelles sont les propriétés à vérifier. Puis l'automate est parcouru pendant l'exécution pour déterminer si la propriété à vérifier est vraie ou fausse. Comme point d'observation du programme, AspectJ utilise des "pointcut" qui sont des points de test correspondant à une exécution de méthode, un appel de méthode, un accès à un attribut (modification ou lecture), à une évaluation de propriété dans une structure conditionnelle (if then else), ou un point de contrôle de flot de données. AspectJ permet notamment de vérifier des propriétés paramétriques.

[Meredith 2012] présente une approche utilisant une plate-forme entièrement configurable pour un langage donné, et une logique donnée à l'aide de greffons adaptés aux langages de programmation ou aux langages de spécification. La plate-forme comporte cinq dimensions de configuration différentes. Ces dimensions sont :

- Le langage de programmation : le langage du programme à vérifier ;
- La logique utilisée : logique pour la spécification des propriétés ;
- la cible : vérification d'une classe d'invariant, vérification globale, vérification pour une interface donnée ;
- le mode d'exécution ;
- les actions à effectuer : la réaction de la plate-forme en cas de validation ou de violation de la propriété par exemple.

Dans cet article, deux langages de programmation ont été étudiés, à savoir Java (JavaMOP) et VHDL (BusMOP). Dans le cas de JavaMOP, les propriétés à vérifier sont transformées en code Java via l'outil AspectJ[Stolz 2006], qui est interfacé avec MOP.

[Drusinsky 2000] présente une approche permettant de vérifier à la fois des propriétés LTL et MTL, où MTL est une extension de la logique LTL. En MTL, tous les opérateurs de LTL sont utilisés avec des contraintes de temps. Une propriété ϕ doit donc être vérifiée dans une fenêtre de temps donnée. Par exemple, pour la propriété "Toujours, pour un temps d'exécution compris entre 10s et 20s, $x > 0$" signifie que la propriété est à vérifier dans la fenêtre de temps entre 10s et 20s mais qu'elle n'est pas à vérifier avant 10s ou après 20s. L'approche de vérification présentée consiste à transformer les formules temporelles en code exécutable fusionné avec le code à vérifier. Le code est généré automatiquement par l'outil Temporal Rover, à partir de commentaires insérés dans le code du programme et contenant les formules temporelles. Temporal Rover accepte des programmes Java, C, C++, Verilog ou VHDL. L'approche consiste à instrumenter le code du programme et à vérifier la propriété sur le code instrumenté. C'est donc une technique intrusive, qui met en œuvre la

vérification du programme en ligne, uniquement.

2.2.2 Vérification a posteriori

Le langage de EAGLE [Barringer 2003], quant à lui, utilise des équations de points fixe pour vérifier une propriété. Ces équations font intervenir les trois primitives temporelles présentées en section 2.1. Par exemple l'opérateur toujours ($\Box$), qui permet de vérifier une propriété ϕ sur l'ensemble d'une trace, suit l'équation de point fixe : $\Box\ \phi = \phi \wedge \circ\ (\Box\ \phi)$. Elle fait intervenir la primitive $\circ$ qui renvoie la vérification de la propriété opérande à l'état suivant. L'utilisation de ces équations permet donc de définir ce qu'il faut vérifier à chaque état de la trace, et ce qu'il faut vérifier à l'état suivant pour que la propriété soit vérifiée. Une implémentation de cette approche a été effectuée par la suite pour le langage Java avec Hawk System, dans [d'Amorim 2005a].

[Barringer 2010] propose une approche pour analyser des fichiers d'événements (log) pour vérifier des propriétés sur ces événements. Deux langages de propriétés sont acceptés : un langage de patron de propriétés temporelles et un langage d'automates acceptant des traces finies. Le langage de patron de propriétés temporelles est utile pour vérifier des propriétés courantes. Ces patrons sont ensuite transformés en automate. Le langage d'automate, quant à lui, complémente les lacunes du langage de patron, pour permettre la vérification de propriétés plus complexes. La vérification se fait par exécution de l'automate sur les fichiers log. L'avantage du langage de patron est qu'il simplifie l'écriture des propriétés les plus courantes. Dès que les propriétés deviennent plus complexes (non exprimables à l'aide des patrons prédéfinis), le langage des automates doit être utilisé. En terme d'expressivité, le langage d'automate permet d'écrire tout type de propriété temporelle. Cependant, l'utilisation des automates n'est pas simple pour l'utilisateur et peut rapidement être source d'erreur, si par exemple une transition est omise.

2.2.3 Cadre de la thèse

Dans le cadre de cette thèse, la vérification du logiciel se fera a posteriori, car il s'agit d'une contrainte industrielle. En effet, la plate-forme d'analyse dynamique simule l'environnement matériel du logiciel, et doit rester le plus générique possible. L'environnement matériel change avec les besoins des différents programmes. Le choix de générer une trace plutôt que de greffer un outil de vérification pour une vérification en ligne découle d'une part des exigences de traçabilité qu'exigent les standards de certification des logiciels, et d'autre part de la nécessité d'avoir une plate-forme la plus générique possible. En effet, pour chaque nouvel environnement matériel simulé, il faudrait adapter l'outil de vérification.

Un autre point fort de la plate-forme est que ce n'est pas le logiciel qui est instrumenté mais la plate-forme. De ce fait, le logiciel vérifié est celui qui est embarqué, et non pas une version altérée pour les besoins de la vérification. Le code du logiciel à vérifier ne sera donc jamais altéré, ce qui simplifie l'argumentation de certification.

2.3 Traces finies

La sémantique d'une logique temporelle se définit classiquement sur une séquence infinie d'états. Or, dans le cadre de cette thèse, la vérification se fait sur des traces finies issues de logiciels embarqués. Les opérateurs de LTL ne sont pas adaptés pour exprimer des propriétés sur des traces finies.

Exemple 1 *En effet, prenons l'opérateur faisant référence à l'état suivant* $\circ$*. Au dernier état de la trace, que devient la propriété* $\circ \phi$*, puisque l'état suivant n'existe pas ?*

Les travaux pour gérer la fin des traces d'exécution peuvent être classés en deux catégories, à savoir ceux adaptant la sémantique de la logique temporelle, et ceux adaptant la trace pour la rendre conforme à la sémantique classique.

2.3.1 Adaptation de la sémantique

2.3.1.1 Logique temporelle bivaluée

[Havelund 2002a] expose des travaux menés sur Java PathExplorer, pour une logique temporelle linéaire orientée vers le passé. Dans cet article, le choix est de modifier la sémantique des opérateurs temporels pour prendre en compte le caractère fini d'une trace. Par exemple, pour l'opérateur $\odot$, lorsqu'une propriété $\odot\ p$ est énoncée sur le dernier état, le choix effectué est de vérifier la propriété p sur le dernier état.

[Giannakopoulou 2001] propose une approche modifiant la sémantique des opérateurs LTL par l'intermédiaire de la modification de l'automate de Büchi généré à partir d'une propriété temporelle donnée. Cette approche comporte deux étapes. La première consiste à transformer la propriété temporelle en automate de Büchi à l'aide d'un tableau, mais sans définir les états finaux. La seconde consiste à définir les états finaux pour qu'une trace finie vérifiant la propriété LTL définie soit acceptée par l'automate. Concrètement, lorsque toute la trace a été parcourue, si l'automate est dans un état final, alors la propriété est considérée vérifiée. Dans le cas contraire, elle est considérée comme non vérifiée.

Le langage EAGLE dans [Barringer 2003] propose une adaptation de la sémantique en définissant des opérateurs minimum et maximum dérivant des opérateurs de LTL. En fin de trace, soit tout est considéré comme accepté (maximum) soit refusé (minimum). Cela permet de gérer finement ce que l'opérateur souhaite pour gérer la fin de la trace. Cela rend cependant le langage et la théorie plus difficiles à appréhender.

Les travaux de [Manna 1995] vont dans le même sens que EAGLE, à savoir qu'ils proposent deux opérateurs duaux dérivés de l'opérateur $\circ$, l'un retournant vrai s'il n'existe pas d'élément suivant sur lequel la propriété $\circ\ p$ est à vérifier, l'autre retournant faux. Cette logique n'est cependant pas définie pour des traces vides.

[Eisner 2003] propose deux versions de LTL pour les traces finies, LTL^+ et LTL^- basées sur une vision respectivement forte et faible de la gestion de la fin de la trace, à la manière de [Barringer 2003] et [Manna 1995], sauf que les opérateurs sont clairement séparés dans deux logiques. Une sémantique pour les traces vides est ici en outre définie. Ainsi, toute trace vide satisfera une propriété de LTL^- et ne satisfera jamais une propriété de LTL^+.

Le langage utilisé pour AspectJ dans [Stolz 2006] dérive également de la logique temporelle linéaire et a été adapté pour être compatible avec des traces finies. L'automate alternant construit à partir de la formule est ensuite utilisé pour vérifier la propriété donnée.

2.3.1.2 Logique temporelle multivaluée

[Bauer 2006a] et [Vorbehalten 2005] proposent une logique temporelle à trois valeurs $\mathbb{B}_3 = \{\bot, \top, ?\}$. $\bot$ et $\top$ sont définis comme complémentaires l'un de l'autre tandis que ? est complémentaire de lui même. Une structure en treillis de De Morgan est utilisée avec comme ordre : $\bot \subset ? \subset \top$. L'approche consiste à dire pour une propriété donnée que si pour toute trace infinie ayant pour préfixe la trace finie à vérifier la propriété est évaluée à vrai ou faux alors la propriété sera évaluée avec la même valeur. Dans le cas où l'évaluation de la propriété n'a pas la même valeur pour toutes les traces infinies dont le préfixe commun est la trace à vérifier, alors l'évaluation avec $\mathbb{B}_3$ est ?.

[Bauer 2010], quant à lui propose une logique temporelle à quatre valeurs $\mathbb{B}_4 = \{\bot, \bot^p, \top^p, \top\}$. L'approche est comparable à [Bauer 2006a, Vorbehalten 2005], autrement dit, si pour toute trace dont le préfixe est la trace vérifiée (respectivement non vérifiée), alors la propriété est évaluée à vraie (respectivement faux). Le troisième cas, à savoir lorsque l'évaluation diffère selon les traces infinies, est séparé en deux cas. La propriété est alors évaluée avec une sémantique comme celle de [Manna 1995]. Si avec cette sémantique la propriété est vraie alors $\top^p$ est renvoyé, et si elle est fausse, $\bot^p$ est renvoyé.

2.3.2 Adaptation de la trace

[Havelund 2001b] expose les travaux menés sur Java PathExplorer, pour une logique temporelle orientée vers le futur. Le logiciel facilite l'instrumentation automatique du code à vérifier. L'instrumentation consiste à ajouter des émetteurs d'événements capturés par des observateurs qui contrôlent la trace pour des propriétés définies. La vérification utilise le système de réécriture de Maude. La logique utilisée est la logique temporelle linéaire. Dans ces travaux, la sémantique de LTL est conservée et la démarche adoptée consiste à dire que dans un processus continu, pour la fin de la trace, l'état ne change pas. Autrement dit, la trace finie est transformée en trace infinie en bouclant sur le dernier état.

2.3.3 Positionnement de la thèse

Dans le cadre de cette thèse, il a été choisi d'adapter la trace d'exécution à la sémantique de la logique temporelle linéaire. La trace est ainsi rendue infinie par bouclage sur le dernier état. Grâce à ce choix, il est possible de vérifier rapidement une propriété temporelle en utilisant Ltl2ba, outil transformant une propriété LTL en automate de Büchi non déterministe. L'utilisation de Ltl2ba permet de se concentrer uniquement sur la vérification de la propriété temporelle et de disposer d'un prototype opérationnel, mais nécessite une trace infinie.

Cependant, pour pallier les manques de cette méthode, notamment pour les propriétés identifiées comme fausses du fait d'une trace incomplète (exemple : les traces sur lesquels il faut vérifier la propriété suivant le patron $\Box\ (P \Rightarrow\Diamond\ Q)$), des informations métriques additionnelles seront disponibles sur la trace, pour aider l'utilisateur à décider l'origine de la non vérification de la propriété (caractère fini de la trace ou bug).

Chapitre 3

Bilan et objectifs

Sommaire

3.1 Bilan

3.1.1 Langages de spécification

Le langage de spécification doit avoir un pouvoir d'expressivité suffisant pour spécifier des propriétés temporelles, pouvant porter sur des variables booléennes ou numériques. Il doit également répondre aux besoins spécifiques du contexte industriel.

Un langage basé sur la logique temporelle linéaire semble prometteur car le pouvoir d'expressivité de cette logique est suffisant pour les propriétés à vérifier. LTL sera donc le langage de spécification de départ. Il sera enrichi au chapitre 4 par des constructions spécifiques pour répondre aux besoins des applications industrielles (confer 4.1).

3.1.2 Runtime verification

Les propriétés sont souvent vérifiées par analyse dynamique formelle, en parallèle avec l'exécution du programme. Parmi les vérifications online, deux méthodes existent :

- la vérification inline, qui consiste à fusionner le code de vérification avec le code du programme,
- la vérification outline, qui consiste à vérifier le programme via un processus concurrent qui communique par socket avec le programme vérifié.

L'avantage de ces méthodes est qu'elles ne nécessitent pas de stocker de trace d'exécution. Cependant, la vérification d'une propriété peut avoir une incidence sur le temps d'exécution du programme à vérifier. Or, dans le cas d'un programme temps réel, une modification du temps d'exécution du programme lors de la vérification peut invalider des propriétés d'ordonnancement.

La vérification a posteriori d'une propriété nécessite de stocker la trace sur disque dur pour la vérifier. Il faut alors manipuler des traces de grandes tailles. Le nombre d'informations collectées a un impact sur l'intervalle de temps pendant lequel un logiciel peut être exécuté. En effet, l'espace mémoire disponible pour enregistrer une trace peut devenir un facteur limitant. Pour pallier ce problème, il est nécessaire d'adopter une stratégie pour minimiser la taille de la trace. La conservation de la trace est importante du point de vue de la certification car elle peut servir d'argument pour justifier le bon fonctionnement du programme vérifié.

3.1.3 Traces finies

Pour vérifier une propriété temporelle sur une trace finie, plusieurs approches existent. Une possibilité consiste à définir une logique dérivée. Des logiques multi-valuées existantes se proposent de déterminer le comportement du programme pour toute trace infinie, dont le préfixe est la trace finie à vérifier. Ce sont des logiques complexes à mettre en œuvre.

D'autres travaux proposent de modifier la sémantique des opérateurs classiques à l'aide d'opérateurs manipulant la fin de la trace différemment selon les besoins de la vérification. Ces langages sont également complexes d'utilisation.

Enfin, une possibilité consiste à adapter non pas la logique mais la trace pour revenir à un cas classique de trace infinie. Mais cette adaptation a des conséquences sur l'évaluation d'une propriété. Les travaux de cette thèse exploreront cette dernière possibilité.

3.2 Objectif

L'objectif de cette thèse consiste à vérifier des propriétés temporelles sur des traces d'exécution de programmes embarqués. Le problème de définition des cas de test eux-mêmes (choix des valeurs d'entrée) n'est pas abordé dans cette thèse. Les cas de test seront supposés définis par ailleurs. La démarche adoptée se décompose en trois étapes.

1. identifier les propriétés à vérifier et proposer un cadre de formalisation,
2. générer la ou les trace(s) d'exécution pour une propriété donnée,
3. vérifier la propriété sur la ou les trace(s).

La première étape consiste à identifier les propriétés à vérifier et à définir un cadre formel pour les exprimer. Pour ce faire, il est nécessaire de déterminer les classes de propriété à considérer (liens entre événements, durée et fréquence, ou mixte des deux). Ce classement sera fait à partir de l'étude de plusieurs applications industrielles.

La formalisation des propriétés effectuée, l'étape suivante consiste à générer la ou les trace(s) d'exécution sur la(les)quelle(s) la propriété doit être vérifiée. Des tests déjà existants servent de support à la génération des traces. La stratégie de test ne fait pas l'objet de cette thèse. Une trace d'exécution devant être stockée sur

disque dur, il est primordial de minimiser autant que possible le nombre de points d'observation utilisés pour générer cette trace, afin de minimiser la taille de cette trace. Cette minimisation doit cependant garantir que la vérification de la propriété sur la trace réduite est équivalente à la vérification de la propriété sur la trace complète [Ferlin 2012b].

Après avoir généré automatiquement la ou les trace(s) d'exécution, il est nécessaire de vérifier la propriété formalisée sur celle-ci. Dans cette optique, deux méthodes ont été évaluées [Ferlin 2012a]. L'approche de vérification la plus efficace a été ensuite choisie. Des contributions spécifiques ont été ajoutées pour traiter de façon originale le problème des traces finies.

Pour mettre en œuvre cette approche, deux prototypes, Breakpointer et AnTarES ont été implémentés. Le premier concerne la définition automatique de points d'observation en fonction d'une propriété formalisée, tandis que le second a pour objectif de vérifier une propriété temporelle sur une trace d'exécution de programme.

Enfin, pour mesurer l'efficacité de ces approches, des expérimentations ont été menées sur des logiciels embarqués, et sur des modèles de la plate-forme d'analyse dynamique.

Deuxième partie

Approche proposée

Chapitre 4

Identification et formalisation des propriétés à vérifier

Sommaire

L'objectif de ce chapitre est d'identifier les classes de propriétés à vérifier puis de les formaliser, afin qu'elles soient interprétables par un outil de vérification. Dans cette perspective, la classification des propriétés complexes effectuée en section 4.1 permettra de déterminer, dans la section 4.2, la logique existante la plus adaptée pour exprimer ces propriétés. La section 4.3 consistera à adapter cette logique à nos besoins, pour finalement formaliser, avec ce langage, les propriétés que l'on cherche à vérifier, dans la section 4.4.

4.1 Classification des propriétés

Les classes de propriétés ont été identifiées à partir de l'étude d'environ dix applications de taille variable (entre 500 et 130000 lignes de C). Pour des raisons de confidentialité, ni les noms des systèmes, ni les propriétés réelles ne seront donnés. Un exemple représentatif dans chacun des cas sera donné.

L'étude a consisté à identifier pour des logiciels embarqués de différents programmes avion Airbus, des propriétés à ce jour uniquement vérifiables par analyse, revue ou scripts ad hoc écrits manuellement. Outre les logiciels embarqués, cette étude de propriétés a également porté sur des modèles de matériels utilisés par la plate-forme d'analyse dynamique, qui simule l'environnement matériel du programme à vérifier. Ces modèles sont écrits en langage C et simulent le comportement du matériel devant accueillir le logiciel à vérifier.

Des classes de propriétés ont déjà été définies dans [Dwyer 1998]. Cette classification a été faite pour simplifier la spécification de propriétés sur des systèmes réactifs et concurrents. En effet, la proposition d'une librairie de patrons de propriétés prédéfinies facilite l'écriture d'une propriété temporelle adaptée à un programme donné. Les propriétés sont classées en deux catégories, qui sont les occurrences d'événements et les ordres d'événements.

Pour les occurrences, il s'agit des propriétés d'existence ou d'absence d'événements en un temps donné ou un intervalle de temps borné par un événement donné, voire même à tout instant.

Les propriétés d'ordre d'événements sont, quant à elles, des propriétés liant chronologiquement des événements entre eux.

Le schéma 4.1 présente ces différentes classes.

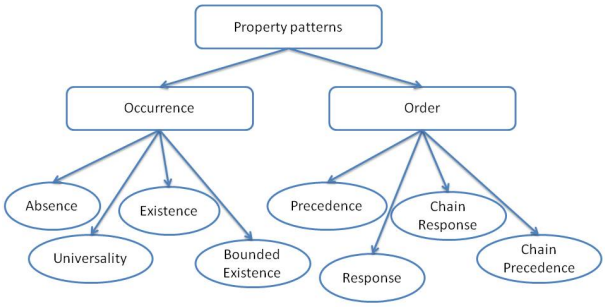

FIGURE 4.1 – Classification des propriétés selon Matthew Dwyer

Par rapport aux applications industrielles considérées, il manque dans ce classement les propriétés de durée. Nous classerons donc, les propriétés à étudier en trois grandes catégories. Lorsque c'est possible, la correspondance avec la classification de [Dwyer 1998] sera donnée.

Les catégories sont :
- les propriétés d'occurrence et d'ordre,
- les propriétés liées aux durées ou fréquences d'événements,
- les propriétés mixtes.

4.1.1 Les propriétés d'occurrence et d'ordre

Les propriétés d'ordonnancement lient des événements par leur ordre d'apparition chronologique.

4.1.1.1 Séquence d'événements

La séquence d'événements consiste à donner un ordre d'apparition à un ensemble donné d'événements. Selon la classification de [Dwyer 1998], on peut les assimiler à des *Chain precedence*.

Exemple 2 *Ce type de propriété peut classiquement se retrouver dans un protocole, tel que le protocole de téléchargement.*
- *Initialiser le protocole*
- *lire l'en-tête*
- *la vérifier*
- *boucle :*
  - *lire un paquet de données*
  - *le vérifier*
  - *le mémoriser s'il est correct*
- *arrêter le protocole.*

4.1.1.2 Réponse à un événement

L'apparition d'un événement A doit entraîner la production systématique d'un événement B dans le futur. Il s'agit de la catégorie *Response* de [Dwyer 1998].

Exemple 3 *L'absence de famine est un cas typique, pour les programmes multi-tâches. Une famine survient lorsqu'une tâche ayant demandé à s'exécuter ne peut jamais le faire. Vérifier l'absence de famine revient à vérifier que lorsqu'une tâche demande à s'exécuter, alors celle-ci finit par s'exécuter.*

À l'inverse l'occurrence d'un événement A peut être conditionnée à l'existence d'un événement antérieur B.

Exemple 4 *En raisonnant dans le passé : l'utilisation d'un port ne peut se faire que si celui-ci a bien été initialisé.*

Il est également envisageable que l'apparition d'un événement A empêche, jusqu'à la fin de l'exécution, l'apparition d'un événement B.

Exemple 5 *Si un programme sort de sa section d'initialisation, aucune nouvelle tâche ne peut être créée.*

4.1.1.3 Maîtrise des événements

Certains logiciels nécessitent, pour chaque événement A, de maîtriser l'ensemble des événements qui ont lieu, et ceci jusqu'à l'apparition d'un événement B donné. Par rapport au classement de [Dwyer 1998], ces propriétés peuvent être classées comme *Bounded Existence* si les propriétés entre A et B sont de type existence, *Absence* si ce sont des absences d'événement, où même Response, s'il s'agit de vérifier une réponse devant avoir lieu avant B.

Exemple 6 *Prenons le cas d'une application classique écrivain/lecteur avec une ressource partagée. Lorsque le programme entre dans une phase d'écriture, aucune lecture de la ressource partagée ne doit se produire jusqu'à la fin de la section d'écriture.*

De même, pendant toute la durée de la phase de lecture, il ne doit y avoir aucune action d'écriture dans la ressource partagée.

4.1.1.4 Absence d'interblocage

Dans un programme multitâche, un interblocage survient lorsque deux sections critiques de deux tâches sont entrelacées et engendrent un blocage du programme. Du fait de l'entrelacement des sections critiques, les deux tâches s'attendent mutuellement. Ce type de propriété appartient à la classe *Absence* de [Dwyer 1998].

Exemple 7 *Reprenons le cas du lecteur écrivain. Supposons qu'il n'est pas correctement implémenté : la libération du sémaphore par l'écrivain après l'écriture dans une variable n'est pas faite. Plus aucun lecteur ne peut accéder à la ressource partagée.*

En outre, à la boucle suivante, l'écrivain ne pourra pas non plus accéder à la ressource puisqu'il demandera à nouveau le sémaphore qui n'a pas été libéré.

4.1.1.5 Absence d'inversion de priorité

En programmation concurrente, une inversion de priorité survient lorsqu'une tâche de faible priorité s'exécute avant une tâche de plus forte priorité. Ce problème survient notamment lorsqu'un sémaphore a été verrouillé par la propriété de faible priorité, et empêche l'activité de la tâche de plus forte priorité. Ce type de propriété appartient à la classe *Absence* de [Dwyer 1998].

Exemple 8 *Avec le cas de l'application client écrivain, supposons que l'écrivain ait une priorité plus forte que le lecteur.*

Supposons que le lecteur soit entré en section critique. Si l'écrivain s'active, il va préempter le lecteur, mais ne pourra pas s'exécuter correctement car le verrou imposé par le lecteur l'empêchera d'entrer en section critique. De ce fait, le lecteur terminera de s'exécuter avant que l'écrivain ne puisse s'exécuter.

4.1.2 Les propriétés liées aux durées ou fréquences d'événements

Des propriétés faisant intervenir des durées ou des quantités d'événements sont parfois également à étudier.

4.1.2.1 Durée de fonctionnement

Pour des raisons de sûreté ou de performance, il peut s'avérer indispensable que des programmes ou des phases s'exécutent en un temps limité. Une phase se caractérise par un point de départ et un point d'arrivée dans le programme.

Exemple 9 *La phase d'initialisation d'un programme doit durer au maximum 10 secondes.*

4.1.2.2 Charge

Il peut s'avérer nécessaire de faire des essais de charge d'un programme pour voir s'il se comporte correctement. Un essai de charge se caractérise par le volume de données à traiter et par le temps de traitement de ce volume.

Exemple 10 *Prenons le cas d'une application client serveur. Il peut être intéressant de savoir en combien de temps le serveur exécute n requêtes, pour éviter une surcharge du serveur lors de l'exploitation.*

4.1.3 Les propriétés mixtes

Il s'agit des propriétés qui font intervenir un ordonnancement d'événements, mais qui sont également liées à une durée.

4.1.3.1 Déclenchement d'événement borné dans le temps

Lors de l'apparition d'un événement A, il peut s'avérer nécessaire de vérifier que l'événement B, qu'il engendre, survient systématiquement dans un intervalle de temps limité.

Exemple 11 *Prenons l'exemple de l'application lecteur/écrivain et ajoutons une troisième tâche qui consiste à surveiller le comportement de l'écrivain. Cette tâche doit s'assurer que lorsque l'écrivain s'active, la donnée est écrite au bout de 5 secondes. Si l'écrivain est trop lent, la tâche doit envoyer un message d'erreur.*

4.1.3.2 Déclenchement par événement répété dans le temps

Il est parfois utile d'aller plus loin en vérifiant que, lorsqu'un événement A survient au moins pendant un temps ou un nombre de fois donné, alors il engendre un événement B qui doit se produire en un temps donné.

Exemple 12 *Avec le cas de l'application écrivain lecteur, on pourrait vouloir qu'un écrivain se déclenche lorsqu'il y a eu 5 lectures d'une donnée pendant un laps de temps de 10 secondes ou bien après 10 lectures.*

4.1.3.3 Déclenchement après un temps donné

Il peut être nécessaire de vérifier qu'un événement doit s'activer après un temps donné.

Exemple 13 *Une tâche d'un programme donné s'active 10 secondes après le démarrage du programme.*

Au contraire, après un temps donné d'exécution, un événement ne peut plus se produire.

Exemple 14 *Après 10 secondes d'exécution, il n'est plus possible de créer de tâche.*

4.2 Choix d'une logique temporelle

La classification des propriétés à étudier a montré qu'il s'agissait de propriétés de relation entre événements, de durée, ou un mixte des deux. Il faut donc choisir une propriété temporelle adaptée à la formalisation de ce type de propriété. Cette logique doit prendre également en compte le fait qu'on travaillera sur des programmes dont l'exécution est finie. Enfin, cette logique doit être utilisable par un opérateur, donc être aussi simple que possible.

4.2.1 La logique temporelle arborescente

La logique temporelle arborescente ou CTL pour Computation Tree Logic repose sur une vision arborescente, discrète et infinie du temps. Elle est également orientée vers le futur. Pour chaque point dans le temps il existe donc au moins un successeur.

Cette logique, de part sa nature, a été écartée. En effet, les propriétés seront écrites sur des traces d'exécutions, à savoir des séquences d'états mémoire d'un programme. Les différents futurs possibles qu'offre cette logique ne sont donc pas utiles dans notre cas.

4.2.2 La logique temporelle temporisée

La logique temporelle temporisée a été introduite pour gérer les aspects de durée qui ne s'expriment pas en logique temporelle classique. Elle permet d'ajouter une contrainte de durée d'événement, d'intervalle de temps entre deux événements.

Cependant, la vérification de propriétés utilisant la logique temporisée fait intervenir des moyens de vérification qui semblent lourds par rapport aux propriétés que nous envisageons de vérifier. En effet, dans notre cas, les notions de durées correspondent à des comparaisons avec des temps limites. Elles sont assimilables à des événements. L'utilisation de la logique temporelle temporisée, qui fait appel à des notions de durée limite de validité pour chaque opérateur, a donc été écartée.

4.2.3 La logique temporelle linéaire

La logique temporelle linéaire ou LTL pour Linear Temporal Logic repose sur une vision linéaire, discrète, et infinie du temps. Plusieurs variantes de LTL existent. Il est possible d'avoir une LTL avec un point d'origine et orientée vers le futur, ou un point d'origine et orientée vers le passé. Ici, sera définie la LTL orientée vers le futur. Ainsi pour chaque point dans le temps il existe un et un seul successeur.

Il est également possible d'ajouter à la logique temporelle orientée futur des opérateurs du passé, qui ont une sémantique sur une trace ayant pour point de départ l'état courant et borné par l'origine. Cette logique n'augmente pas l'expressivité de la LTL mais simplifie grandement l'écriture de propriétés temporelles orientées vers le passé. C'est la PLTL.

Soit $\mathcal{P}$ un ensemble de variables propositionnelles et p un élément de $\mathcal{P}$. Soit l'alphabet $\Sigma = 2^{\mathcal{P}}$. Σ^ω est l'ensemble des mots infinis de l'alphabet Σ.

4.2.3.1 Syntaxe pour LTL

Définition 1 (Syntaxe pour LTL) *La syntaxe de la logique temporelle linéaire est définie par :*

ltl	$::= \top$	*constante true*
	$\mid \bot$	*constante false*
	$\mid p$	*variable propositionnelle*
	$\mid \neg\, ltl$	*opérateur non*
	$\mid ltl \vee ltl$	*opérateur ou*
	$\mid ltl \wedge ltl$	*opérateur et*
	$\mid \circ\, ltl$	*opérateur temporel suivant*
	$\mid \odot\, ltl$	*opérateur temporel précédent*
	$\mid \Box\, ltl$	*opérateur temporel toujours*
	$\mid \boxdot\, pltl$	*opérateur temporel toujours passé*
	$\mid \Diamond\, ltl$	*opérateur temporel futur*
	$\mid \blacklozenge\, ltl$	*opérateur temporel passé*
	$\mid ltl \;\; \mathcal{U} \;\; ltl$	*opérateur temporel jusque*
	$\mid ltl \;\; \mathcal{S} \;\; ltl$	*opérateur temporel depuis*

4.2.3.2 Sémantique pour LTL

Définition 2 (Sémantique pour LTL) *Soit un mot $\sigma = \sigma_0 ... \sigma_\omega$ un mot infini de Σ^ω. Soit ϕ et ψ, deux formules LTL. Soit $i \in \mathbb{N}$. La relation $\sigma, i \vDash \phi$ (ϕ est vrai à la position i) est définie par :*

- $\sigma, i \vDash \top$
- $\sigma, i \nvDash \bot$
- $\sigma, i \vDash p$ *si* $p \in \sigma_i$

- $\sigma, i \vDash \neg\ \phi\ si\ \sigma, i \nvDash \phi$
- $\sigma, i \vDash \psi \vee \phi\ si\ \sigma, i \vDash \phi\ ou\ \sigma, i \vDash \psi$
- $\sigma, i \vDash \psi \wedge \phi\ si\ \sigma, i \vDash \phi\ et\ \sigma, i \vDash \psi$
- $\sigma, i \vDash \circ\ \phi\ si\ \sigma, i+1 \vDash \phi$
- $\sigma, i \vDash \odot\ \phi\ si\ i > 0\ et\ \sigma, i-1 \vDash \phi$
- $\sigma, i \vDash \Diamond\ \phi\ si\ \exists j \geq i, \sigma, j \vDash \phi$
- $\sigma, i \vDash \Diamond\ \phi\ si\ \exists j \leq i, j \geq 0, \sigma, j \vDash \phi$
- $\sigma, i \vDash \Box\ \phi\ si\ \forall j \geq i, \sigma, j \vDash \phi$
- $\sigma, i \vDash \boxdot\ \phi\ si\ \forall j \leq i, j \geq 0, \sigma, j \vDash \phi$
- $\sigma, i \vDash \psi\ \mathcal{U}\ \ \phi\ si\ \exists j \geq i, \sigma, j \vDash \phi\ et\ \forall k, i \leq k < j, \sigma, i \vDash \psi$
- $\sigma, i \vDash \psi\ \mathcal{S}\ \ \phi\ si\ \exists j \leq i, j \geq 0, \sigma, j \vDash \phi\ et\ \forall k, j \leq k < i, \sigma, i \vDash \psi$

À noter que les opérateurs $\Box$ et $\Diamond$ se définissent à partir de l'opérateur $\mathcal{U}$ par les relations suivantes :

- $\Box\ P = P\ \mathcal{U}\ \bot$
- $\Diamond\ P = \top\ \mathcal{U}\ P$

La logique temporelle linéaire est suffisamment expressive pour formaliser les propriétés temporelles à vérifier. Elle permet de gérer les relations entre événements et les notions de durées adaptées aux propriétés. Elle sera donc la base du langage de spécification. Ce langage sera cependant adapté et enrichi pour être plus adapté aux besoins industriels et plus facile d'utilisation.

4.3 Langage de spécification proposé

Le langage temporel utilisé doit pouvoir prendre en compte des propriétés faisant intervenir des variables numériques. Le langage devra donc supporter des opérations de comparaison entre numériques entiers, ou entre numériques rationnels [1]. Il devra également intégrer les opérateurs numériques de base (addition, soustraction, multiplication et division).

Pour certaines propriétés rencontrées, il peut être intéressant de connaître, non pas la valeur d'une variable, mais la date à laquelle celle-ci a été modifiée pour la dernière fois. Le nombre de modification de valeur peut également s'avérer utile. De plus, la valeur d'une variable précédant une ou plusieurs modifications peut s'avérer indispensable, notamment lorsqu'il faut travailler avec des intervalles de temps. Des indicateurs de changement de sémantique devront être utilisés pour permettre de travailler avec ces données.

Il peut également s'avérer nécessaire d'utiliser des formules et des fonctions définies antérieurement. Cela évitera de réécrire des formules souvent utilisées ou simplifiera l'écriture en cas de formule complexe. Il en est de même pour l'écriture de fonctions utilisées pour des opérations numériques.

En outre, dans certains cas, il est utile de pouvoir écrire des propriétés avec une variable paramétrique. Cela évite la multiplication du nombre de propriétés à écrire, et éventuellement l'oubli de l'une d'entre elles.

1. voir 6.3.1.3 pour l'utilisation des rationnels plutôt que des flottants.

Enfin, les propriétés de séquence sont lourdes à exprimer avec la logique temporelle linéaire. De ce fait, un langage d'expressions régulières a été défini pour écrire plus simplement ce type de propriétés.

4.3.1 Syntaxe du langage utilisé

Définition 3 (Syntaxe du langage) *La syntaxe du langage utilisé est définie par :*

$Main$	$::= REVERSE\#\ Lt$	*propriété temporelle passée*
	$\mid\ Lt$	*propriété temporelle future*
	$\mid\ LtReg$	*expression régulière*

Main est le point d'entrée de la syntaxe. REV est une balise permettant de déterminer si on travaille en LTL ou en PLTL. Lt correspond à une formule de la logique temporelle.

Le langage permet également d'utiliser une expression régulière, sans opérateur temporel, définie par un langage d'expressions régulières. Le langage d'expressions régulières sera décrit en 4.3.3.

Lt	$::= Comp$	*opération de comparaison*
	$\mid\ Booleen$	*Variable et constante booléenne*
	$\mid\ (\ Lt\ )$	*parenthèses*
	$\mid\ \neg\ Lt$	*opérateur non*
	$\mid\ Lt \vee Lt$	*opérateur ou*
	$\mid\ Lt \wedge Lt$	*opérateur et*
	$\mid\ Lt \Rightarrow Lt$	*opérateur implique*
	$\mid\ \circ\ Lt$	*opérateur temporel suivant*
	$\mid\ \Box\ Lt$	*opérateur temporel toujours*
	$\mid\ \Diamond\ Lt$	*opérateur temporel futur*
	$\mid\ Lt\ \mathcal{U}\ Lt$	*opérateur temporel jusque*
	$\mid\ Formule\ (LtListe)$	*formule prédéfinie*
$LtListe$	$::= Lt$	
	$\mid\ Lt\ LtListe$	

Une propriété temporelle Lt est composée des opérateurs de la logique classique, ainsi que des opérateurs $\circ$, $\Box$, $\Diamond$ et $\mathcal{U}$ de la logique temporelle linéaire.

Les opérateurs $\Rightarrow$ et $\wedge$ sont des opérateurs de confort. En effet :

- $p \Rightarrow q = \neg\ p \vee q$
- $p \wedge q = \neg\ (\neg\ p \vee \neg\ q)$

Il est parfois utile d'avoir défini au préalable des patrons de propriétés qui sont souvent employés. C'est la possibilité qu'offre le langage, avec Formule le nom du

patron défini préalablement, et la liste de propriétés en logique temporelle LTliste correspondant aux paramètres des patrons. Il est possible de définir des patrons à partir d'autres patrons.

Exemple 15 *Supposons que nous souhaitions écrire la propriété :*

$\Box\,(p \Rightarrow (p\ \mathcal{U}\ q) \vee (p\ \mathcal{U}\ r)) \wedge \Box\,(s \Rightarrow (s\ \mathcal{U}\ t) \vee (s\ \mathcal{U}\ u))$

Pour simplifier l'écriture de cette propriété, on peut définir la formule :

$formule\_1(a,b,c) = \Box\,(a \Rightarrow (a\ \mathcal{U}\ b) \vee (a\ \mathcal{U}\ c))$

Il suffira alors d'écrire la propriété :

$formule\_1(p,q,r) \wedge formule\_1(s,t,u)$

Booleen	$::=$	$\top$	*constante true*
	$\mid$	$\bot$	*constante false*
	$\mid$	p	*variable propositionnelle*
	$\mid$	$p?n$	n^e *valeur précédente de* p *sur une trace partielle*

p est un élément de l'ensemble des variables propositionnelles $\mathcal{P}$.

Le suffixe $?n$, où n est un entier, accolé à la variable propositionnelle p permet de faire référence à la n^e valeur précédente de p.

Comp	$::=$	$Exp < Exp$	*opérateur* $<$
	$\mid$	$Exp <= Exp$	*opérateur* $\leq$
	$\mid$	$Exp > Exp$	*opérateur* $>$
	$\mid$	$Exp >= Exp$	*opérateur* $\geq$
	$\mid$	$Exp = Exp$	*opérateur* $=$
	$\mid$	$Exp\ != Exp$	*opérateur* $\neq$

Pour pouvoir définir des propriétés sur des variables numériques, il est indispensable d'ajouter les opérations de comparaisons entre numériques.

Exp	$::= Int$	*constante entière bits*
	$\mid Q$	*constante rationnelle*
	$\mid p_N$	*variable numérique*
	$\mid p_N\$n$	*variable paramétrée d'identifiant* $n \in \mathbb{N}$
	$\mid p'?T$	*temps de dernière modification de* p'
	$\mid p'?C$	*compteur de modification de* p'
	$\mid p_N?n$	n^e *valeur précédente de* p_N
	$\mid p'?n?T$	n^e *temps de modification de* p'
	$\mid p'?n?C$	n^e *compteur de modification de* p'
	$\mid Exp + Exp$	*opérateur* $+$
	$\mid Exp - Exp$	*opérateur* $-$
	$\mid -Exp$	*opérateur opposé*
	$\mid Exp * Exp$	*opérateur* $\times$
	$\mid Exp/Exp$	*opérateur* $\div$
	$\mid Exp\ div\ Exp$	*opérateur division euclidienne*
	$\mid (Exp)$	*parenthèses*
	$\mid Fonction(EListe)$	*fonction*
	$\mid Exp\&bExp$	*opérateur et bit-à-bit*
	$\mid Exp\|bExp$	*opérateur ou bit-à-bit*
	$\mid !bExp$	*opérateur non bit-à-bit*
	$\mid Exp\hat{}\,Exp$	*opérateur ou exclusif bit-à-bit*
	$\mid Exp \ll Exp$	*opérateur décalage à gauche bit-à-bit*
	$\mid Exp \gg Exp$	*opérateur décalage à droite bit-à-bit*

$EListe$	$::= Exp$	
	$\mid Exp\ EListe$	

À ce stade, la grammaire n'est pas suffisante pour s'assurer qu'une formule est bien formée. En pratique, il faut vérifier qu'une variable propositionnelle p *est bien booléenne, et qu'une variable* p_N *est soit une variable entière soit une variable rationnelle, puisque le nom d'une variable numérique et le nom d'une variable booléenne suivent les mêmes règles de syntaxe.*

Les opérateurs bit à bit nécessitent la vérification du type des propriétés, car ces opérateurs sont définis pour des mots binaires (Entiers). Ils ne sont pas définis pour les rationnels.

p_N *est un élément de l'ensemble des variables numériques* $\mathcal{P}_N$. p' *est un élément de* $\mathcal{P}_N$ *ou de l'ensemble des variables propositionnelles. Le suffixe* $?C$ *accolé à* p' *permet de faire référence au compteur des modifications de la variable* p'. *À chaque*

nouvelle modification de p', ce compteur est incrémenté. Le suffixe ?T accolé à p permet de faire référence au temps où la variable p_N est modifiée. Le suffixe ?n, où n est un entier permet de faire référence à la n^e valeur précédente de p_N sur la trace partielle.

Les suffixes ?C et ?T peuvent être accolés au suffixe ?n et permettent alors de faire référence respectivement au compteur et au temps de modification de p' lors de la n^{ieme} valeur précédente. Dans le cas où la n^{ieme} valeur n'existe pas, c'est la valeur initiale de la variable qui est donnée.

Enfin, l'opérateur opposé ($-x$) est un opérateur de confort pour $0 - x$

Exemple 16 *Soit le programme suivant :*

x=2;y=2;
y=3;
x=3;
y=4;
x=4;
y=x ;

La première ligne du programme initialise les deux variables x et y. La trace d'exécution obtenue est la suivante :

état	0	1	2	3	4	5
temps	0	10	20	30	40	50
valeur de x	2		3		4	
valeur de y	2	3		4		4

TABLEAU 4.1 – Trace d'exécution

Si l'état courant est l'état 5*, alors x?2=2, y?1=3. En effet, y a été modifié à l'état 3 et à l'état 5. Cependant, la valeur à l'état 3 et l'état 5 reste la même. y*?1 *fait donc référence à la valeur de y, avant que y prenne la valeur 4. y*?7?$T = 0$ *car il y a eu au total 3 modifications de la valeur de y (aux états 0,1 et 3). Enfin, comme il y a eu 3 modifications de la valeur de y, y*?$C = 3$.

Une variable paramétrée p_N\$n est une variable dont le domaine de définition est celui de la variable p_N. L'identifiant entier permet d'écrire plusieurs variables paramétrées indépendantes mais dont les valeurs sont celles du domaine de p_N. En pratique, la variable paramétrée sera remplacée par une constante (sous-section 6.3.5).

Les expressions numériques prennent en compte les quatre opérations classiques sur les rationnels ou des entiers de taille arbitraire. Enfin, il est également possible, comme pour les patrons de LTL, de définir et utiliser des fonctions (Fonction (EListe), où Fonction est le nom de la fonction définie).

4.3.2 Sémantique du langage utilisé

Le langage permet de travailler soit avec une logique temporelle du futur uniquement, soit avec une logique temporelle du passé uniquement. Aucun mélange des

opérateurs n'est possible. L'utilisation de la logique temporelle du futur ou du passé est un choix du spécifieur. En effet, s'il peut être plus facile d'écrire une propriété avec l'une ou l'autre des logiques, elles ont le même pouvoir d'expressivité.

De ce fait, la sémantique de la logique du future est similaire à la logique du passé. On travaillera donc avec des mots infinis. L'écoulement du temps dans un sens ou l'autre n'a pas d'incidence. Le cas de la logique du passé sera traité dans 6.3.3.

Définition 4 (État et trace) *Soit $\mathcal{V}$ un ensemble de variables et $\mathcal{VT}ype$ un ensemble possiblement infini de valeurs.*

$\mathbb{B}$, $\mathbb{N}$,$\mathbb{Z}$ et $\mathbb{Q}$ désignent respectivement l'ensemble des booléens, des entiers naturels, des entiers relatifs, et des rationnels. $\mathcal{VT}ype$ est l'union de ces ensembles. $\mathcal{VT}ype = \mathbb{B} \cup \mathbb{N} \cup \mathbb{Z} \cup \mathbb{Q}$

Un état est défini comme la fonction $\Sigma : \mathcal{V} \cup \{\tau\} \rightarrow \mathcal{VT}ype$, où τ désigne la variable temps, à valeur dans $\mathbb{N}$. Bien que les traces lues soient partielle, cette fonction est complète puisque la trace est ré-assemblée à la volée pendant l'évaluation de la propriété temporelle.

Soit $\sigma = \sigma_0 ... \sigma_\omega$ un mot infini de Σ^ω. $\sigma_i(v)$ désigne la valeur associée à $v \in \mathcal{V}$ dans l'état $\sigma_i \in \Sigma$.

Définition 5 (Numéro de l'état où une variable a été modifiée ($\mu_{\sigma,v}$)) *Soit une variable $v \in \mathcal{V}$ telle que $\sigma_i(v) \in \mathcal{VT}ype$. Soit l'ensemble $S(\sigma, i, v) = \{j \in \mathbb{N}, j \leq i$ tel que $\sigma_i(v) \neq \sigma_j(v)\}$. L'indice du dernier état tel que v ait été modifié dans σ à l'instant i est accessible via la fonction :*

$$\begin{aligned} \mu_{\sigma,v}(i) :& Max\ S(\sigma, i, v) \text{ si } S(\sigma, i, v) \neq \emptyset \\ & 0 \text{ sinon} \end{aligned} \tag{4.1}$$

Définition 6 (Généralisation de $\mu_{\sigma,v}$) *L'état tel que v ait subi la n^e modification précédente s'écrit :*

$$\mu^{n+1}_{\sigma,v}(i) = \mu^n_{\sigma,v}(\mu_{\sigma,v}(i)) \tag{4.2}$$

Définition 7 (Compteur de modifications) *Il est également possible de calculer récursivement le nombre de modifications subies par une variable :*

$$\begin{aligned} \kappa_{\sigma,v} :& \mathbb{N} \longrightarrow \mathbb{N} \\ \kappa_{\sigma,v}(0) =& 1 \\ \kappa_{\sigma,v}(n) =& \kappa_{\sigma,v}(n-1) \text{ si } \sigma_n(v) = \sigma_{n-1}(v) \\ & 1 + \kappa_{\sigma,v}(n-1) \text{ sinon} \end{aligned} \tag{4.3}$$

Si une variable subit une nouvelle affectation, mais que la valeur de la variable ne change pas, le compteur ne sera pas incrémenté. Ce choix découle de la façon dont sont définis les points d'observations dans le chapitre 5.

Exemple 17 *Supposons, par exemple que la variable observée soit dans un tableau, à l'indice 2 et qu'une boucle dans le programme à vérifier change la valeur de toutes les cases. Pour un tableau, il n'est pas possible de savoir quelle case est modifiée. Le point d'observation est activé à chaque fois qu'une case du tableau est modifiée. De ce fait, plusieurs états où tableau(2) n'est pas modifié sont collectés. Utiliser la définition précédente évite ce problème.*

Plus généralement, les points d'observations étant calculés par sur-approximation, la définition précédente empêche l'incrémentation des compteurs temps et modifications d'une variable dont une affectation ne change pas la valeur.

Définition 8 (Sémantique des opérateurs LTL) *Soit ϕ et ψ, deux formules LTL. Soit $i \in \mathbb{N}$. Soit une variable propositionnelle $p \in \mathbb{B}$. La relation $\sigma, i \vDash \phi$ (ϕ est vrai à la position i) est définie par :*

- $\sigma_i \vDash \top$
- $\sigma_i \vDash p$ *si* $\sigma_i(p) \in \mathbb{B}$
- $\sigma_i \vDash p?n$ *si* $\sigma_{\mu^{n+1}_{\sigma,p}(i)+1}(p) \in \mathbb{B}$, *si* $\sigma_{\mu^{n+1}_{\sigma,p}(i)+1}(p) \vDash p$ *et si* $n \in \mathbb{N}$
- $\sigma_i \vDash \neg\ \phi$ *si* $\sigma_i \nvDash \phi$
- $\sigma_i \vDash \psi \vee \phi$ *si* $\sigma_i \vDash \phi$ *ou* $\sigma_i \vDash \psi$
- $\sigma_i \vDash \circ\ \phi$ *si* $\sigma_{i+1} \vDash \phi$
- $\sigma_i \vDash \Diamond\ \phi$ *si* $\exists j \geq i$ tel que $\sigma_j \vDash \phi$
- $\sigma_i \vDash \Box\ \phi$ *si* $\forall j \geq i$, *alors* $\sigma_j \vDash \phi$
- $\sigma_i \vDash \psi\ \mathcal{U}\ \phi$ *si* $\exists j \geq i$, tel que $\sigma_j \vDash \phi$ *et* $\forall k, i \leq k < j$ *alors* $\sigma_i \vDash \psi$
- $\sigma_i \vDash Comp$ *si* $\sigma_i \vDash_{Comp} Comp$

Définition 9 (Sémantique des opérations de comparaison) *Soient e_1 et e_2 deux expressions numériques. La relation pour les opérations de comparaison $\sigma, i \vDash_{Comp} Comp$ est définie par :*

- $\sigma, i \vDash_{Comp} e_1 \bowtie e_2$ *si* $\exists (e_{1v}, e_{2v}) \in \mathbb{Z}^2$ tel que $[\![e_1]\!]_{\mathbb{Z}}(\sigma_i) = e_{1v}$ *et* $[\![e_2]\!]_{\mathbb{Z}}(\sigma_i) = e_{2v}$ *et* $e_{1v} \bowtie e_{2v}$
- $\sigma, i \vDash_{Comp} e_1 \bowtie e_2$ *si* $\exists (e_{1v}, e_{2v}) \in \mathbb{Q}^2$ tel que $[\![e_1]\!]_{\mathbb{Q}}(\sigma_i) = e_{1v}$ *et* $[\![e_2]\!]_{\mathbb{Q}}(\sigma_i) = e_{2v}$ *et* $e_{1v} \bowtie e_{2v}$

avec $\bowtie \in \{\leq, \geq, <, >, =, \neq\}$.

Définition 10 (Sémantique des expressions numériques dans $\mathbb{Z}$) *Soient e_1 et e_2 deux expressions numériques. Soit $c_{\mathbb{Z}} \in \mathbb{Z}$ une constante. Soit $x \in \mathcal{V}$ une variable numérique (à valeur dans $\mathbb{Z}$ ou dans $\mathbb{Q}$). Soit $n \in \mathbb{N}$. La relation pour les entiers est définie par :*

- $[\![c_{\mathbb{Z}}]\!]_{\mathbb{Z}}(\sigma_i) = c_{\mathbb{Z}}$.
- $[\![x]\!]_{\mathbb{Z}}(\sigma_i) = \sigma_i(x)$ *si* $\sigma_i(x) \in \mathbb{Z}$.
- $[\![x?n]\!]_{\mathbb{Z}}(\sigma_i) = \sigma_{\mu^{n+1}_{\sigma,x}(i)+1}(x)$ *si* $\sigma_i(x), \sigma_{\mu^{n+1}_{\sigma,x}(i)+1}(x) \in \mathbb{Z}^2$.
- $[\![x?T]\!]_{\mathbb{Z}}(\sigma_i) = \sigma_{\mu_{\sigma,x}(i)+1}(\tau).(\sigma_i(x), \sigma_{\mu_{\sigma,x}(i)+1}(x) \in (\mathbb{Z} \cup \mathbb{Q} \cup \mathbb{B})^2)$
- $[\![x?T?n]\!]_{\mathbb{Z}}(\sigma_i) = \sigma_{\mu^{n+1}_{\sigma,x}(i)+1}(\tau).(\sigma_i(x), \sigma_{\mu^{n+1}_{\sigma,x}(i)+1}(x) \in (\mathbb{Z} \cup \mathbb{Q} \cup \mathbb{B})^2)$
- $[\![x?C]\!]_{\mathbb{Z}}(\sigma_i) = \kappa_{\sigma,x}(i)$ $(\sigma_i(x) \in \mathbb{Z} \cup \mathbb{Q} \cup \mathbb{B})$.
- $[\![x?C?n]\!]_{\mathbb{Z}}(\sigma_i) = \kappa_{\sigma,x}(\mu^{n+1}_{\sigma,x}(i)+1).(\sigma_i(x), \sigma_{\mu^{n+1}_{\sigma,x}(i)+1}(x) \in (\mathbb{Z} \cup \mathbb{Q} \cup \mathbb{B})^2)$

- $[\![-e]\!]_{\mathbb{Z}}(\sigma_i) = -[\![e]\!]_{\mathbb{Z}}(\sigma_i)$
- $[\![e_1 + e_2]\!]_{\mathbb{Z}}(\sigma_i) = [\![e_1]\!]_{\mathbb{Z}}(\sigma_i) + [\![e_2]\!]_{\mathbb{Z}}(\sigma_i)$
- $[\![e_1 - e_2]\!]_{\mathbb{Z}}(\sigma_i) = [\![e_1]\!]_{\mathbb{Z}}(\sigma_i) - [\![e_2]\!]_{\mathbb{Z}}(\sigma_i)$
- $[\![e_1 * e_2]\!]_{\mathbb{Z}}(\sigma_i) = [\![e_1]\!]_{\mathbb{Z}}(\sigma_i) \times [\![e_2]\!]_{\mathbb{Z}}(\sigma_i)$
- $[\![e_1 \text{ div } e_2]\!]_{\mathbb{Z}}(\sigma_i) = [\![e_1]\!]_{\mathbb{Z}}(\sigma_i) \text{ div } [\![e_2]\!]_{\mathbb{Z}}(\sigma_i)$ *où div est la division euclidienne.*
- $[\![! e]\!]_{\mathbb{Z}}(\sigma_i) =! [\![e]\!]_{\mathbb{Z}}(\sigma_i)$, *où* ! *est l'opérateur logique bit à bit négation.*
- $[\![e_1 \,\&\, e_2]\!]_{\mathbb{Z}}(\sigma_i) = [\![e_1]\!]_{\mathbb{Z}}(\sigma_i) \,\&\, [\![e_2]\!]_{\mathbb{Z}}(\sigma_i)$, *où* & *est l'opérateur logique bit à bit conjonction.*
- $[\![e_1 \mid e_2]\!]_{\mathbb{Z}}(\sigma_i) = [\![e_1]\!]_{\mathbb{Z}}(\sigma_i) \mid [\![e_2]\!]_{\mathbb{Z}}(\sigma_i)$, *où* | *est l'opérateur logique bit à bit disjonction.*
- $[\![e_1 \,\hat{}\, e_2]\!]_{\mathbb{Z}}(\sigma_i) = [\![e_1]\!]_{\mathbb{Z}}(\sigma_i) \,\hat{}\, [\![e_2]\!]_{\mathbb{Z}}(\sigma_i)$, *où* ^ *est l'opérateur logique bit à bit "ou exclusif".*
- $[\![e_1 \ll e_2]\!]_{\mathbb{Z}}(\sigma_i) = [\![e_1]\!]_{\mathbb{Z}}(\sigma_i) \ll [\![e_2]\!]_{\mathbb{Z}}(\sigma_i)$, *où* $\ll$ *est l'opérateur de décalage à gauche.*
- $[\![e_1 \gg e_2]\!]_{\mathbb{Z}}(\sigma_i) = [\![e_1]\!]_{\mathbb{Z}}(\sigma_i) \gg [\![e_2]\!]_{\mathbb{Z}}(\sigma_i)$, *où* $\gg$ *est l'opérateur de décalage à droite.*

Définition 11 (Sémantique des expressions numériques dans $\mathbb{Q}$) *Soient e_1 et e_2 deux expressions numériques. Soit $c_{\mathbb{Q}} \in \mathbb{Q}$ une constante. Soit $x \in \mathcal{V}$ une variable numérique (à valeur dans $\mathbb{Q}$). La relation pour les rationnels est définie par :*

- $[\![c_{\mathbb{Q}}]\!]_{\mathbb{Q}}(\sigma_i) = c_{\mathbb{Q}}$.
- $[\![x]\!]_{\mathbb{Q}}(\sigma_i) = \sigma_i(x)$ si $\sigma_i(x) \in \mathbb{Q}$
- $[\![x?n]\!]_{\mathbb{Q}}(\sigma_i) = \sigma_{\mu^{n+1}_{\sigma,x}(i)+1}(x)$ si $\sigma_i(x), \sigma_{\mu^{n+1}_{\sigma,x}(i)+1}(x) \in \mathbb{Q}^2$
- $[\![-e]\!]_{\mathbb{Q}}(\sigma_i) = -[\![e]\!]_{\mathbb{Q}}(\sigma_i)$
- $[\![e_1 + e_2]\!]_{\mathbb{Q}}(\sigma_i) = [\![e_1]\!]_{\mathbb{Q}}(\sigma_i) + [\![e_2]\!]_{\mathbb{Q}}(\sigma_i)$
- $[\![e_1 - e_2]\!]_{\mathbb{Q}}(\sigma_i) = [\![e_1]\!]_{\mathbb{Q}}(\sigma_i) - [\![e_2]\!]_{\mathbb{Q}}(\sigma_i)$
- $[\![e_1 * e_2]\!]_{\mathbb{Q}}(\sigma_i) = [\![e_1]\!]_{\mathbb{Q}}(\sigma_i) \times [\![e_2]\!]_{\mathbb{Q}}(\sigma_i)$
- $[\![e_1/e_2]\!]_{\mathbb{Q}}(\sigma_i) = [\![e_1]\!]_{\mathbb{Q}}(\sigma_i)/[\![e_2]\!]_{\mathbb{Q}}(\sigma_i)$

4.3.3 Le langage d'expressions régulières

Le langage d'expressions régulières a été défini pour exprimer simplement des propriétés de séquences d'événements.

4.3.3.1 Les types de séquences

Les séquences d'événements peuvent être organisées en trois classes :
- les séquences fortes,
- les séquences étendues,
- les séquences faibles.

La figure 4.2 présente les trois types de séquences.

Dans la suite, une séquence d'événements (prédicats) possède au minimum 2 éléments. $n \in \mathbb{N}^* \backslash \{1\}$ est l'arité de la séquence. Les éléments de la séquence sont $x_0, ..., x_{n-1}$, où x_i est un prédicat. Trois types de séquences sont définis ci-dessous :

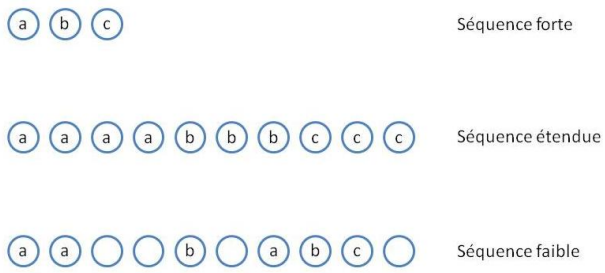

FIGURE 4.2 – Les types de séquence

Définition 12 (Séquence forte) *Une trace d'exécution satisfait une séquence forte si*

- *Pour tout* $i \in [\![0, n-2]\!]$*, il ne peut pas y avoir d'événement s'intercalant entre* x_i *et* x_{i+1}
- *Pour tout* $i \in [\![0, n-2]\!]$*,* x_i *n'est pas répété sur plusieurs états successifs.*

Définition 13 (Séquence étendue) *Une trace d'exécution satisfait une séquence étendue si*

- *Pour tout* $i \in [\![0, n-2]\!]$*, il ne peut y avoir d'événement s'intercalant entre* x_i *et* x_{i+1}
- *Pour tout* $i \in [\![0, n-2]\!]$*, la répétition de* x_i *est autorisée sur plusieurs états successifs.*

Définition 14 (Séquence faible) *Une trace d'exécution satisfait une séquence faible si*

- *Pour tout* $i \in [\![0, n-2]\!]$*, il peut y avoir un nombre fini d'événements s'intercalant entre* x_i *et* x_{i+1}*, y compris des événements* x_j *pour tout* $j \in [\![0, n-1]\!]$
- *Pour tout* $i \in [\![0, n-2]\!]$*, la répétition de* x_i *est autorisée sur plusieurs états successifs. Cette règle est une particularisation de la règle précédente.*

4.3.3.2 Exemple de séquences écrites en LTL

Il est peu aisé d'écrire une propriété de séquence à l'aide de LTL. Pour chaque type de séquences défini en 4.3.3.1, sa version en logique temporelle est présentée de façon récursive, de manière à en contracter l'écriture. Les notations $seq_{f,n}$, $seq_{E,n}$ et $seq_{F,n}$ sont temporaires. Elles seront définies différemment par la suite.

Exemple 18 *Une séquence forte à n éléments* ($seq_{F,n}$) *peut être écrite en LTL, à l'aide de la sous formule* ϕ_F *comme :*

$$
\begin{aligned}
\phi_F(a,b) &= a \wedge\!\circ\, b \\
\forall n \in \mathbb{N}, & \\
seq_{F,1}(a_0,a_1) &= \phi_F(a_0,a_1) \\
seq_{F,2}(a_0,a_1,a_2) &= \phi_F(a_0,\phi_F(a_1,a_2)) \\
seq_{F,n}(a_0,..,a_n) &= \phi_F(a_0,\phi_F(a_1,\phi_F(a_2,...a_n)...) = \quad seq_{F,n-1}(a_0,...,a_{n-2},\phi(an_1,a_n))
\end{aligned}
$$

Exemple 19 *Une séquence étendue à n éléments ($seq_{E,n}$) peut s'écrire en LTL, en fonction de la fonction ϕ_E intermédiaire définie ci-dessous, par :*

$$
\begin{aligned}
\forall n \in \mathbb{N}, & \\
\phi_E(a,b) &= a \Rightarrow (aUb) \\
seq_{E,1}(a_0,a_1) &= a_0 \wedge \phi_E(a_0,a_1) \\
seq_{E,2}(a_0,a_1,a_2) &= a_0 \wedge \phi_E(a_0,a_1) \wedge \phi_E(a_1,a_2) \\
seq_{E,n}(a_0,...,a_n) &= a_0 \wedge_{i=0}^{n-1} \phi_E(a_i,a_{i+1}) = \quad seq_{E,n-1}(a_0,...,a_{n-1}) \wedge \phi_E(a_{n-1},a_n)
\end{aligned}
$$

Exemple 20 *Enfin, une séquence faible à n éléments ($seq_{f,n}$) peut s'écrire en LTL, avec l'aide de la sous-formule ϕ_f de la manière suivante :*

$$
\begin{aligned}
\forall n \in \mathbb{N}, & \\
\phi_f(a,b) &= a \Rightarrow\!\Diamond\, b \\
seq_{f,1}(a_0,a_1) &= a_0 \wedge \phi_f(a_0,a_1) \\
seq_{f,n}(a_0,...,a_n) &= a_0 \wedge_{i=0}^{n-1} \phi_f(a_i,a_{i+1}) = \quad seq_{f,n}(a_0,...,a_{n-1}) \wedge \phi_f(a_{n-1},a_n)
\end{aligned}
$$

Les propriétés temporelles de séquences s'avèrent très vite lourdes à écrire. De plus, dans le chapitre 8, on verra que l'utilisation de ce type de propriétés a des conséquences majeures sur le temps de vérification d'une propriété temporelle. C'est pourquoi il a été décidé d'écrire un langage d'expressions régulières inspiré de PSL [Foster] pour définir efficacement des propriétés de séquences. En outre, le chapitre 6 présentera un mécanisme pour éviter l'explosion du temps de vérification d'une propriété de séquence.

4.3.3.3 Syntaxe

Définition 15 (Syntaxe du langage d'expressions régulières) *La syntaxe du langage d'expressions régulières est définie par :*

$Main$	$::=$	Sre	*Séquence se produisant 1 fois*
	$\mid$	$Sre\ Predicat$	*Séquence se produisant jusqu'à ce que Predicat soit vrai*
	$\mid$	$Sre*$	*séquence se produisant n fois,* $n \in \mathbb{N}$
	$\mid$	$Sre+$	*séquence se produisant n fois,* $n \in \mathbb{N}^*$
Sre	$::=$	$Sequence$	*Séquence*
	$\mid$	$[Predicat]\ Sequence$	*Séquence conditionnée par Predicat*
$Sequence$	$::=$	$(Liste)$	*Séquence faible*
	$\mid$	$[Liste]$	*Séquence étendue*
	$\mid$	$\{Liste\}$	*Séquence forte*
$Liste$	$::=$	$Predicat$	
	$\mid$	$Predicat\ ;\ Liste$	
$Predicat$	$::=$	$(Predicat)$	*parenthèses*
	$\mid$	$\neg\ Predicat$	*opérateur non*
	$\mid$	$Predicat \vee Predicat$	*opérateur ou*
	$\mid$	$Predicat \wedge Predicat$	*opérateur et*
	$\mid$	$Predicat \Rightarrow Predicat$	*opérateur implique*
	$\mid$	$Comp$	*operation de comparaison, variable, constante*

Les propriétés de logique propositionnelles sont utilisées pour les expressions régulières. Elles ne peuvent inclure aucun opérateur de la logique temporelle.

4.3.3.4 Sémantique

Définition 16 (Sémantique du langage d'expressions régulières) *Soit un mot* $\sigma = \sigma_0 ... \sigma_\omega$. *Soit une liste ordonnée de* $n+1$ *(*$n \in \mathbb{N}^*$*) propositions de la logique propositionnelle* $a_0, ..., a_n$. *La sémantique du langage d'expressions régulières sur* σ *est la suivante :*

- $\sigma_i \vDash \{a_0; ...; a_n\}$ *si* $\forall j \in [\![0; n]\!], \sigma_{j+i} \vDash a_j$
- $\sigma_i \vDash [a_0; ...; a_n]$ *si* $\exists (u_j)_{j \in [\![0,n]\!]}, u_0 = i$ *et* $\forall k \in [\![1, n-1]\!], u_{k+1} > u_k$ *et* $\forall k' \in [\![0, n]\!], \sigma_{u_{k'}} \vDash a_{k'}$ *et* $\forall n_k \in [\![u_k, u_{k+1}]\!], \sigma_{u_k} \vDash a_k$
- $\sigma_i \vDash (a_0; ...; a_n)$ *si* $\exists (u_j)_{j \in [\![0,n]\!]}, u_0 = i$ *et* $\forall k \in [\![1, n-1]\!], u_{k+1} > u_k$ *et* $\forall k' \in [\![0, n]\!], \sigma_{u_{k'}} \vDash a_{k'}$

Pour la suite, Seq_n *désigne une séquence quelconque (faible, forte, étendue) bâtie à partir de la liste ordonnée* $a_0, ..., a_n$.

- $\sigma_i \vDash Seq_n+$ *si* $\exists m \in \mathbb{N} * \cup \infty, \exists (u_n)_{n \in [\![1,m]\!]}, \forall n \in [\![1, m]\!], \sigma_{u_n} \vDash Seq_n$

– $\sigma_i \vDash Seq_n*$ *si* $\exists m \in \mathbb{N} \cup \infty, \exists (u_n)_{n \in [\![1,m]\!]}, \forall n \in [\![1,m]\!], \sigma_{u_n} \vDash Seq_n$
– $\sigma_i \vDash Seq_n[p]$ *si* $\exists m \in \mathbb{N} \cup \infty, \sigma_m \vDash p \wedge a_n, \exists (u_n)_{n \in [\![0,m]\!]}, \forall n \in [\![0,m]\!], \sigma_{u_n} \vDash Seq_n$

Pour la suite, Seq'_n *désigne une séquence de modalité quelconque, répétable (*,+,[p]) ou non, bâtie à partir de la liste ordonnée* $a_0, ...a_n$.

– $\sigma_i \vDash [p]Seq'_n$ *si* $\sigma_i \vDash p$ *et* $\sigma_i \vDash Seq'_n$ *ou si* $\sigma_i \nvDash p$

Cas particuliers :

– $\sigma_0 \vDash [1]Seq'_n$ *si* $\sigma_0 \vDash Seq'_n$.
– $\sigma_0 \vDash [0]Seq'_n$ *si* $\exists j > 0, \sigma_i \vDash Seq'_n$.

4.3.3.5 Exemple de propriétés de séquence écrite avec des expressions régulières

En reprenant les exemples décrits dans 4.3.3.2 on constate qu'avec le langage d'expressions régulières, il est bien plus simple d'écrire une propriété de séquence.

Pour cette section, on définit pour tout $n \in \mathbb{N}$, $\psi(a_0, ...a_n)$ comme étant :

$\psi_1(a_0, a_1) = a_0; a_1$
$\psi_n(a_0, ...a_n) = \psi_{n-1}(a_0, ...a_{n-1}); a_n$

Exemple 21 *On peut donc écrire une séquence forte comme :*
$\forall n \in \mathbb{N}, \{\psi_n(a_0, ...a_n)\}$

Exemple 22 *Une séquence étendue s'écrira donc :*
$\forall n \in \mathbb{N}, [\psi_n(a_0, ...a_n)]$

Exemple 23 *Enfin, pour une séquence faible, on a :*
$\forall n \in \mathbb{N}, (\psi_n(a_0, ...a_n))$

On dispose donc de deux langages différents. Le premier sera adapté à l'écriture des propriétés temporelles classifiées en section 4.1 et peut faire appel à des prédicats ou fonctions écrits au préalable, dans le but de simplifier le travail de l'opérateur. Le second langage sera particulièrement adapté aux propriétés de séquences, qui certes sont spécifiables avec le premier langage, mais qui sont rapidement complexes à écrire et à manipuler.

4.4 Exemple de propriétés formalisées

Maintenant, les deux langages d'expression de propriétés étant présentés, il reste à formaliser les propriétés classifiées de la section 4.1.

4.4.1 Les propriétés d'occurrence et d'ordre

4.4.1.1 Séquence d'événements

Nous traiterons la formalisation de l'exemple 2.

Pour l'écriture du protocole en logique formelle, il faut procéder en deux temps. En effet, il y a la séquence principale et la séquence de la boucle. La boucle se définit comme une étape de la séquence principale. La séquence de la boucle sera vérifiée avec une seconde propriété.

Définissons une variable s pour chaque étape de la séquence principale, et attribuons lui une valeur spécifique. De même pour la boucle, en définissant une variable b.

Les tableaux 4.2 regroupent les informations sur les valeurs de s et b

Événement	valeur de s
Neutre	0
Initialisation	1
Lecture entête	2
Vérification entête	3
Boucle	4
Arrêt protocole	5

Événement	valeur de b
Neutre	0
lire un paquet	1
vérifier un paquet	2
mémoriser un paquet	3

TABLEAU 4.2 – Correspondance entre les événements et les valeurs de s et b

Les expressions régulières seront utilisées pour définir simplement les deux propriétés à vérifier.

Les propriétés s'écriront donc :

$$\begin{aligned}[s=1;s=2;s=3;s=4;s=5]*\\ [s=4][b=1;b=2;b=3]*\end{aligned} \tag{4.4}$$

4.4.1.2 Réponse à un événement

Absence de famine. Reprenons l'exemple 3.

Soit la variable f_i caractérisant l'appel de la fonction de prise de sémaphore pour la tâche i. Nous supposerons qu'il y a un même sémaphore pour n tâches données. Pour rappel, un sémaphore est une variable permettant la restriction d'accès à des ressources partagées, le but étant d'éviter que deux tâches accèdent simultanément à une ressource partagée (l'une en écriture l'autre en lecture, ou les deux en écriture, par exemple). Le tableau 4.3 présente les différentes valeurs de f_i, pour tout i dans $[\![1;n]\!]$ et leurs significations.

Événement	valeur de f_i
Neutre (fin d'accès à une section critique)	0
Demande d'accès à une section critique	1
Accès à une section critique obtenue	2

TABLEAU 4.3 – Correspondance entre la valeur de f_i et les événements

La propriété formalisée est, pour chaque i :

$$\Box\,(f_i = 1 \Rightarrow \Diamond\,(f_i = 2)) \tag{4.5}$$

Cette formule peut également s'écrire, avec les séquences :

$$[f_i = 1; f_i = 2]* \tag{4.6}$$

Initialisation d'un objet avant son utilisation. Reprenons l'exemple 4. Nous souhaitons nous assurer qu'avant l'utilisation d'un port, celui-ci doit avoir été préalablement initialisé.

Cette propriété est un peu particulière, car il faudrait connaître à l'avance l'ensemble des valeurs des ports utilisées pour pouvoir vérifier que le port a bien été créé.

Nous raisonnerons ici avec des propriétés paramétriques (voir sous-section 6.3.5). Soit p_c une variable entière indiquant que le port p_c a été créé. Soit p_u une variable entière indiquant que le port p_u a été utilisé.

Soit n un entier. La propriété à vérifier s'exprime en logique du passé, pour tout n, de la façon suivante :

$$\boxdot ((p_u = n) \Rightarrow \Diamond (p_c = n)) \tag{4.7}$$

Avec notre langage, nous l'écrirons donc de la manière suivante :

$$REVERSE\# \Box (p_u = n \Rightarrow \Diamond (p_c = n)) \tag{4.8}$$

Pas de création de tâche après la fin de la section d'initialisation. Dans le cas de l'exemple 5, la propriété permet de s'assurer qu'en dehors de la section d'initialisation, il est impossible de créer une nouvelle tâche.

Soit *initial* la variable booléenne qui est à *true* si pour toute la section d'initialisation et à $false$ sinon.

Un appel à la fonction de création de tâche se modélise grâce à la variable *nouvelle_tache* qui est à *true* dès que la fonction de création est appelée et qui repasse à $false$ dès que la nouvelle tâche est créée.

La propriété formalisée est donc :

$$\Box (initial \Rightarrow \neg\, nouvelle\_tache) \tag{4.9}$$

4.4.1.3 Maîtrise des événements

Dans l'exemple 6, il ne doit pas y avoir de lecture quand il y a écriture d'une variable. Nous supposerons qu'il y a n tâches.

Soit la variable $e_{i,r}$, pour la tâche i qui est à *true* lorsque que l'écriture dans une variable partagée r commence et à $false$ lorsque l'écriture est finie, ou lorsqu'il n'y a pas d'écriture en cours.

Soit la variable $l_{i,r}$, pour la tâche i qui suit le même principe que la variable e_i, mais pour la lecture de la variable partagée r.

Il suffit alors de vérifier pour tout i et pour toute ressource partagée que :

$$\Box (e_{i,r} \Rightarrow \neg\, l_{i,r}) \tag{4.10}$$

4.4.1.4 Absence d'interblocage

Reprenons le formalisme adopté en 4.4.1.2. Il est également possible de s'assurer que lorsqu'une tâche rentre en section critique, elle finit par en sortir avec la propriété suivante, pour chaque i

$$\Box\ (f_i = 2 \Rightarrow \Diamond\ (f_i = 0)) \tag{4.11}$$

Cette seconde équation permet de traiter le cas des interblocages.

4.4.1.5 Absence d'inversion de priorité

Traitons le cas de l'exemple dans 4.1.1.5. Soit une variable entière *tache* permettant de représenter la tâche courante traitée. Le numéro de la tâche est également son numéro de priorité. Plus *tache* est grand, plus la priorité est grande. Nous prendrons un exemple à trois tâches.

Il s'agit donc de s'assurer de la propriété de séquencement suivante, à supposer que les priorités sont fixes.

$$[tache = 1; tache = 2; tache = 3]* \tag{4.12}$$

4.4.2 Les propriétés liées aux durées ou fréquences d'événements

4.4.2.1 Durée de fonctionnement

Nous cherchons à vérifier (exemple de la section 4.1.2.1) qu'une phase d'initialisation d'un programme est de 10 secondes au plus.

Si *temps* est la variable entière contenant le temps exprimé en millisecondes et *initial* la variable booléenne à *true* pendant la phase d'initialisation et à $false$ sinon, alors les propriétés s'écrivent de la façon suivante :

$$\Box\ ((temps > 10000) \Rightarrow \neg\ initial) \tag{4.13}$$

$$initial \tag{4.14}$$

$$\Box\ (initial \Rightarrow \Diamond\ (\neg\ initial)) \tag{4.15}$$

L'équation 4.13 traduit l'idée de limite, tandis que l'équation 4.14 s'assure que la phase d'initialisation démarre dès le premier état et l'équation 4.15 permet de s'assurer qu'un jour le programme sortira de la section initiale. Seule la première équation est indispensable. Les deux autres permettent éventuellement de préciser davantage le comportement du programme.

4.4.2.2 Charge

Nous souhaitons effectuer un test de charge, comme dans l'exemple de la section 4.1.2.2.

Soit la variable *temps*1 représentant la valeur du temps au début du test de charge et *temps* la valeur du temps courant. Il faut s'assurer que le serveur traite

N (constante) requêtes. Soit T_N (constante) le temps maximum que doit mettre le serveur pour traiter N requêtes. Soit n le numéro de la requête traitée. n s'incrémente de 1 pour chaque nouvelle requête traitée.

La propriété formalisée est donc :

$$\Box\ ((temps - temps1 < T_N) \wedge (temps - temps1 >= T_N) \Rightarrow n >= N) \qquad (4.16)$$

4.4.3 Les propriétés mixtes

4.4.3.1 Déclenchement d'événement borné dans le temps

Reprenons l'exemple de la section 4.1.3.1.

Soit un événement A et $tempsA$ l'instant auquel A apparaît. Soit $temps$ le temps courant. Soit l'événement B et $tempsB$ le temps auquel B apparaît. La variable entière $evenement$ permet de définir si A survient ($evenement = 1$), B apparaît ($evenement = 2$) ou si le programme est dans un état neutre ($evenement = 0$). T_0 est l'intervalle de temps limite à ne pas franchir entre l'apparition de A et de B.

Les propriétés formalisées sont donc :

$$\Box\ (evenement = 1 \Rightarrow\Diamond\ (evenement = 2)) \qquad (4.17)$$

La propriété 4.17 permet de spécifier l'existence de B.

$$\Box\ (evenement = 2 \Rightarrow (tempB - tempA) < T_0) \qquad (4.18)$$

La propriété 4.18 permet de spécifier que l'intervalle de temps maximum est bien respecté.

$$\Box\ (evenement = 2 \Rightarrow\Diamond\ evenement = 0)) \qquad (4.19)$$

La propriété 4.19 permet de spécifier que $evenement$ revient bien à zéro. Cette propriété est optionnelle.

4.4.3.2 Déclenchement par événement répété dans le temps

Reprenons l'exemple de la section 4.1.3.2. Cette propriété est complexe à formaliser. En effet, il est nécessaire de pouvoir compter le nombre de changements de valeur d'une variable témoin de la lecture pour la propriété sur le nombre d'occurrences de lecture. Supposons que nous ayons de ce compteur représenté par $compte\_lecture$. Le lancement d'une section d'écriture sera caractérisé par la variable $ecriture$ égale à $true$. Cette variable est à $false$ lorsqu'il n'y a pas d'écriture.

Formellement, nous avons donc :

$$\Box\ (compte\_lecture = 10 \Rightarrow\Diamond\ (ecriture)) \qquad (4.20)$$

S'il doit y avoir écriture 10 secondes après la dernière lecture, au plus, il faut connaître le temps $temps\_d\_lecture$ de la dernière lecture. Soit $temps$ le temps courant exprimé en nombres entiers, en millisecondes.

$$\Box\ ((temps - temps\_d\_lecture) > 10000 \Rightarrow\Diamond\ (ecriture)) \qquad (4.21)$$

4.4.3.3 Déclenchement après un temps donné

Pour les deux exemples suivants, tirés de la section 4.1.3.3, soit *temps* le temps courant.

Activation d'une tâche après au moins 10 secondes d'exécution. Avec l'exemple 13, soit $t\_activationT$ le temps d'activation de la tache T et T le booléen qui est à *true* quand la tâche T s'active. T est à $false$ sinon.

La propriété s'exprime de la façon suivante :

$$\Box\ ((temps < t\_activationT \Rightarrow \neg\ T) \wedge (temps >= t\_activationT \Rightarrow \Diamond\ (T)) \tag{4.22}$$

Impossibilité de créer une tâche après 10 secondes. Reprenons l'exemple 14

Soit $creer\_tache$ une variable booléenne qui est à vrai lorsque la fonction de création de variable est appelée et qui repasse à faux lorsque la tâche est bien créée.

La propriété formalisée est donc :

$$\Box\ (temps > 10000 \Rightarrow \neg\ creer\_tache) \tag{4.23}$$

Conclusion du chapitre

L'objectif principal de ce chapitre était de formaliser les propriétés pour qu'elles soient interprétables par un outil de vérification. Pour ce faire, on a classifié les propriétés afin de choisir le langage le plus adapté. Ce choix effectué, une adaptation a été nécessaire pour simplifier l'écriture des propriétés. Enfin, les propriétés à vérifier ont été formalisées. À présent, il faut vérifier la propriété sur une trace d'exécution générée automatiquement.

Chapitre 5

Génération automatique de traces d'exécution

Sommaire

L'objectif de ce chapitre est de générer automatiquement des traces d'exécution contenant les informations nécessaires à la vérification d'une propriété donnée. Il est indispensable que ces traces d'exécution soient les plus petites possibles. La trace d'exécution est générée à partir de la définition de ces points d'observation et de cas de tests déjà existants. La stratégie de test n'est pas abordée dans le cadre de cette thèse. La section 5.1 définira les contraintes liées à la plate-forme d'analyse dynamique étudiée, et les possibilités offertes par l'analyse statique pour la définition de points d'observation. La section 5.2 détaillera l'approche choisie pour générer automatiquement des points d'observation. La correction et les limites observées de cette approche feront l'objet de la section 5.3

5.1 Environnement d'exécution

5.1.1 La plate-forme d'analyse dynamique et ses contraintes

La vérification de propriétés temporelles se fera sur des programmes C dans un contexte de certification suivant la norme DO-178. La modification du programme en vue de sa vérification est donc prohibée.

Pour vérifier une propriété sur un programme, on dispose d'une plate-forme d'exécution dynamique permettant d'exécuter un programme sur un matériel simulé correspondant aux calculateurs embarqués.

Cette plate-forme dispose d'une interface de débogage classique qui, pour les besoins industriels, a été adaptée. Les modifications de l'interface permettent notamment d'enregistrer via un processus concurrent les valeurs des variables analysées dans une base de données dont le schéma est entièrement définissable selon les besoins.

Le schéma de la base de données choisi est une table de quatre colonnes, contenant le numéro de l'état (entier), le temps (entier), le nom de la variable modifiée (chaîne de caractères) et la valeur de la variable modifiée (chaîne de caractères). Le temps correspond au temps simulé. Il est calculé à partir du nombre de cycle du processeur simulé et du temps d'un cycle.

L'utilisation d'un programme concurrent, pour écrire les informations ad hoc ne ralentit pas l'exécution du programme, contrairement à une vérification en parallèle de propriétés données. En outre, la conservation de cette base de données peut être un élément de justification lors de la certification d'un logiciel.

La figure 5.1 schématise le fonctionnement de la plate-forme d'analyse dynamique.

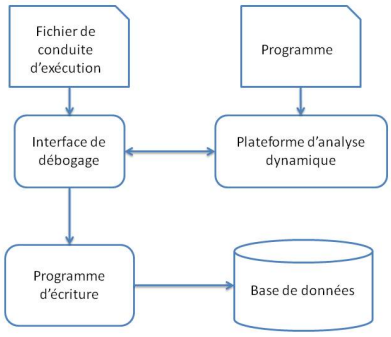

FIGURE 5.1 – Fonctionnement de la plate-forme d'analyse dynamique

Á cause du temps pendant lequel le programme est exécuté, les fonctions d'écoutes de variables, liées à l'interface de débogage, sont proscrites. En effet, une fonction d'écoute de variables donne à chaque pas d'exécution du programme la valeur de

celle-ci, ce qui peut complètement saturer la base de données utilisée.

La pose d'un nombre de points d'observation, qui fige le programme pour récupérer des données en mémoire, est, quant à elle, autorisée, tant que le nombre de points d'observation est limité.

L'exécution du programme est conduite depuis l'interface de débogage de façon interactive ou de façon automatique grâce à l'utilisation de fichiers de script contenant les instructions d'interruption et d'opérations à mener lors de celles-ci, telles que les informations à extraire de la mémoire du programme étudié.

Il faudra donc générer, de façon automatique, un script qui définit les points d'interruption et les commandes d'extraction nécessaires à la vérification d'une propriété donnée. L'analyse statique est capable de gérer la génération automatique d'un tel script.

5.1.2 Analyse statique de programmes avec Frama-C

5.1.2.1 Frama-C

La plate-forme d'analyse statique Frama-C [Cuoq 2012], dédiée aux programmes C, a été choisie parce qu'elle est déjà expérimentée chez Airbus pour vérifier des propriétés par analyse statique.

Frama-C est une plate-forme co-développée par le commissariat à l'énergie atomique (CEA) et l'Inria. Elle est dédiée à l'analyse de code source de programmes écrits en C. Elle possède une architecture modulaire autorisant la collaboration de plusieurs techniques d'analyse statique. Cette approche collaborative permet de combiner les informations issues d'un ou plusieurs greffons pour effectuer d'autres analyses. Il est également possible d'écrire en OCaml, en suivant un guide de développement, des greffons pour Frama-C.

Frama-C est capable de :

- déterminer un ensemble de valeurs possibles pour les variables du programme à chaque point d'exécution,
- simplifier un programme,
- naviguer dans le flot de données du programme, de la définition vers l'utilisation et inversement,
- prouver des propriétés formelles.

Le langage de spécification ACSL [Baudin 2008] est utilisé pour la preuve de propriétés. La spécification peut être partielle, concentrée sur un comportement particulier que doit avoir le programme vérifié. Cette analyse est basée sur le calcul de plus faible pré-condition de Hoare [Hoare 1969].

5.1.2.2 La bibliothèque CIL

La bibliothèque CIL [Necula 2002] (C Intermediate Language) permet d'analyser et transformer efficacement un programme C. Elle contient un parseur et un outil de vérification des types. Le programme C est transformé en un programme C normalisé ayant la même sémantique.

5.1.2.3 Le greffon Occurrence

Occurrence est un greffon de Frama-C dont l'objectif consiste à lister dans l'ensemble d'un programme C, les utilisations d'une variable donnée. Chaque point de programme, où cette variable est utilisée en lecture ou en écriture est répertorié.

Occurrence fait appel à un second greffon appelé Value Analysis. Ce dernier calcule par interprétation abstraite une sur-approximation du domaine des valeurs de la variable étudiée, à chaque instruction du programme. Value Analysis effectue donc une analyse sémantique du programme, et ce à partir d'un point d'entrée donné (par défaut la fonction main). Les alias d'une variable étudiée sont donc également pris en compte.

Le greffon Value Analysis a besoin soit du code source de chaque fonction appelée dans le programme, soit d'une définition simplifiée du fonctionnement de celle-ci. Les définitions des fonctions de librairies externes doivent donc également être fournies à Value Analysis.

5.2 Génération automatique de points d'observation

L'automatisation de la génération de points d'observation doit donc prendre en compte les contraintes de la plate-forme d'analyse dynamique et les possibilités de l'outil d'analyse statique Frama-C définies dans la section précédente.

L'utilisation de fonctions d'écoute de variables du programme à analyser est proscrite. Il faut donc définir une stratégie de récupération, par analyse statique, des points du programme où la valeur de chaque variable intervenant dans la propriété formelle est modifiée. Comme chaque modification de variable entraîne nécessairement la définition d'un point d'observation, et ralentit l'exécution du programme, il est vital de minimiser le nombre de points d'observation tout en ayant, à tout instant, la valeur de chaque variable intervenant dans la propriété.

5.2.1 Le cas d'une variable du programme

Proposition 1 *Pour connaître la valeur d'une variable à tout instant du programme, depuis sa création, la connaissance de ses valeurs à chaque fois que celle-ci est modifiée est nécessaire et suffisante.*

Preuve : Soit une variable x prenant ses valeurs dans le domaine X. La valeur d'une variable à l'instant t s'écrit $[\![x]\!]_t$. Soit B_x l'ensemble fini des points du programme P où la variable est modifiée.

Un point de programme correspond au produit d'une localisation de la variable dans le programme, par la valeur de la variable du programme, et par le temps où le programme atteint cette localisation.

On supposera que la variable x ne peut pas être modifiée à deux endroits du programme simultanément, dans le cas de programmes multitâches, car la mise en place de mécanismes de sémaphore doit empêcher ce phénomène. L'intervalle de

temps entre la capture de deux états n'étant pas constante, on se ramènera à une bijection du temps sur $\mathbb{N}$.

On ne trouvera donc jamais deux points d'observation ayant la même date. B_x est ordonné par le temps d'exécution. La fonction $\tau : B_x \mapsto \mathbb{N}$ renvoie la valeur du temps pour un élément de B_x donné.

La fonction $\mathcal{V} : B_x \mapsto X$ renvoie la valeur de la variable x au point donné.

Soit b_α et b_β deux points d'observation successifs de B_x.

Alors $\forall t \in \mathbb{N} \backslash\, \tau(b_\alpha) \leq t < \tau(b_\beta), [\![x]\!]_t = \mathcal{V}(b_\alpha)$ et si $t = \tau(b_\beta)$, alors $[\![x]\!]_t = \mathcal{V}(b_\beta)$.

La propriété étant vraie pour tout couple de points de B_x successifs, elle est donc vraie pour l'ensemble du programme, depuis la création de la variable x. On a donc démontré que si on connaît l'ensemble des points du programme où une variable est modifiée, alors on connaît la valeur de la variable à tout instant du programme.

Réciproquement, si on connaît la valeur de la variable à tout instant du programme, on peut reconstruire l'ensemble B_x. La construction d'une sur-approximation de B_x, en ajoutant des points de programme entre la création de la variable x et la fin du programme ne donnera aucune information supplémentaire ■.

Une variable de programme peut avoir un type primaire (entier, flottant booléen ...). Elle peut également avoir une structure plus complexe (record, record de record...). Par la suite, on considèrera les variables de type primaire. Pour un record, il faudra récuperer chaque variable primaire le composant.

Une variable de programme peut être modifiée de quatre manières différentes. Les points d'observation étant nécessaires et suffisants à chaque modification d'une variable, il est maintenant nécessaire d'identifier les différentes manières de modifier une variable, à savoir :

- lors de l'initialisation de la variable,
- par une affectation directe,
- par une affectation via un pointeur,
- par affectation dans un tableau.

Le greffon Occurrence de Frama-C détecte l'utilisation d'une variable pour les quatre cas. Un inconvénient d'Occurrence est qu'il ne fait pas de différence entre une utilisation d'une variable en lecture et une affectation d'une valeur à une variable. Pour minimiser le nombre de points d'observation, il est indispensable de faire cette différence en utilisant un filtre syntaxique, afin de limiter la taille de la trace d'exécution générée.

5.2.2 Définition d'un filtre syntaxique

Pour limiter la collecte des données à des points où une variable est modifiée, un filtre syntaxique a été mis en place. Ce filtre consiste à ne garder que les points de programme où une affectation de la variable étudiée est effectuée. Il ne s'agit en aucun cas d'une restriction de l'analyse à l'affectation directe de variable, mais bien d'une restriction postérieure à l'analyse sémantique du programme, qui conserve donc les affectations via un pointeur. Le schéma 5.2 positionne le filtre syntaxique dans l'analyse sémantique.

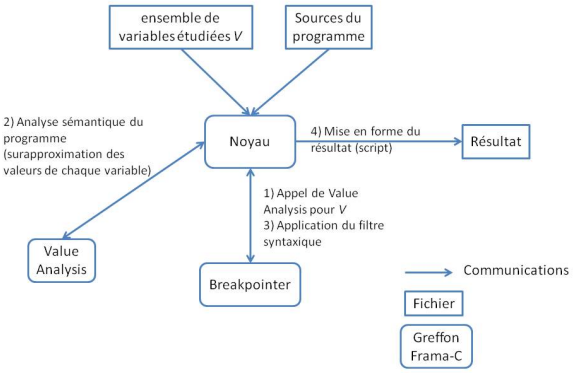

FIGURE 5.2 – Le filtre syntaxique

Au final, les points d'observation sont restreints à l'affectation d'une valeur à une variable, que ce soit par affectation directe ou via un pointeur.

La définition du filtre syntaxique repose sur l'utilisation de la bibliothèque CIL qui sert d'arbre syntaxique abstrait à Frama-C.

Une instruction C correspond au type *stmtkind* de la bibliothèque CIL. C'est un type somme OCaml qui peut correspondre à :

- Une instruction non composite (de type *instr*) qui peut être :
  - une affectation
  - un appel de fonction
  - une ligne d'instruction assembleur
  - un skip
- un return
- un Goto
- un Break
- un If then else
- un Switch
- une boucle (Loop)
- un block d'instructions
- une séquence non spécifiée par la norme ISO/C
- un TryFinally
- un TryExcept

Pour chacun des éléments a ci-dessus, le filtre va collecter le point d'observation ou non et autoriser ou empêcher l'analyse d'une partie ou de tout élément composant a.

Le comportement du filtre pour chaque *stmtkind* est défini dans le tableau 5.1 et pour chaque *instr* dans le tableau 5.2. Ainsi, pour chaque élément, la collecte du point d'observation se fait ou non et le sous-arbre de cette structure est parcouru ou non.

Element	Collecte	Analyse de ses éléments
Instruction	Voir tableau 5.2	Voir tableau 5.2
Return	non	non
Goto	non	non
Break	non	non
Continue	non	non
If a then b else c	non pour a	analyse de b et c
Switch	non	oui
Loop	oui	oui
Block	oui	oui
Séquence non spécifiée	oui	oui
TryFinally	oui	oui
TryExcept	oui	oui

TABLEAU 5.1 – Comportement du filtre pour une structure de codage

Element	Collecte	Analyse de ses éléments
Affectation	oui	non
Appel de fonction	nom de la fonction	oui
Assembleur	non	non
Skip	non	non
Annotation	non	non

TABLEAU 5.2 – Comportement du filtre pour une instruction

Exemple 24 *Soit l'exemple suivant, issu du manuel utilisateur de Frama-C [Stous 2013].*

```
1  int rr=1;
2
3  int opa(int r) {
4    return r+1;
5  }
6
7  void opb () {
8    if(rr<4998) {
9      rr+=2;
10   }
11 }
12
13 void opc () {
14   rr=600;
15 }
16
17 int main() {
18   if (rr<5000) {
```

```
19        rr=opa(rr);
20      }
21      opb();
22      goto L6;
23      opc();
24    L6:
25      return 1;
26  }
```

Supposons dans 24 qu'on souhaite étudier le comportement de la variable *rr*. Sans le filtre, un point d'observation est défini aux lignes 8,9,18 et 19. Aucun point d'observation n'est défini à la ligne 14, car l'analyse sémantique du programme montre que c'est une ligne de code qui n'est pas exécutée. En effet, la fonction *opc* n'est pas appelée : c'est du code mort, puisque que l'instruction goto fait sauter la ligne où l'appel de *opc* a lieu. Avec la mise en place du filtre, seules les lignes 9 et 19 seront retenues, car il s'agit d'affectations de valeur à *rr*.
Les propriétés portant sur des variables du programme sont courantes, cependant il existe également des propriétés portant sur des appels de fonction.

5.2.3 Appels de fonction

À l'aide de l'interface de débogage, il est possible de définir une variable qui n'appartient pas au programme pour enregistrer un appel de fonction. Ce sont les variables fantômes (ou variables ghost). Elles ne sont utiles que pour la spécification et la vérification d'une propriété et ne modifient pas le programme. C'est une technique non intrusive qui se retrouve fréquemment en analyse statique de programme, lors de preuves unitaires ou d'intégration.

La modélisation par une variable d'un appel de fonction étant définie, il faut déterminer les points de programme où une fonction est appelée. Pour ce faire, une analyse syntaxique du programme est suffisante. En effet, il n'y a pas de pointeurs associés à des fonctions dans les logiciels embarqués.

Par ailleurs, il peut parfois s'avérer nécessaire de poser un point d'observation avant ou après une fonction. Prenons l'exemple de la fonction qui effectue une requête de prise de sémaphore.

Pour s'assurer de l'absence de famine, il est nécessaire de savoir quand le programme entre en section critique et quand il en sort. Dans ce cas, il sera nécessaire de poser un point d'observation juste avant l'appel de la fonction et un juste après l'appel.

Il est donc intéressant de pouvoir poser de façon automatique des points d'observation relatifs par rapport à l'appel de la fonction, car il devient possible de travailler finement avec ces appels de fonction.

Exemple 25 *Soit la fonction C suivante :*

```
1  static void * serveur (void * p_data)
2  {
3    while (1)
4      {
5        psleep (get_random (2));
6        compteur_lock=compteur_lock+1;
7        pthread_mutex_lock (& mutex);
8        ressource=ressource+1;
9        pthread_mutex_unlock (& mutex);
10       compteur_unlock=compteur_unlock+1;
11       psleep (get_random (2));
12     }
13 }
```

Dans l'exemple 25, pour surveiller l'absence de famine dans le cadre d'une modification de *ressource*, il faut savoir quand la fonction *pthread_mutex_lock* est appelée et quand le programme entre en section critique, c'est-à-dire lorsque la fonction *pthread_mutex_lock* a fini de s'exécuter.

Il faut donc poser un point d'observation avant l'appel de *pthread_mutex_lock*, soit à la ligne 7 (la plate-forme d'analyse dynamique s'arrête en début de ligne d'instruction) et un autre point une ligne après, soit à la ligne 8.

Pour ces deux points d'observation, une variable entière ghost (définie pour des besoins de spécification) *acces_ressource_client* qui prend les valeurs 1 (prise de mutex), 2 (entrée en section critique) modélise les appels des deux fonctions. Un point d'observation est posé en fin de section critique (une fois que la fonction *pthread_mutex_unlock* a été exécutée), soit à la ligne 11, pour que la variable *acces_ressource_client* prenne la valeur neutre 0. Cela permet, en outre, d'avoir un marqueur pour la fin de la section critique.

Trois points d'observation relatifs ont donc été définis par rapport aux appels des fonctions *pthread_mutex_lock* et *pthread_mutex_unlock*. L'utilisation de l'analyse statique pour poser des points d'observation relatifs par rapport aux appels de fonction augmente la sûreté de l'analyse, puisqu'aucun point d'observation ne sera oublié.

Cette approche d'automatisation définie, il est indispensable de s'assurer de la correction de cette approche.

5.3 Correction sous hypothèses

Le fait d'utiliser un outil d'analyse statique adapté à des programmes monotâches et de ne collecter qu'un nombre minimal de points d'observation nécessite de s'assurer de la correction de la démarche utilisée, et d'en expliciter les hypothèses. Notamment, il faut d'une part caractériser les limites d'utilisation de cette méthode pour des programmes multitâches. D'autre part, il faut s'assurer de la correction

de la vérification de la propriété $\phi(x_0, .., x_n)$ à l'aide d'une trace où les valeurs des opérandes $x_0, ..., x_n$ ne sont collectées que lorsque celles-ci sont modifiées.

5.3.1 Cas des programmes multitâches

Value Analysis n'effectue pas une analyse de programmes multitâches. Cependant, il existe une possibilité de contourner partiellement ce problème. Value analysis nécessite une fonction comme point d'entrée pour faire son évaluation. Plutôt que d'effectuer l'analyse sur la fonction d'entrée du programme (la fonction main, en général), il est possible de lancer l'analyse pour chaque fonction principale de tâche.

Value Analysis effectue son analyse sur le code source d'un programme et de toutes les bibliothèques utilisées par ce programme. Il faut donc disposer du code source des bibliothèques multitâches ou d'une modélisation de celles ci, interprétable par Frama-C. On peut se limiter à une partie de chaque librairie utilisée.

Exemple 26 *Supposons que le programme $\mathcal{P}$ appelle la fonction $\mathcal{F}$ de la librairie $\mathcal{L}$. Il faut donc avoir le code source ou une modélisation de $\mathcal{F}$ et de tout l'arbre d'appel des fonctions utilisées par $\mathcal{F}$.*

Un autre inconvénient de cette méthode est qu'il n'est pas possible d'effectuer une analyse de pointeurs croisés.

Exemple 27 *Soit la fonction mainA, fonction principale de la tâche A et mainB fonction principale de la tâche B. Soit p un pointeur local à mainA et v une variable locale à mainB. Si p pointe sur v, alors une modification de v à partir de mainA est possible, mais n'est pas détectable en effectuant une analyse statique séparée de chaque fonction principale de tâche.*

Hypothèse 1 *Dans le programme multitâche à vérifier, il n'y a pas d'alias croisés.*

Pour effectuer une vérification sur un programme multitâche, il est également indispensable de savoir quelle tâche a généré un état. En pratique la collecte du numéro de la tâche lors de la définition d'un point d'observation suffit.

Hypothèse 2 *Le contexte de la tâche est défini pour chaque point d'observation.*

À noter que l'équipe de Frama-C évalue un plugin appelé Mthread, qui a pour vocation de faire une analyse sémantique de programmes multitâches.

5.3.2 Correction de la vérification

La correction de la vérification se divise en deux sous-problèmes. Il faut d'une part, prendre en compte le niveau d'extraction de la valeur d'une variable avec le niveau d'expression d'une propriété temporelle.

D'autre part, il faut s'assurer, compte tenu du niveau choisi, de la correction de la vérification de la propriété.

5.3.2.1 Les différents niveaux

La figure 5.3 présente les différents niveaux de visibilité d'exécution du programme.

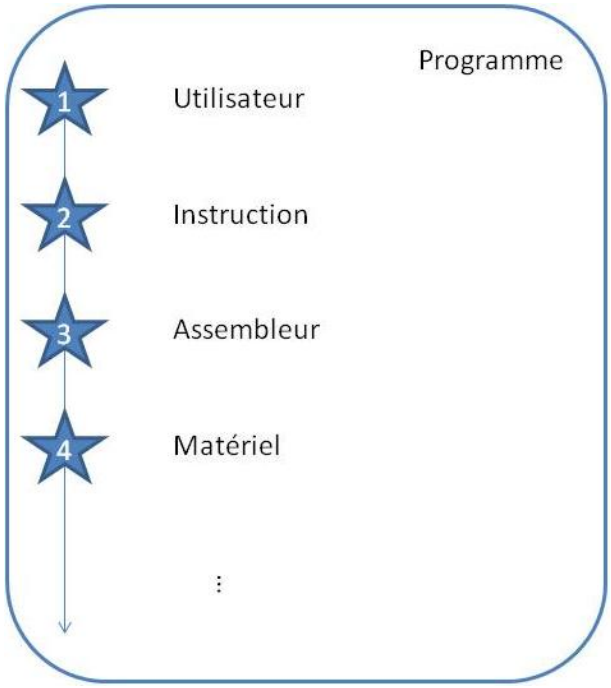

FIGURE 5.3 – Les niveaux de visibilité d'exécution du programme

Le plus haut niveau (1) est celui de l'utilisateur. Il démarre un programme avec des entrées données et s'attend à un service donné. Le niveau suivant correspond au niveau instruction du langage utilisé, ici le C. Pour chaque instruction C, il y a un paquet d'instructions assembleur, ce qui constitue le niveau 3. Le niveau 4 est le niveau matériel.

Les propriétés sont exprimées au niveau des instructions C. L'extraction des variables s'effectue, quant à elle, au niveau assembleur.

Hypothèse 3 *Pour que l'extraction des variables effectuée au niveau trois soit correcte, il est nécessaire et suffisant d'avoir une équivalence sémantique entre le niveau 2 (code source) et le niveau 3 (assembleur).*

Autrement dit, les opérations d'optimisation de compilateur ne doivent pas changer la sémantique opérationnelle du programme et donc nuire à la vérification d'une propriété donnée.

Exemple 28 *Soit le programme suivant :*

```
1 #include <stdio.h>
2 #include <stdlib.h>
```

```
3  int main (void)
4  {
5     int donnee=0;
6     donnee=5;
7     donnee=10;
8     printf("donnee=%d!\n",donnee);
9     return 0;
10 }
```

Lors de la compilation, des algorithmes d'optimisation suppriment des instructions inutiles telles que la ligne 6 du programme précédent. De ce fait, la propriété "donnee ne vaut jamais 5" est évaluée à vraie pour un programme optimisé, alors qu'elle est évaluée à fausse pour un programme non optimisé. La correction de notre approche nécessite de vérifier la propriété sur un programme non optimisé.

Dans la suite, on se placera au niveau de l'instruction C.

5.3.2.2 Correction de la vérification sous hypothèse

Il est nécessaire de s'assurer de la correction de la vérification d'une propriété sur une trace d'exécution dont les états sont cadencés non pas à chaque instruction de code C, mais bien à chaque point d'observation. Il faut donc démontrer que les propriétés vérifiées sur la trace d'exécution partielle définie par les points d'observation sont également forcément vérifiées sur la trace d'exécution complète du programme, et réciproquement.

Cette démonstration s'appuie sur l'utilisation de différentes traces théoriques définies par des transformations mathématiques. Un raisonnement par équivalence de vérification d'une trace à une autre est effectué.

Le tic d'horloge se définit comme le passage entre deux instructions C successives.

Soit un programme P. Soit V_p l'ensemble des variables de P. Soit E_P une exécution de P.

Soit $\Delta_{E_P} = [\![0; t_E]\!]$ l'ensemble des tics d'une exécution, majoré par le dernier tic t_E. $M_{E_P}(i)$ est l'état mémoire du programme au tic i.

Une trace d'exécution est une séquence d'états σ_i référencés par le tic i où Π définit la séquence ($\prod_{i=0}^{n} \sigma_i = \sigma_0\sigma_1...\sigma_n$). Un état est assimilé à un état mémoire du programme ou une projection de cet état mémoire sur un sous ensemble des variables du programme.

Dans la suite, toute trace d'exécution théorique, servant de support au raisonnement sera notée τ. Soit τ_{E_P} la trace d'exécution théorique dont les états sont cadencés à chaque tic d'horloge.

L'égalité $\tau_{E_P} = \prod_{\Delta_{E_P}} M_{E_P}(i)$ effectue le lien entre une trace et les états mémoires du programme.

Soit B_P l'ensemble des points d'observation. Soit N le nombre de points d'observation activés durant l'exécution E_P. Soit une formule LTL $\Phi(y_0, ..., y_K)$ utilisant $K+1$ variables indépendantes de V_p, $y_0, ..., y_K$. Pour simplifier les notations, on

notera $\vec{Y} = (y_0, ..., y_K)$. Soit T_{E_P} la trace récupérée, pour une exécution E_P et l'ensemble des points d'observation B_P donné.

$T_{E_P} = \prod_{i=0}^{N} M_{E_P}(i)_{|\vec{Y}}$ définit le lien entre la trace récupérée T_{E_P} et la projection $M_{E_P}(i)_{|\vec{Y}}$ des états mémoires $M_{E_P}(i)$ sur les variables du vecteur $\vec{Y}$.

Le but est de montrer que vérifier ϕ sur τ_{E_P} est équivalent à montrer ϕ sur T_{E_P}.

Les deux transformations suivantes sont définies à partir de τ_{E_P} :

- Projection 1 : les variables inutiles à la vérification de la propriété considérée sont supprimées. Le nombre d'états reste le même. Cette projection prend τ_{E_P} en entrée et génère $\tau_{E_{P_1}}$
- Projection 2 : cette projection réduit le nombre d'états. Elle prend $\tau_{E_{P_1}}$ en entrée et génère $\tau_{E_{P_2}}$

La figure 5.4 schématise les différentes traces utilisées pour la démonstration

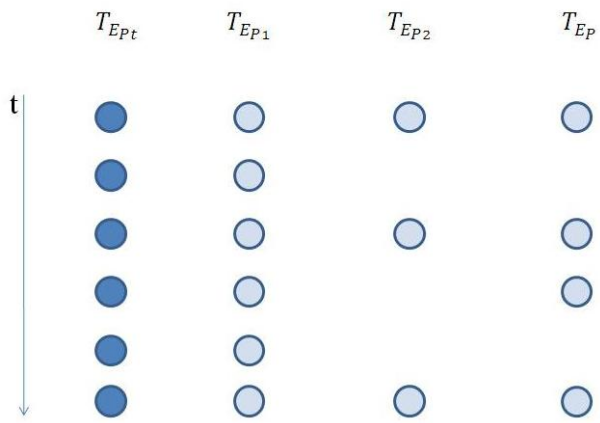

FIGURE 5.4 – Lien entre les traces théoriques et la trace récupérée

Soit p_1 la première projection. Ainsi :

$$\tau_{E_{P_1}} = p_1(\prod_{\Delta_{E_P}} M_{E_P}(i)) = \prod_{\Delta_{E_P}} M_{E_P}(i)_{|\vec{Y}} \tag{5.1}$$

Soit p_2 la seconde projection. Alors :

$$\tau_{E_{P_2}} = p_2(\prod_{\Delta_{E_P}} M_{E_P}(i)_{|\vec{Y}}) = \prod_{\Delta_{E_{P2}}} M_{E_P}(i)_{|\vec{Y}} \tag{5.2}$$

Pour effectuer cette projection, lorsque n états de tic consécutifs sont identiques, à savoir pour i, $i+1$ ... $i+n-1$ $\in \Delta_{E_P}, \forall j \in [\![0, K]\!], x_{j,i} = x_{j,i+1}$, l'état i sera conservé et les états $i+1$ à $i+n-1$ seront supprimés. Cet algorithme est appliqué pour tout i jusqu'à ne plus trouver de suite d'états de tic consécutifs égaux. $\Delta_{E_{P2}}$ sera l'ensemble des tics cadençant $\tau_{E_{P2}}$.

La projection 2 vérifie la propriété suivante : $\forall i \in \Delta_{E_{P2}}, \exists j \in [\![0, K]\!], y_{j,i} \neq y_{j,i+1}$

Définition 17 (Chemins stutter-équivalents) *[Markey 2003] Soient π et π' deux chemins. Ces deux chemins sont dits stutter-équivalents s'il existe deux suites infinies strictement croissantes (i_k) et (j_k), vérifiant $i_0 = j_0$, et telles que pour tout entier k, les états*
$\pi_{i_k},\ \pi_{i_k+1}\ ...,\ \pi_{i_{k+1}-1}, \pi'_{j_k},\ \pi'_{j_k+1}...,\ \pi'_{j_{k+1}-1}$ *sont identiques.*

Définition 18 (Formule stutter-invariante) *[Markey 2003] Une formule ϕ de PLTL est stutter-invariante si elle ne peut pas distinguer deux chemins stutter-équivalents, c'est-à-dire que, si π et π' sont deux chemins stutter-équivalents, et si (i_k) et (j_k) sont deux suites qui témoignent de la stutter-équivalence de π et π', alors pour tout k, $\pi, i_k \vDash \phi \Longleftrightarrow \pi', j_k \vDash \phi$.*

Définition 19 (Fragment $L(\mathcal{U})$) *[Markey 2003] Le fragment $L(\mathcal{U})$ est la logique temporelle définie par les opérateurs $\{\neg, \vee, \wedge, \mathcal{U}\}$, et les opérateurs temporels pouvant être définis à partir de $\mathcal{U}$.*

$\mathcal{U}$ est l'opérateur binaire telle que : $\phi\,\mathcal{U}\,\psi$ signifie que la propriété ϕ est vraie jusqu'à ce que la propriété ψ devienne vraie.

Définition 20 (Formules stutter-invariantes de LTL) *[Markey 2003] Une formule de PLTL est stutter-invariante si, et seulement si, elle peut être exprimée dans le fragment $L(\mathcal{U})$.*

Proposition 2 *Vérifier une formule sur $\tau_{E_{P2}}$ est équivalent à vérifier cette formule sur $\tau_{E_{P1}}$.*

Preuve : $\tau_{E_{P2}}$ et $\tau_{E_{P1}}$ sont des chemins stutter-équivalents, à condition de travailler dans le fragment $L(\mathcal{U})$ ou $L(\mathcal{U}, \mathcal{S})$[Markey 2003] Nous avons donc pour tout k, $\Pi, i_k \vDash \phi \Leftrightarrow \Pi', j_k \vDash \phi$, en se restreignant à $L(\mathcal{U})$ ■.

Proposition 3 *Vérifier une formule sur τ_E est équivalent à vérifier cette formule sur $\tau_{E_{P1}}$:*

Preuve : La projection 1 conserve le nombre d'états et garantit également la propriété : $\forall i \in [\![0, K]\!], y_i \in V_P$.

Les variables dans $V_P \backslash \vec{Y}$ sont indépendantes de $\vec{Y}$. La projection 1 consiste à ne pas prendre en compte $V_P \backslash \vec{Y}$ dans les états de $\tau_{E_{P_1}}$ par rapport à τ_E. Ces variables n'intervenant pas dans la propriété temporelle à vérifier, l'équivalence est donc vérifiée.

À condition de travailler dans le fragment $L(\mathcal{U})$, la vérification de propriétés temporelle est donc correcte ■.

Proposition 4 *Vérifier une formule sur $\tau_{E_{P_2}}$ est équivalent à vérifier cette formule sur T_{E_P}.*

Preuve : Les deux traces diffèrent en ce que T_{E_P} peut avoir des états successifs égaux. Les deux traces sont de ce fait stutter-équivalentes ■.

Exemple 29

```
int entier=0;
for (int i=0; i<100000;i++){
        entier=10;}
```

En posant un point d'observation à la dernière ligne, la trace T_{E_P} aura 100000 *états identiques mais la trace $T_{E_{P_2}}$ aura un seul état.*

Le cas des expressions régulières n'a pas été formellement démontré. Cependant, une expression régulière pouvant s'écrire sous la forme d'une formule LTL, il est possible d'en déduire que :

- les séquences faibles et étendues sont des éléments du fragment $\mathcal{L}(\mathcal{U})$, car elles s'écrivent uniquement à l'aide des opérateurs temporels $\mathcal{U}$,$\Diamond$ et $\Box$
- les séquences fortes nécessitent l'utilisateur de l'opérateur $\circ$, elles sont donc exclues du fragment $\mathcal{L}(\mathcal{U})$

Conclusion À condition de travailler dans le fragment $L(\mathcal{U})$, la vérification de propriétés temporelles sur la trace issue de l'exécution est bien correcte.

Conclusion du chapitre

L'objectif principal de ce chapitre était de définir une approche pour générer automatiquement une trace d'exécution pour une propriété à vérifier dans un programme donné. Pour ce faire, les contraintes industrielles concernant les analyses statiques et dynamiques de programme ont été identifiées. À partir de ces contraintes, une approche qui minimise la taille de la trace d'exécution, en fonction de la propriété à vérifier a été définie. Cette approche a finalement été validée, à condition de travailler dans un fragment de la logique temporelle linéaire qui n'accepte pas des propriétés faisant intervenir l'opérateur suivant ($\circ$) ou des séquences fortes. Les limites concernant les programmes multitâches ont également été fixées, à savoir :

- il ne doit pas y avoir d'alias croisés,
- le contexte de la tâche est défini pour chaque point d'observation,
- les opérations d'optimisation du compilateur ne doivent pas changer la sémantique opérationnelle du programme.

Chapitre 6

Vérification des traces d'exécution

Sommaire

L'objectif de ce chapitre est de définir l'approche permettant de vérifier des traces d'exécution de programme construite en adéquation avec les hypothèses de la section 5.3. Toute trace respectant les hypothèses de 5.3 est donc vérifiable par l'approche décrite dans ce chapitre.

La section 6.1 abordera les algorithmes et méthodes utilisées pour vérifier une propriété temporelle, sur des traces infinies. La section 6.2 définira deux méthodes de vérification différentes et la méthode la plus efficace sera déterminée. La section 6.3 identifiera les différentes étapes de la méthode utilisée pour vérifier une propriété temporelle sur une trace d'exécution. La section 6.4 abordera le problème de fin de trace, et explicitera les choix effectués pour pallier ce problème.

6.1 Vérification de propriétés temporelles

La vérification de propriétés de logique temporelle linéaire par model checking fait appel aux automates.

Dans un premier temps, il faut rappeler la définition d'un automate à états. Classiquement, un automate à états n'accepte que des mots de longueur finie. Dans un deuxième temps, puisque LTL a une sémantique sur des mots de longueur infinie, les automates de Büchi, qui eux acceptent des mots infinis, sont définis à partir de la définition d'un automate à états. Dans un dernier temps, l'algorithme de vérification d'une propriété LTL à partir des automates Büchi sera rappelé.

6.1.1 Les automates

Définition 21 (Automate) *Un automate est un quintuplet* $A = (Q, \Sigma, \rightarrow, q_0, F)$ *avec :*

- Q *un ensemble d'états*
- Σ *un alphabet*
- $\rightarrow \subseteq Q \times \Sigma \times Q$ *une relation de transition*
- $q_0 \in Q$ *un état initial*
- $F \subseteq Q$ *un ensemble d'états finaux.*

Définition 22 (Condition d'acceptation d'un mot fini) *Un mot* $w \in \Sigma^*$ *de longueur* n *est un mot de* $\mathcal{L}(A)$*, où* A *est un automate, si et seulement s'il existe une suite* $(q)_{i,i\in[\![0;n]\!]}$ *commençant à* q_0*, telle que pour tout* $i \in [\![0; n-1]\!]$*,* $(q_i, w_i, q_{i+1}) \in \rightarrow$ *et* $q_n \in F$*.*

Un automate A reconnaît le langage régulier $\mathcal{L}(A)$ [Kleene 1956] défini sur l'alphabet Σ. Tous les mots de $\mathcal{L}(A)$ sont des mots finis.

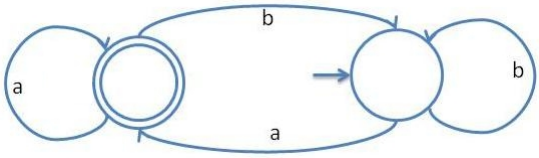

FIGURE 6.1 – Automate reconnaissant les mots de l'alphabet $\{a, b\}$ finissant par a

Exemple 30 *L'automate 6.1 accepte l'ensemble des mots formés des lettres a et b terminant par a. Les mots "abababa" et "aaaabba" sont des mots de* $\mathcal{L}(A)$*. Les mots "bbbbbab" et "ab" ne sont pas des mots de* $\mathcal{L}(A)$*.*

6.1.2 Les automates de Büchi

Un automate de Büchi est un automate dont les conditions d'acceptation d'un mot ont été modifiées pour accepter un mot infini. La définition 21 reste valide

pour les automates de Büchi. La définition 22, quant à elle, est remplacée par la définition 23.

Définition 23 (Condition d'acceptation d'un mot infini) *Un mot $w \in \Sigma^\omega$ est un mot de $\mathcal{L}(B)$, où B est un automate de Büchi, s'il existe une suite $(q)_{i,i\in\mathbb{N}}$ commençant à q_0, telle que pour tout $i \in \mathbb{N}$, $(q_i, w_i, q_{i+1}) \in \rightarrow$ et pour tout $j \in \mathbb{N}$ il existe $k \in \mathbb{N}$ tel que $k > j$ et $q_k \in F$.*

Exemple 31 *Avec l'automate 6.1, s'il s'agit d'un automate de Büchi alors il accepte les mots contenant une infinité de a.*

6.1.3 Le model checking classique

Pour vérifier une propriété LTL sur un programme par model checking, il faut d'abord modéliser le comportement du programme par un modèle $\mathcal{M}_P$. Le model checking d'une formule LTL ϕ sur un modèle $\mathcal{M}_P$ du programme P se compose de quatre étapes successives [Schnoebelen 1999] :

1. Transformation de ϕ en automate $\mathcal{B}_{\neg\phi}$.
2. Transformation de $\mathcal{M}_P$ en automate de Büchi $\mathcal{B}_{\mathcal{M}_P}$
3. Calcul de l'automate $\mathcal{B}_\otimes$, produit synchronisé de $\mathcal{B}_{\neg\phi}$ et $\mathcal{B}_{\mathcal{M}_P}$, reconnaissant le langage $\mathcal{L}(\mathcal{B}_{\neg\phi}) \bigcap \mathcal{L}(\mathcal{B}_{\mathcal{M}_P})$
4. Vérification que le langage reconnu par $\mathcal{B}_\otimes$ est vide ou non. $\mathcal{M}_P \vDash \phi$ si et seulement si $\mathcal{L}(\mathcal{B}_\otimes) = \emptyset$

Il faut maintenant choisir, en s'aidant des définitions et approches de vérification définies ci-dessus, la méthode de vérification la plus efficace et adaptée pour vérifier une propriété temporelle dans le contexte de cette thèse.

6.2 Deux méthodes pour vérifier une propriété LTL

Pour vérifier une propriété temporelle, sur un programme, plusieurs classes de méthodes existent. Le model checking, qui a pour but de vérifier une propriété pour toutes les exécutions possibles d'un programme complet, est l'une d'entre elles. La Runtime verification, quant à elle, a pour but de vérifier une propriété sur une exécution du programme [1].

Pour cette seconde classe, une méthode consiste à réécrire la propriété temporelle grâce à une équation de point fixe. D'autres méthodes consistent à transformer la propriété temporelle en automate de Büchi déterministe et à exécuter cet automate soit pendant l'exécution du programme, soit sur une trace d'exécution a posteriori. La transformation d'un automate non déterministe en un automate déterministe est cependant exponentielle [Sakarovitch 2003].

Deux méthodes ont été étudiées pour vérifier une propriété temporelle sur une trace d'exécution. Elles consistent à :

1. Voir le chapitre 2 pour plus d'informations sur les différents travaux.

- transformer la trace en automate avec bouclage sur le dernier état de la trace et vérifier la propriété en utilisant la démarche du model checking classique, en s'appuyant sur la démarche de 6.1.3. Il ne s'agit cependant pas de model checking, puisque la propriété est vérifiée non pas sur le programme, mais sur une trace d'exécution du programme. Le modèle $\mathcal{M}_P$ est remplacé par la trace d'exécution.
- transformer la formule LTL en automate de Büchi et vérifier la propriété en exécutant l'automate avec la trace.

Pour choisir la démarche la plus appropriée, des études de performance ont été réalisées sur un outil existant, NuSMV [Cimatti 2000]. Comme la plate-forme d'analyse dynamique peut générer des traces de taille importante (de l'ordre de 10^9 états), la démarche adoptée doit pouvoir traiter des traces de cet ordre de grandeur sans débordement de mémoire. Le temps de vérification doit être également le plus court possible.

6.2.1 Le model checker NuSMV

L'outil NuSMV a été choisi. En effet, d'une part il est en mesure de vérifier une propriété par model checking classique, d'autre part il peut lire une trace d'exécution et exécuter un automate de Büchi avec cette trace en entrée, ce qui permet d'essayer les deux méthodes de vérification envisagées. En outre, il peut travailler avec des propriétés LTL sur des variables numériques.

Pour la première approche, l'outil a besoin d'un fichier contenant le modèle à vérifier. Ici, le modèle est remplacé par une trace d'exécution transformée en automate. Chaque état de la trace contient les valeurs des variables intervenant dans la propriété à vérifier. Un état est relié à son successeur par une transition 1. L'outil a également besoin d'une propriété LTL.

Pour la seconde approche, l'outil nécessite un fichier contenant l'automate obtenu à partir d'une propriété temporellle, et un fichier XML contenant la trace d'exécution.

6.2.2 Générateur de trace

Pour chacune des deux approches, un générateur de trace a été implémenté. Le format de sortie diffère selon la méthode employée. Cependant, les caractéristiques de la trace sont les mêmes, dans le but de comparer les deux approches.

Ce générateur produit des traces de taille 10^N, où N est un paramètre d'entrée du programme. Des essais de performance sont menés pour $N \in [\![0, 9]\!]$. Chaque trace contient une unique variable booléenne B. Soit b_i la valeur de la variable B à l'état i, alors la valeur de B sur toute la trace a été définie récursivement comme :

$$b_0 = true \tag{6.1}$$

$$\forall n \in [\![1, 10^N]\!], b_n = \neg\, b_{n-1} \tag{6.2}$$

La propriété à vérifier est également la même pour les deux approches :

$$\Box\,(B \Rightarrow \Diamond\,(\neg\,B)) \tag{6.3}$$

6.2.3 Première approche : vérification par Model checking

Cette première approche consiste à transformer la trace en automate et à utiliser un algorithme de model checking classique (confer 6.1.3) pour vérifier une propriété temporelle ϕ donnée.

La figure 6.2 présente les résultats de cette approche. On constate qu'à partir de 10^5 états, il n'est plus possible de vérifier la trace. Un message d'erreur de NuSMV, *segmentation fault*, apparaît. Ce message correspond à une surcharge mémoire vive de la machine par le logiciel. En d'autres mots, NuSMV demande plus de ressource mémoire que n'est capable d'en donner la machine. Cette surcharge provient du produit synchronisé effectué entre la trace d'exécution transformée en automate et la formule temporelle transformée en automate de Büchi.

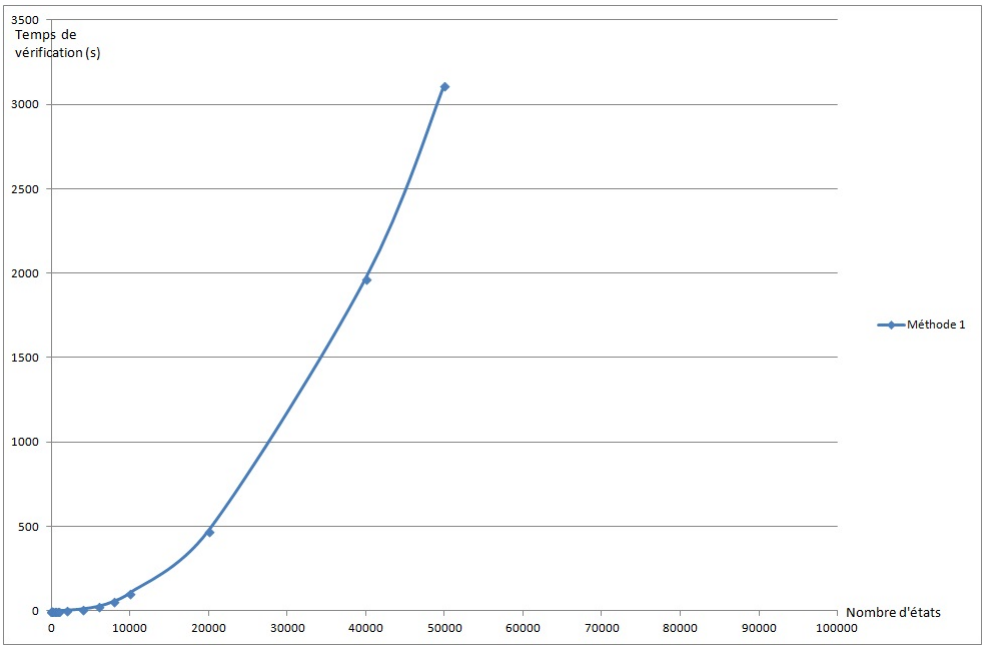

FIGURE 6.2 – Première approche : Transformation de la trace en automate

Cette approche ne permet pas de vérifier des propriétés LTL sur des traces de plus de 10^4 états avec la configuration de la machine donnée (Erreur de mémoire "Segmentation fault" au dessus de 5.10^4 états.). De plus, même si une machine ayant une meilleure configuration est utilisée, le temps de vérification explose clairement : entre 10^2 et 10^3 états, le temps de calcul est multiplié par 20 et entre 10^3 et 10^4 états, il est multiplié par 100. Cette approche n'est donc pas viable.

6.2.4 Seconde approche : vérification par exécution d'automate de Büchi

Cette seconde approche consiste dans un premier temps à transformer la formule à évaluer en automate de Büchi. Dans un second temps, l'automate de Büchi est exécuté avec la trace à vérifier en guise d'entrée.

6.2.4.1 Transformation d'une propriété LTL avec Ltl2ba

La transformation d'une propriété temporelle en automate de Büchi produit l'automate décrit par la figure 6.3. Il a été généré à partir de l'outil Ltl2ba [Gastin 2001]

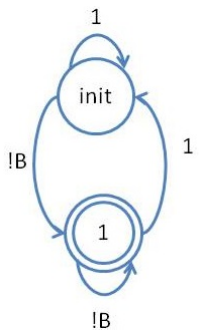

FIGURE 6.3 – Automate de Büchi de la propriété $\Box\,(B \Rightarrow \Diamond\,(\neg\,B))$

qui est capable de transformer automatiquement une propriété temporelle en automate de Büchi. Or, la complexité de l'algorithme de transformation de propriété temporelle en automate de Büchi est au pire exponentielle en fonction de l'imbrication des opérateurs temporels, d'après [Oddoux 2003]. Malgré cela, dans la plupart des cas rencontrés (chapitre 4), le temps de transformation ne sera pas un élément limitant, puisque mis à part le cas des séquences, les formules rencontrées sont de petites tailles. Le cas des séquences est traité différemment dans 6.3.4.

Les opérateurs acceptés par Ltl2ba pour les formules LTL sont décrits dans le tableau 6.1. Ltl2ba n'accepte donc que les opérateurs du futur.

LTL	Ltl2ba	LTL	Ltl2ba
$\Box$	G	$\neg$	!
$\Diamond$	F	$\Rightarrow$	->
$\mathcal{U}$	U	$\wedge$	&&
$\circ$	X	$\vee$	||

TABLEAU 6.1 – Correspondance entre les opérateurs LTL et Ltl2ba

Le langage de sortie est Promela [Gerth 1997]. Enfin, les automates de Büchi générés ne sont pas déterministes. Pour les essais de performance, la traduction entre Promela et le langage de NuSMV est faite manuellement.

6.2.4.2 Résultats de la seconde approche

Le tableau 6.4 synthétise les résultats de cette seconde approche et les compare avec l'approche précédente.

Pour des petites traces, la première approche est plus efficace. Mais, d'une part le temps de vérification de l'approche précédente croît exponentiellement, pour cette formule, avec le nombre d'états dans la trace, et d'autre part, passé un certain nombre d'états dans la trace, la mémoire est surchargée. La transformation de propriétés temporelles en automates de Büchi a une complexité plus qu'exponentielle avec la profondeur de la formule à transformer (imbrication successive des opérateurs temporels). Cependant pour les formules rencontrées, le temps de transformation est relativement court. Dans cet essai, il est de 0.26 secondes.

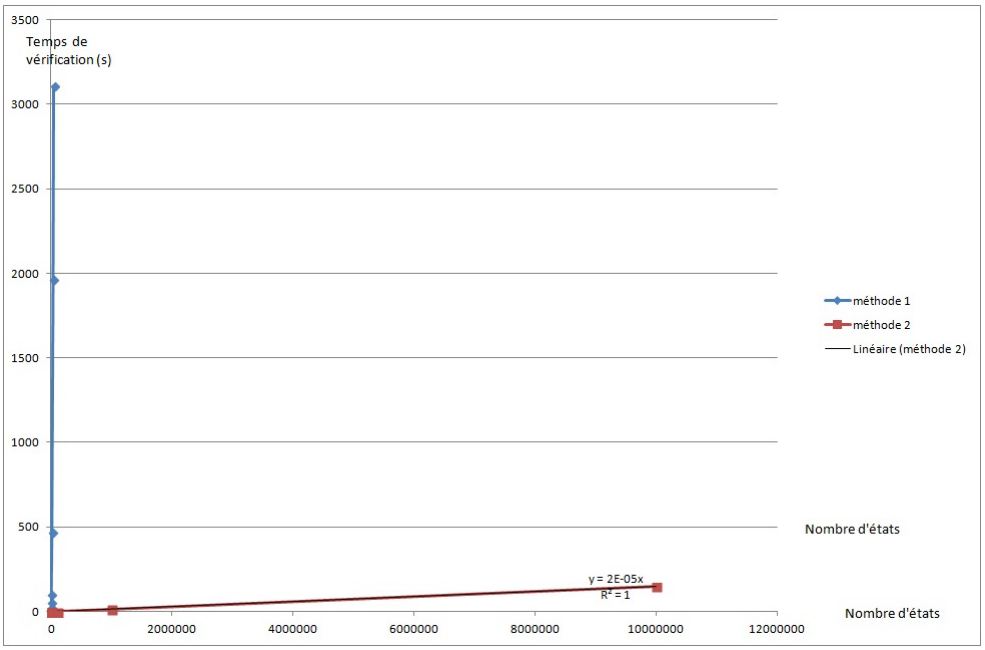

FIGURE 6.4 – Synthèse des deux approches

Au vu des résultats, cette approche est la plus prometteuse pour vérifier des propriétés sur des traces de grande taille (de l'ordre de 10^8 états), puisqu'il est possible de vérifier des propriétés sur des traces de grande taille d'une part et le temps de calcul, sur cet exemple, est linéaire avec la taille de la trace d'autre part [2]. C'est cette approche qui est donc choisie.

6.3 Vérification par exécution de l'automate de Büchi

NuSMV a permis de choisir l'approche utilisée par la suite pour vérifier une propriété temporelle. Cependant, cet outil ne permet pas de gérer finement la fin

2. Une analyse de complexité est menée en 6.3.2.

de trace 6.4. En outre, l'utilisation de NuSMV nécessite la définition de traducteurs pour traduire un automate Promela en automate compatible pour NuSMV. De même, il est nécessaire d'avoir un traducteur de trace d'exécution depuis les différents formats de traces vers le format XML lu par NuSMV. Ces différentes transformations prennent du temps et même de l'espace mémoire, dans le cas des traces d'exécution, puisque les traces à analyser sont de l'ordre de 10^9 états. Enfin, il est souhaitable d'avoir un outil spécifique entièrement maîtrisé, pour simplifier les procédures de certification.

L'écriture d'un outil de vérification de traces en reprenant cette seconde approche est donc privilégiée, car cela permettra d'avoir un outil répondant précisément aux besoins exprimés par l'industriel. Il faut cependant définir précisément l'algorithme de vérification employé. L'approche de vérification d'une propriété sur une trace quelconque sera d'abord définie. Puis les propriétés LTL orientées vers le passé seront traitées. Ensuite, seront définis les algorithmes de traitement d'expressions régulières [3]. Enfin, les propriétés paramétrées [4] seront abordées.

6.3.1 Cas d'une propriété LTL du futur quelconque.

La figure 6.5 schématise la succession des différentes étapes de vérification.

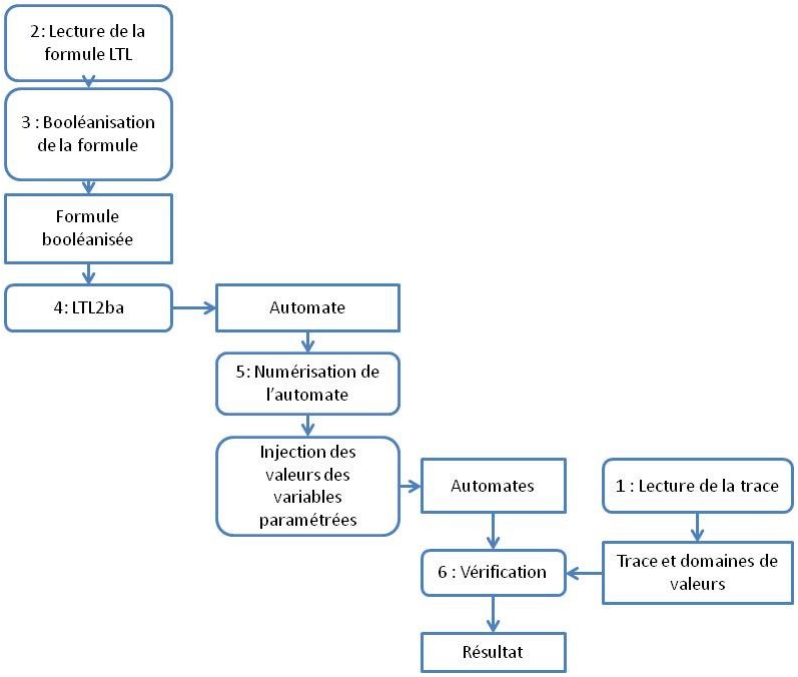

FIGURE 6.5 – Approche de vérification

La vérification d'une propriété sur une trace d'exécution se fait grâce aux étapes suivantes :

3. Voir 4.3.3.
4. Voir 4.3.

1. Lecture de la trace d'exécution : deux modes de lecture sont envisagés : une lecture classique de la trace et une lecture à l'envers de la trace (pour les formules du passé).
2. Analyse syntaxique de la formule temporelle.
3. Booléanisation de la formule : chaque opération de comparaison numérique est remplacée par une variable booléenne. Les relations entre opérations de comparaison et variables propositionnelles sont conservées dans une table de correspondance.
4. Transformation de la formule en automate de Büchi avec l'aide de Ltl2ba.
5. Numérisation de chaque transition de l'automate de Büchi : chaque booléen dans la table de correspondance est remplacé par l'opération de comparaison correspondante.
6. Exécution de l'automate avec la trace d'exécution : l'exécution se fait avec un automate non déterministe et ne conserve que l'ensemble des états courants de l'automate.

La lecture de la trace d'exécution n'est utile que pour l'étape d'exécution de l'automate. Elle peut donc être parallélisée avec les étapes 2 à 5.

6.3.1.1 Lecture de la trace

L'étape de lecture de trace consiste à mettre en mémoire la trace d'exécution issue d'un fichier au format VCD ou Sqlite3. La trace d'exécution à disposition est partielle, dans le sens où seules les valeurs des variables modifiées sont répertoriées dans un état donné. Les valeurs des variables qui n'ont pas subi de changement par rapport à l'état précédent ne sont pas stockées. Il est pourtant nécessaire d'avoir une trace complète pour vérifier une propriété LTL.

Exemple 32 *Le schéma 6.6 présente une trace partielle telle qu'elle est générée et sa version complète.*

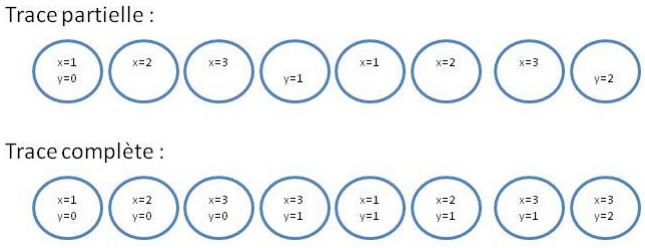

FIGURE 6.6 – Une trace partielle et sa version complète

Pour disposer d'une trace complète, il est nécessaire d'avoir un état dans lequel l'ensemble des variables de la trace soient initialisées. L'état 0 de la trace joue ce rôle. La vérification d'une propriété se fera donc à partir de l'état 1, puisque l'état

0 est uniquement un état d'initialisation des variables. Pour cette étape de lecture, pour minimiser l'espace mémoire occupé par la trace, chaque état de la trace en mémoire est partiel. Il ne contient que les valeurs des variables modifiées. Dans la suite σ désigne la trace complète. La trace partielle lue est $\sigma_{\mathcal{P}}$.

Un état de la trace est ré-assemblé au moment de l'exécution de l'automate. En effet, si $\sigma_{\mathcal{P}0}$ contient l'ensemble des variables de la trace et leur valeur, il est possible d'en déduire récursivement, pour tout i σ_i. Il suffit de mettre à jour σ_i à partir de σ_{i-1}, à chaque pas d'exécution de l'automate, uniquement en changeant les valeurs des variables modifiées.

Soit $\mathscr{U}$ la fonction de mise à jour de l'état courant de la trace. Soit $v_0, ..., v_n$ les variables (nom,valeur) à mettre à jour.

La définition du ré-assemblage de la trace est donc :

$$
\begin{aligned}
&\forall i \in [\![0, |\sigma| - 1]\!], \\
&\sigma_0 = && \sigma_{\mathcal{P}0} \\
&\sigma_{i+1} = && \mathscr{U}(\sigma_i, \sigma_{\mathcal{P}i}) \\
&\mathscr{U}(E, v) = && E', \forall x \in v, E'[nom(x)] = x, et\ x \notin v, E'[nom(x)] = E[nom(x)]
\end{aligned}
$$

6.3.1.2 Analyse syntaxique de la formule

L'étape de lecture de la formule consiste à lire le fichier contenant la formule temporelle, à l'analyser syntaxiquement afin d'obtenir un objet formule correspondant à un format mémoire autorisant des transformations de cette formule. En effet, comme des opérations doivent être effectuées sur la formule temporelle, il est nécessaire de disposer d'un arbre syntaxique abstrait (AST) pour modéliser et transformer toute formule LTL possible. Les transformations de formules à définir sont la booléanisation et la numérisation. Par la suite, toute transformation d'une formule se fera via l'AST la représentant.

6.3.1.3 Booléanisation de la formule

Étant donné que Ltl2ba n'accepte pas les formules contenant des variables numériques, il est nécessaire de remplacer chaque opération de comparaison d'éléments numériques par des variables propositionnelles. C'est l'opération de "booléanisation". À partir d'un AST d'une formule temporelle lue, l'opération de booléanisation est appliquée et une formule interprétable par Ltl2ba est ainsi générée. L'opération de booléanisation génère également une table de correspondance entre les variables booléennes et les opérations de comparaison correspondantes.

Dans la suite, les variables booléennes possèderont des noms génériques "booleen_<id>", ou id est un numéro propre à la formule substituée.

Pour simplifier l'écriture des algorithmes dans la suite, les différents opérateurs seront rangés par classe.

Soit $\mathcal{T}$ la classe des opérateurs temporels. Soit $\mathcal{B}$ la classe des opérateurs booléens. Soit $\mathcal{C}$ la classe des opérateurs de comparaison. Soit $\mathcal{A}$ la classe des opérateurs arithmétique. $\mathcal{A}$ se décompose en $\mathcal{A}_{\mathbb{N}}$ et $\mathcal{A}_{\mathbb{Q}}$, respectivement les classes des opérateurs arithmétiques pour les entiers et les rationnels. La sous-classe d'une classe $\mathcal{X}$ dont les opérateurs sont d'arité n s'écrit $\mathcal{X}^n$. L'ensemble des variables ou constantes booléennes s'écrit $\mathbb{B}$. L'ensemble des variables numériques s'écrit $\mathbb{V}$

TABLEAU 6.2 – Classe d'opérateur.

Classe	Définition
$\mathcal{T}$	$\{\Box, \mathcal{U}, \Diamond\}$
$\mathcal{B}$	$\{\wedge, \vee, \neg, \Rightarrow\}$
$\mathcal{C}$	$\{<, >, =, \neq, \leq, \geq\}$
$\mathcal{A}$	$\mathcal{A}_{\mathbb{N}} \cup \mathcal{A}_{\mathbb{Q}}$
$\mathcal{A}_{\mathbb{N}}$	$\{+, -, \times, \div, \&b, \|b, !b, \hat{}, \ll, \gg\}$
$\mathcal{A}_{\mathbb{Q}}$	$\{+, -, \times, \div\}$

L'opération de booléanisation est une fonction $\mathscr{B}$ nécessitant une formule ϕ et une table de correspondance tab. Pour un élément de Lt[5], l'algorithme est :

$$\begin{aligned}
&\forall \phi_1, \phi_2 \in Lt, \\
&\forall \triangle \in \mathcal{T}^1 \cup \mathcal{B}^1, && \mathscr{B}(\triangle\phi_1, tab) = \triangle(\mathscr{B}(\phi_1, tab)) \\
&\forall \triangle \in \mathcal{T}^2 \cup \mathcal{B}^2, && \mathscr{B}(\phi_1 \triangle \phi_2, tab) = (\mathscr{B}(\phi_1, tab)) \triangle (\mathscr{B}(\phi_2, tab)) \\
&\forall b \in \mathbb{B}, && \mathscr{B}(b, tab) = b
\end{aligned}$$

Pour un élément de Exp[5], l'algorithme est :

$$\begin{aligned}
\forall \nu_1, \nu_2 \in Exp, \quad & \forall \triangle \in \mathcal{C} \\
\mathscr{B}(\nu_1 \triangle \nu_2, tab) = & \\
si\ \exists X, tab[X] = \nu_1 \triangle \nu_2 \in tab, \quad & boolean_X \\
sinon X = card(tab) + 1, tab[X] = \nu_1 \triangle \nu_2, \quad & boolean_X
\end{aligned}$$

Concrètement, lorsqu'une opération de comparaison n'est pas trouvée dans la table tab, elle est ajoutée dans celle-ci. La booléanisation ne doit pas consister uniquement à remplacer chaque opération de comparaison par une variable booléenne.

Exemple 33 *Soit la propriété*

$$\Box\left((x = 0) \Rightarrow ((x = 0)\ \mathcal{U}\ (x = 1) \wedge \Box\left((x = 1) \Rightarrow (x = 1)\ \mathcal{U}\ (x = 2)\right)\right)\right) \tag{6.4}$$

5. Voir la grammaire dans 4.3.

Si la booléanisation consiste uniquement à remplacer chaque opération de comparaison par une variable booléenne, alors la formule booléanisée devient :

$$\Box\,(a \Rightarrow (b\,\mathcal{U}\,c \wedge \Box\,(d \Rightarrow e\,\mathcal{U}\,f))) \tag{6.5}$$

La transformation de cette formule fournit un automate de Büchi à 6 états. Chaque état a 4 transitions.

Si la booléanisation consiste à remplacer chaque opération de comparaison identique par une même variable booléenne, alors la formule booléanisée devient :

$$\Box\,(a \Rightarrow (a\,\mathcal{U}\,b \wedge \Box\,(b \Rightarrow b\,\mathcal{U}\,c))) \tag{6.6}$$

La transformation de cette formule fournit un automate de Büchi à 4 états. Chaque état a 3 transitions.

Il est donc indispensable de pouvoir détecter deux opérations de comparaison sémantiquement identiques. Cette détection optimise la taille de l'automate. Pour détecter deux opérations de comparaison identiques, il est nécessaire de normaliser chaque opération de comparaison, de manière à obtenir une formule qui ne dépende pas de placement purement syntaxique.

Exemple 34 *Soient deux opérations de comparaison :*

$$w + x > y + z \tag{6.7}$$
$$z + y < w + x \tag{6.8}$$

Ces deux formules signifient la même chose mais sont syntaxiquement écrites différemment.

Une solution consiste à ne conserver que deux opérateurs de comparaison : $<$ et $=$. Les autres opérateurs peuvent se déduire de ces deux opérateurs par utilisation des symétries et des opérateurs duaux.

TABLEAU 6.3 – Opérateurs de comparaison et expressions équivalentes employées

Expression	Expression équivalente
$x > y$	$y < x$
$x \geq y$	$\neg\,(x < y)$
$x \leq y$	$\neg\,(y < x)$
$x \neq y$	$\neg\,(x = y)$

Pour les éléments syntaxiques liés à l'ordre des opérateurs, tels que les additions soustractions, multiplications, divisions, il suffit de définir des opérateurs d'arité n pour les additions et multiplications. Pour les soustractions, elles peuvent être transformées en addition par passage à l'opposé. Concernant les divisions, elles peuvent

être transformées en multiplication par passage à l'inverse, uniquement dans le cas des rationnels. C'est pourquoi les flottants ne sont pas utilisés.

$$x - y = x + -y$$
$$x/y = x \times 1/y$$

Cependant, cette optimisation n'est que partielle. En effet, l'opération de booléanisation entraîne une perte d'information sur les liens entre les variables numériques.

Exemple 35 *Soient les opérations de comparaison : $x < 0$ et $x > 0$. Ces deux opérations seront remplacées respectivement par $booleen\_0$ et $booleen\_1$. Entre ces deux variables booléennes, le fait que si $booleen\_0$ est vrai, alors $booleen\_1$ est nécessairement faux et inversement n'est pas conservé.*

Proposition 5 *La perte d'information liée à la booléanisation n'a pas d'incidence sur la validité de la vérification d'une propriété, mais sur le temps de vérification d'une propriété.*

Preuve : Soient deux opérations de comparaisons $c_0(x_0, ...x_n)$ et $c_1(y_0, ..., y_m)$ faisant respectivement intervenir $n+1$ et $m+1$ variables respectivement différentes. S'il n'existe pas de variables numériques communes entre l'ensemble des opérandes de c_0 et l'ensemble des opérandes de c_1, alors il n'y a pas, a priori, de relation implicite entre c_0 et c_1. Dans la suite, l'intersection des ensembles des opérandes de c_0 et c_1 est non vide.

Soit $\triangle$, une relation logique. Cette relation lie implicitement c_0 et c_1 telle que $c_0(x_0, ...x_n) \triangle c_1(y_0, ..., y_m)$.

À la booléanisation, la relation $booleen_0 \triangle booleen_1$ n'est pas conservée. c_0 et c_1 sont respectivement remplacés par $booleen_0$ et $booleen_1$.

Chaque transition de l'automate de Büchi généré à partir de la formule temporelle booléanisée, possède une formule de logique propositionnelle.

Soit une transition de cet automate liant les variables $booleen_0$ et $booleen_1$ par un prédicat $p(booleen\_0, booleen\_1, q_0.., q_N)$. À supposer que q_0, q_N soient fixées. p dépend uniquement de $booleen\_0$ et $booleen\_1$ dans la suite de la démonstration. $\tilde{p}(booleen\_0, booleen\_1) = p(booleen\_0, booleen\_1, q_0.., q_N)$.

Trois cas sont possibles :

$$booleen_0 \triangle booleen_1 \Rightarrow \quad \tilde{p}(booleen\_0, booleen\_1) \tag{6.9}$$
$$booleen_0 \triangle booleen_1 \Rightarrow \quad \neg\,(\tilde{p}(booleen\_0, booleen\_1)) \tag{6.10}$$
$$booleen_0 \triangle booleen_1 \Rightarrow \quad true \tag{6.11}$$

Pour le cas 6.9, l'existence de cette relation implique que la propriété $\tilde{p}$ est toujours vraie. Pour le cas 6.10, l'existence de cette relation implique que la propriété $\tilde{p}$ est toujours fausse. Pour le cas 6.11, l'existence de cette relation n'a pas d'incidence sur la propriété.

Lors de la numérisation, le remplacement des variables *booleen_0* et *booleen_1* conduit à la formule $\tilde{p}(c_0(x_0,..x_n),c_1(y_0,...y_m))$ pour cette transition. La relation implicite $c_0(x_0,...x_n)\triangle c_1(y_0,...,y_m)$ est rétablie.

Pour les cas 6.9 et 6.10, la propriété est soit toujours vraie, soit toujours fausse. Si elle est toujours fausse, la transition correspondante devrait être enlevée. En effet, comme la formule est toujours évaluée à faux, l'état que pointe la transition n'est pas atteignable. Si elle est toujours vraie, l'évaluation de cette formule peut être remplacée par la constante *true*, ce qui accélère le temps de vérification.

Pour le dernier cas (6.11), la propriété doit tout le temps être calculée.

Ce raisonnement doit être fait pour chaque opération de comparaison et pour chaque q_i intervenant dans chaque prédicat p de chaque relation de transition de l'automate de Büchi ■.

6.3.1.4 Transformation de la propriété temporelle en automate de Büchi

Pour cette étape, l'outil Ltl2ba, transforme la formule booléanisée en automate de Büchi non déterministe. La déterminisation d'un tel automate étant de complexité exponentielle en temps [Sakarovitch 2003], elle ne sera pas effectuée. En contrepartie, il est nécessaire de définir de nouveaux algorithmes pour exécuter un automate de Büchi non déterministe. En pratique, lors de l'exécution de l'automate, il n'y a pas un état courant de l'automate mais un ensemble d'états courants.

6.3.1.5 Numérisation de chaque transition de l'automate de Büchi

Pour chaque variable booléenne dont le nom est une entrée de la table de correspondance, la variable est remplacée par la formule correspondante. Cette opération de transformation est à faire sur chaque transition de l'automate de Büchi obtenu.

Soit $\mathcal{P}$ l'ensemble des prédicats du premier ordre. Soit $\mathcal{N}$ l'opération de numérisation d'un prédicat à l'aide d'un tableau de correspondance *tab*. L'algorithme de numérisation est donc :

$$\begin{aligned}
\forall p_1,p_2,\in\mathcal{P}& & \\
\forall\triangle\in\mathcal{B}^1,& \quad \mathcal{N}(\triangle p_1,tab)=\triangle(\mathcal{N}(p_1,tab)) & \\
\forall\triangle\in\mathcal{B}^2,& \quad \mathcal{N}(p_1\triangle p_2,tab)=\mathcal{N}(p_1,tab)\triangle\mathcal{N}(p_2,tab) & \\
\forall p\in\mathbb{B}_V,\mathcal{N}(p,tab)=& & \\
si\ p=booleen_X,X\in[\![0,card(tab)-1]\!]& \quad tab[X] & \\
sinon& \quad p &
\end{aligned}$$

6.3.1.6 Exécution de l'automate avec la trace d'exécution

L'exécution se déroule en deux temps qui sont :

- le cas nominal
- la fin de la trace, qui sera traitée dans la section 6.4.

Lors de l'exécution de l'automate, il faut travailler avec un ensemble d'états courants, puisque l'automate généré par Ltl2ba est non déterministe.

La démarche du cas nominal est la suivante :

1. L'état suivant de la trace est chargé, s'il existe. Lors du chargement, les compteurs de modifications sont incrémentés à la volée, lorsqu'une variable change de valeur, ainsi que pour les temps de modification. La variable globale temps τ est incrémentée systématiquement.
2. Si l'état suivant de la trace n'existe pas, l'algorithme nominal se termine et l'algorithme de fin de trace prend le relais.
3. La formule de chaque transition pour chaque état de l'automate appartenant à l'ensemble des états courants est évaluée. Si la formule est vraie, l'état pointé par la transition est ajouté au nouvel ensemble des états courants de l'automate.
4. L'ensemble des états suivants devient l'ensemble des états courants. Dans cet ensemble, chaque état est unique, alors que deux transitions différentes peuvent conduire à un même état. Cela évite une explosion combinatoire du nombre d'états et donc du nombre de transitions à vérifier. La figure 6.7 schématise la simplification
5. Si l'état courant est vide, alors la propriété est fausse. L'algorithme se termine.
6. Sinon, l'algorithme recommence à l'étape 1.

La figure 6.7 résume la démarche précédente sur une trace contenant deux variables booléennes p et q. Les états barrés d'une croix rouge sont les états appartenant déjà à l'état courant de l'automate de Büchi.

Soit une trace d'exécution σ. Soit $E_{\mathcal{A}}(i)$ l'ensemble des états courants de l'automate de Büchi $\mathcal{A}$ à l'état i de σ. L'état initial de $\mathcal{A}$ est $I_{\mathcal{A}}$. Pour un état $\varepsilon \in E_{\mathcal{A}}(i)$ donné, T_ε est l'ensemble des transitions ayant pour origine ε. Soit une transition τ donnée. L'évaluation de la formule de τ est $[\![\tau]\!]$. L'état de $\mathcal{A}$ accessible à partir de τ, si $[\![\tau]\!]$ est vrai, est $\nu(\tau)$.

L'algorithme formalisé est donc, pour tout n dans $[\![0, |\sigma| - 1]\!]$. :

$$\begin{aligned} E_{\mathcal{A}}(0) &= I_{\mathcal{A}} \\ E_{\mathcal{A}}(n) &= \bigcup_{\varepsilon \in E_{\mathcal{A}}(n-1)} \cup_{\tau \in T_\varepsilon, [\![\tau]\!]_{\sigma i} = true} \nu(\tau) \end{aligned}$$

Concrètement, cette équation récursive donne l'ensemble des états courants de l'automate à l'état σ_i de la trace. S'il existe i, tel que i est dans $[\![0, |\sigma|-1]\!]$, $E_{\mathcal{A}}(i) = \emptyset$ et pour tout j dans $[\![0, |\sigma| - 1]\!]$, $j < i$, $E_{\mathcal{A}}(j) \neq \emptyset$ alors la propriété est fausse à l'état i et pour tout k de $[\![0, |\sigma| - 1]\!]$, tel que $k \geq i$, $E_{\mathcal{A}}(k) = \emptyset$.

L'algorithme de fin de trace sera traité dans la sous-section 6.4.1.

6.3.2 Complexité de l'algorithme

La complexité en temps de l'algorithme dépend essentiellement de trois paramètres :

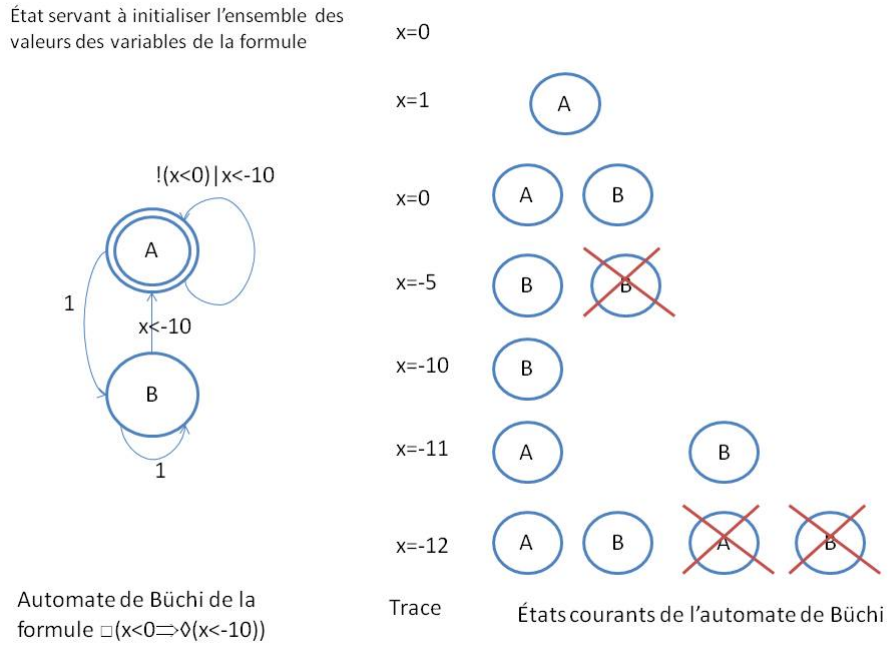

FIGURE 6.7 – Simplification des états courants

– le cardinal t de l'ensemble $\rightarrow$ des transitions de l'automate de Büchi exécuté,
– le cardinal e de l'ensemble des états Q de l'automate de Büchi exécuté,
– le nombre $|\sigma|$ d'états dans la trace d'exécution.

Dans le pire des cas, à chaque étape de calcul, l'ensemble des états courants est égal à l'ensemble des états de l'automate de Büchi.

$$\forall n \in [\![1, |\sigma| - 1]\!], E_{\mathcal{A}}(n) = Q \tag{6.12}$$

Pour un pas de calcul i, il est donc nécessaire de parcourir chaque transition de chaque état q_o,...,$q_{card(Q)-1}$ de Q et d'évaluer la formule de la transition. Le parcours de chaque état commence par q_0.

Lorsqu'une transition d'un état q_p est évaluée à vraie, alors il faut rajouter q_p dans $E_{\mathcal{A}}(i)$, à condition qu'il n'y soit pas déjà. Dans le pire des cas, lorsque les transitions de q_0 ont été parcourues, tous les états de $\mathcal{A}$ ont été ajoutés. Cependant, il est nécessaire de parcourir $E_{\mathcal{A}}(i)$ pour s'assurer qu'il contient déjà q_p. Le temps de parcours de $E_{\mathcal{A}}(i)$ a une complexité en $\mathcal{O}(\log_2 e)$

La complexité est donc en :

$$\mathcal{O}(|\sigma| \times t \times \log_2(card(Q))) \tag{6.13}$$

La complexité en mémoire dépend :
– du nombre n de variables dans la trace σ.

- du nombre p de valeurs précédentes (profondeur) conservées pour chaque variable. Si aucune valeur précédente n'est conservée, alors $p = 0$.
- de l'automate complet $\mathcal{A}_m$
- du nombre maximum d'états de l'automate de Büchi pour l'état i de σ : $card_{max,i\in[\![0,|\sigma|-1]\!]}(E_{\mathcal{A}})$
- du nombre maximum d'états de l'automate de Büchi pour l'état i+1 de σ : $card_{max,i\in[\![0,|\sigma|-1]\!]}(E_{\mathcal{A}})$

Dans le pire des cas, $card_{max,i\in[\![0,|\sigma|-1]\!]}(E_{\mathcal{A}}) = card(Q)$.
La complexité en mémoire est donc :

$$\mathcal{O}(n \times (p+1) + \mathcal{A}_m + 2 \times card(Q)) \tag{6.14}$$

6.3.3 Vérification de propriétés LTL du passé

L'outil de génération d'automates de Büchi Ltl2ba ne permet pas de travailler avec la PLTL (Past LTL).

Comme l'analyse de la trace s'effectue a posteriori, une solution consiste à lire la trace d'exécution à l'envers, tout en continuant à travailler avec Ltl2ba et les opérateurs de LTL classiques. Ainsi, il est possible de travailler avec des propriétés écrites avec une LTL orientée purement vers le passé ou avec une LTL orientée purement vers le futur. Cependant, aucun mélange des opérateurs n'est toléré. Il est pourtant nécessaire de s'assurer qu'une telle lecture est conforme à la sémantique adoptée dans le chapitre 4.

Sémantique de la logique du passé

Définition 24 (Trace inverse) *Soit une trace d'exécution $\sigma = \sigma_0\sigma_1...\sigma_{|\sigma|-1}$ de longueur $|\sigma|$. La trace d'exécution $\tilde{\sigma}$ est la trace d'exécution construite à partir de σ en inversant l'ordre de lecture, en ajoutant en tout début une réplique de l'état $\sigma_{|\sigma|-1}$ et en supprimant l'état σ_0 à la fin. La taille de la trace est de ce fait conservée : $|\tilde{\sigma}| = |\sigma|$.*

Concrètement, $\tilde{\sigma} = \sigma_{|\sigma|-1}\sigma_{|\sigma|-1}\sigma_{|\sigma|-2}\sigma_{|\sigma|-3}...\sigma_1$

Interpréter les opérateurs du passé sur σ revient à interpreter leurs équivalents futur sur $\tilde{\sigma}$.

Le bouclage de fin de trace se fait alors sur l'état σ_1 (ou $\tilde{\sigma}_{|\tilde{\sigma}|-1}$). Comme pour la lecture classique, la vérification de la propriété sur la trace $\tilde{\sigma}$ ne commence qu'à partir de l'état $\tilde{\sigma}_0$.

Ré-assemblage de la trace Le ré-assemblage de la trace pour une lecture inversée ne se traite pas de la même manière que pour la lecture classique. En effet, la trace lue ne contient pour chaque état que les variables modifiées et leur nouvelle valeur. Pour une variable non modifiée, la valeur de celle-ci est égale à celle de l'état précédent. Pour une lecture inversée, il est donc nécessaire de lire la trace dans le sens classique, puis d'effectuer un traitement sur les données enregistrées pour lire

la trace à l'envers. Le schéma 6.8 résume l'algorithme de traitement de la trace pour une lecture inverse.

Dans la suite, σ fait référence à la trace, telle que pour chaque état la valeur de chaque variable est connue. $\sigma_{\mathcal{P}}$ fait référence à la trace lue, à savoir que seules les variables qui changent de valeur à l'état σ_i sont connues. Soit la fonction μ telle que dans la définition 5.

Définition 25 (Lien entre les numéros d'états d'une trace et sa trace inverse) *Soit la fonction γ, qui pour un numéro d'état de σ retourne le numéro d'état de $\tilde{\sigma_{\mathcal{P}}}$.*

$$\begin{aligned} \gamma : \quad & \mathbb{N} \to \mathbb{N} \\ \gamma(0) = \quad & |\sigma| - 1 \\ \forall n \in [\![1, |\sigma - 1|]\!], \quad \gamma(n) = \quad & |\sigma| - n \end{aligned}$$

La fonction qui pour un numéro d'état de $\tilde{\sigma}$ retourne le numéro d'état correspondant de σ s'écrit γ^{-1}. γ et γ^{-1} sont également utilisables sur $\sigma_{\mathcal{P}}$ et $\tilde{\sigma}_{\mathcal{P}}$, en remplaçant respectivement σ par $\sigma_{\mathcal{P}}$ et $\tilde{\sigma}$ par $\tilde{\sigma}_{\mathcal{P}}$.

L'égalité suivante est respectée :

$$\begin{aligned} \sigma_{|\sigma|-1} = \quad & \tilde{\sigma_0} \\ \forall n \in [\![1, |\sigma_{\mathcal{P}}| - 1]\!], \sigma_n = \quad & \tilde{\sigma}_{\gamma(n)} \end{aligned}$$

$\mathcal{V}$ désigne l'ensemble des variables de la trace. (confer la définition 4).

Définition 26 (Ensemble des variables modifiées pour un état donné) *Soit $\mathcal{V}_N$ l'ensemble des variables modifiées à l'état $\sigma_{\mathcal{P}N}$. Alors :*

$$\mathcal{V}_{\gamma(n)} = \{v \in \mathcal{V}, \sigma_n \neq \sigma_{n+1}\}$$

En posant $\gamma(n) = N$, il vient :

$$\mathcal{V}_N = \{v \in \mathcal{V}, \sigma_{\gamma^{-1}(N)} \neq \sigma_{\gamma^{-1}(N)+1}\} \tag{6.15}$$

Seules des traces partielles sont conservées en mémoire. Les traces complètes sont recomposées à l'exécution de l'automate de Büchi de la propriété à vérifier.

Définition 27 (Expression d'un état de la trace partielle inverse) *La trace partielle inversée $\tilde{\sigma}_{\mathcal{P}}$ s'écrit donc :*

$$\begin{aligned} \forall N \in [\![1, |\sigma_{\mathcal{P}}|]\!], & \\ \sigma_{\mathcal{P}N} : \quad & \mathcal{V}_N \to \mathcal{V}\mathcal{T}ype \\ \sigma_{\mathcal{P}N}(v) = \quad & \sigma_{\gamma^{-1}(N)}(v) \quad = \sigma_{\mathcal{P}\mu_{\sigma,v}(\gamma^{-1}(N))}(v) \\ \sigma_{\mathcal{P}0} = \quad & \sigma_{\mathcal{P}1} \end{aligned}$$

Dans la suite, le raisonnement reposera sur l'ordre chronologique des événements lorsqu'un numéro d'état est évoqué. La première trace de la figure 6.8 est la trace d'exécution contenant pour chaque état la valeur de toutes les variables de la trace. Lorsqu'une trace est lue dans le sens normal, seules les variables modifiées apparaissent dans un état. Dans la figure 6.8, la seconde trace ($\sigma_{\mathcal{P}|x}$), seuls les états où x est modifié sont représentés. Une fois la trace en mémoire, les états de traces restent partiels. Autrement dit, un état ne contient que les valeurs des variables modifiées.

La troisième trace est la trace partielle symétrique de $\sigma_{\mathcal{P}|x}$ construite à partir de la trace précédente. Cette trace peut être lue uniquement à l'envers.

Concrètement, un état de numéro d'identification i de la trace contient la valeur de la variable x à partir de ce nouvel état. Si la trace est lue à l'envers, il faut qu'à l'état numéro $i-1$ la valeur de x soit connue. Il faut donc récupérer la valeur de x dans l'état où celui-ci a été modifié dans $\sigma_{\mathcal{P}}$

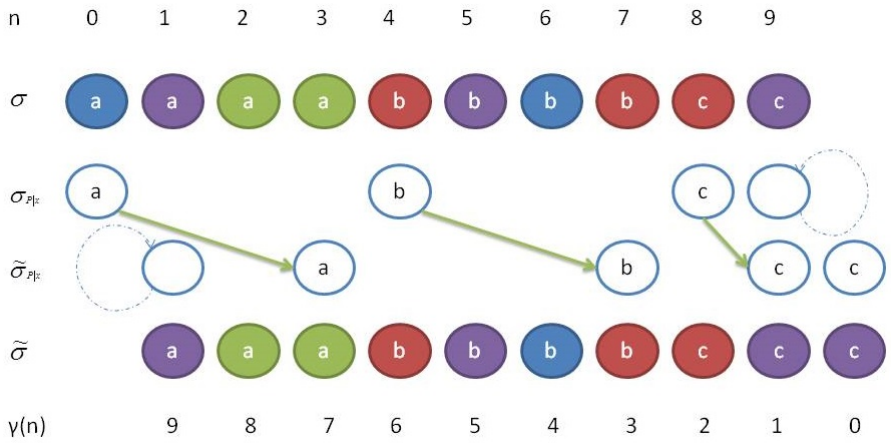

FIGURE 6.8 – Traitement pour une lecture inversée

6.3.4 Propriétés définies par une expression régulière

Les propriétés de séquences [6] peuvent être définies par une expression régulière plutôt que par des formules LTL qui sont longues à transformer avec Ltl2ba, du fait de l'imbrication des différents opérateurs temporels.

L'expression régulière est transformée directement en automate de Büchi. Cet automate sera ensuite exécuté suivant la procédure définie dans la sous-section 6.3.1.

Étant donné qu'il existe une quantité indénombrable d'expressions régulières, la définition des différents algorithmes est faite à partir d'une séquence de n+1 éléments notés $a_0, ..., a_n$. En pratique, ces éléments correspondent à des formules sans opérateur temporel.

Pour générer l'automate de Büchi associé à une séquence, plusieurs étapes sont nécessaires :

6. Voir 4.3.3.1.

- création de deux états extrêmes ;
- création des états et transitions caractéristiques du type de séquence (faible, étendue ou forte) ;
- modification des transitions pour prendre en compte la condition de vérification de la séquence, à savoir la trace doit être vérifiée dès le premier état, dès que le premier élément de la séquence est rencontré, ou dès qu'une condition est remplie ;
- prise en compte du nombre de répétitions de la séquence, par l'ajout des états finaux.

La première étape consiste à créer deux états, l'un étant l'état initial et l'autre, appelé état extrême, l'état où la séquence a été lue une fois entièrement. Ces états seront complétés au fur et à mesure. Le caractère final d'un état sera entre autres rajouté par la suite.

Un état de l'automate dont le numéro d'identifiant est i s'écrit e_i. La fonction f, pour un état e_i et un booléen donné b, définit si e_i est final ($b = true$) ou non. La fonction $Init$ est définie de manière analogue à f, mais pour le caractère initial de l'état e_i. De même pour la fonction $Skip$, qui détermine si l'état e_i est un état qui permet de sauter le reste de la trace, lorsque e_i est atteint. L'utilisation de $Skip$ détruit toutes les transitions existantes de l'état.

La fonction $lier(e, e', f)$ crée une transition de e vers e' avec comme formule f. Si f est la constante $true$ alors il s'agit d'une transition toujours activée. Si f est la constante $false$ alors la transition qui existait déjà entre e et e' est détruite. Si une transition entre e et e' existait déjà, alors elle est remplacée.

La fonction $creer(i)$ génère un état non final, non initial, qui ne saute pas le reste de la trace lorsque cet état est créé, et dont l'identifiant est i.

$$\begin{aligned} e_0 = \quad & creer(0) \\ e_X = \quad & creer(extreme) \\ & Init(e_0, true) \end{aligned}$$

6.3.4.1 Type de séquence

La transition entre e_{init} et e_1 sera traitée ici comme s'il n'y avait pas de condition d'activation c_{ini} de la séquence, à savoir la séquence est à vérifier que si le booléen s'est avéré vrai. Le cas des conditions d'activation sera traité plus loin dans cette sous-section. Les types de séquences sont définis en 4.3.3.1.

La séquence forte écrite $\{a_0, a_1, ...a_n\}$ dans le langage d'expressions régulières se traduit par un automate qui ne possède que des transitions correspondant à un élément de la séquence. Une transition lie e_0 à e_1 avec la formule a_0, puis e_1 est lié à e_2 avec une transition de formule a_1..., jusqu'à l'état extrême lié avec l'état précédant e_{n-1} par une transition de formule a_n. Le schéma 6.9 permet de visualiser ces états et transitions.

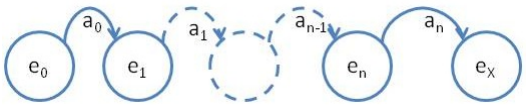

FIGURE 6.9 – Traitement de la séquence forte

L'algorithme de génération est donc le suivant :

$$\forall i \in [\![1, n]\!], e_i = creer(i); \quad lier(e_{i-1}, e_i, a_{i-1}); \\ lier(e_n, e_X, a_n)$$

La séquence étendue écrite $[a_0; a_1...a_n]$ dans le langage d'expressions régulières se traduit par un automate qui ne possède que des transitions correspondant à un élément de la séquence, mais dont les transitions d'un état vers lui même sont ajoutées. Une transition lie donc e_0 à e_1 avec la formule a_0, puis e_1 est lié à lui même avec une transition de formule a_0. L'état est également lié à e_2 avec une transition de formule a_1... Le schéma 6.10 permet de visualiser ces états et transitions.

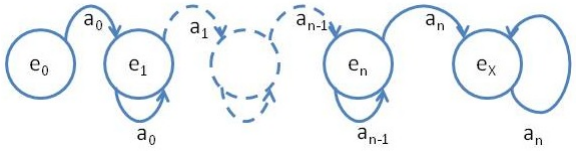

FIGURE 6.10 – Traitement de la séquence étendue

L'algorithme de génération est donc le suivant :

$$\forall i \in [\![1, n]\!], e_i = creer(i); \quad lier(e_{i-1}, e_i, a_{i-1}); \quad lier(e_i, e_i, a_{i-1}); \\ lier(e_n, e_X, a_n)$$

La séquence faible écrite $(a_0, a_1, ..., a_n)$ dans le langage d'expressions régulières se traduit par un automate qui possède des transitions correspondant à un élément de la séquence mais également des transitions autorisant tout état de trace sur chaque état de l'automate. Une transition lie e_0 à e_1 avec la formule a_0, puis e_1 est lié à e_2 avec une transition de formule a_1. e_1 est également lié avec lui même avec une transition autorisant tout. De même, pour e_2, e_3 ... jusqu'à e_n. Le schéma 6.11 permet de visualiser ces états et transitions.

L'algorithme de génération est donc le suivant :

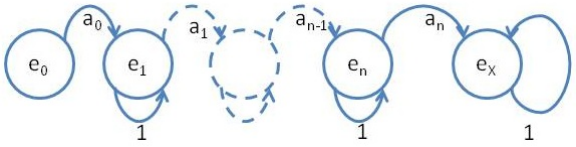

FIGURE 6.11 – Traitement de la séquence faible

$$\forall i \in [\![1,n]\!], e_i = creer(i); \quad lier(e_{i-1}, e_i, a_{i-1}); \quad lier(e_i, e_i, true);$$
$$lier(e_n, e_X, a_n)$$

6.3.4.2 Condition d'activation de la séquence

Une séquence se vérifie :
- soit dès le début de la trace,
- soit dès que le premier élément de la séquence se produit,
- soit dès que le premier élément de la séquence et une condition additionnelle se produisent.

Le cas d'une séquence forte sera considéré pour présenter cette sous-section, et faire le lien avec les propriétés.

Si la séquence commence dès le début de la trace (6.12), la séquence s'écrit $[1]\{a_0; a_1; ...; a_n\}$ ou en abrégé $\{a_0; a_1; ...; a_n\}$. Il faut alors juste ajouter une transition entre e_0 et e_1 avec pour formule, le premier élément de la trace.

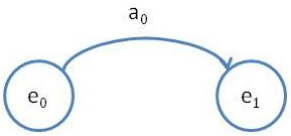

FIGURE 6.12 – La séquence commence immédiatement

Toutes les actions ont déjà été réalisées précédemment.

Si la séquence peut ne pas commencer immédiatement (6.13), la séquence s'écrit $[0]\{a_0; a_1; ...; a_n\}$. Deux transitions sont alors nécessaires : une avec la formule correspondant au premier élément de la trace de e_0 vers e_1 et une autre avec la négation de cette formule.

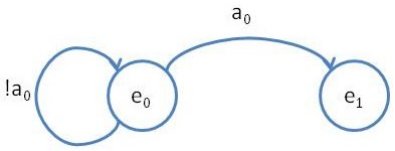

FIGURE 6.13 – La séquence commence dès l'apparition du premier élément

$$lier(e_0, e_0, !a_0)$$

Si la séquence ne commence que lorsqu'une condition d'activation est requise (figure 6.14), la séquence s'écrit $[c_{ini}]\{a_0; a_1; ...; a_n\}$. Il faut alors deux transitions : une avec la conjonction de la formule pour la condition et de la formule correspondant au premier élément de la trace de e_0 vers e_1 et une autre avec la négation de cette formule.

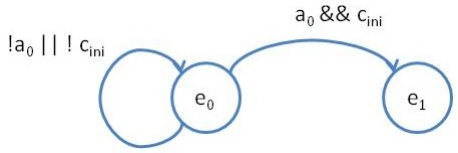

FIGURE 6.14 – La séquence commence dès l'apparition du premier élément et de la condition simultanément.

$$lier(e_0, e_0, !a_0 \parallel !cond); \quad lier(e_0, e_1, a_0 \,\&\&\, cond);$$

6.3.4.3 Répétition de séquence

Une séquence peut se produire :
- une fois,
- une infinité de fois ou zéro fois,
- une infinité de fois avec un minimum d'une fois,
- jusqu'à ce que la condition x soit vraie simultanément avec la fin d'une séquence complète.

Une fois : Dans ce cas, il suffit de rendre e_X final, et d'ajouter une transition skip à e_X, pour sauter le reste de la trace. Toute autre transition existant déjà dans cet état est alors supprimée. La figure 6.15 traite le cas d'une séquence étendue unique sans condition initiale d'activation.

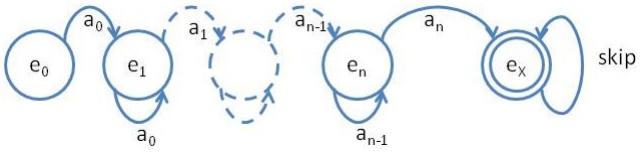

FIGURE 6.15 – Traitement de la séquence étendue unique

L'algorithme est donc :

$$f(e_X, true); \quad skip_a(e_X, true);$$

Une infinité de fois ou zéro fois : Dans ce cas, l'état initial et l'état extrême sont des états finaux. Il faut également ajouter une transition de l'état extrême à l'état 1 avec une formule correspondant au premier élément de la séquence. La figure 6.16 traite le cas d'une séquence étendue, se répétant autant de fois que voulu (0 fois inclus), sans condition initiale d'activation.

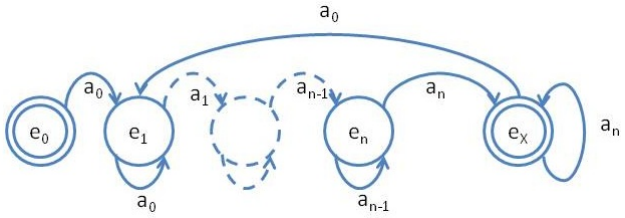

FIGURE 6.16 – Traitement de la séquence étendue étoilée

L'algorithme est donc :

$$f(e_X, true); \quad f(e_0, true); \quad lier(e_X, e_1, a_0);$$

Une infinité de fois et au moins une fois : Dans ce cas, seul l'état extrême est un état final. Il faut également ajouter une transition de l'état extrême à l'état 1 avec une formule correspondant au premier élément de la séquence. La figure 6.17 traite le cas d'une séquence étendue, se répétant autant de fois que possible avec un minimum d'une fois, sans condition initiale d'activation.

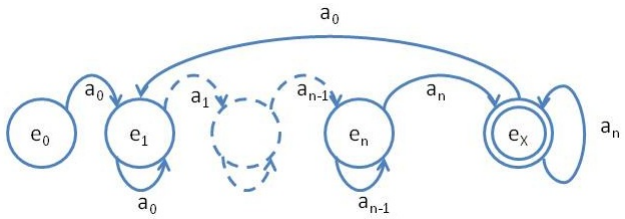

FIGURE 6.17 – Traitement de la séquence étendue plus

L'algorithme est donc :

$$f(e_X, true); \quad lier(e_X, e_1, a_0);$$

Jusqu'à ce que la condition c_{fin} soit vraie simultanément avec la fin d'une séquence complète : Dans ce cas, seul e_X est un état final. Il faut également ajouter une transition de e_X à e_1 avec une formule correspondant au premier élément de la séquence, a_0. La figure 6.18 traite le cas d'une séquence étendue avec condition de terminaison et sans condition initiale d'activation.

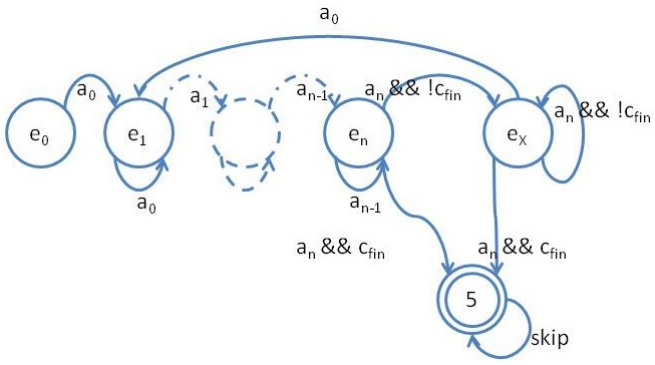

FIGURE 6.18 – Traitement de la séquence avec condition de terminaison

L'algorithme est donc :

$$\begin{array}{rl} e_{fin} = creer(fin); & lier(e_X, e_X, a_n \;\&\&\; !c_{fin}); \\ Skip(e_{fin}, true); & lier(e_n, e_X, a_n \;\&\&\; !c_{fin}); \\ f(e_{fin}, true); & lier(e_n, e_{fin}, a_n \;\&\&\; c_{fin}); \\ lier(e_X, e_1, a_0); & lier(e_X, e_{fin}, a_n \;\&\&\; c_{fin}); \end{array}$$

L'automate de Büchi ainsi généré est exécuté selon la méthode décrite en 6.3.1.

6.3.5 Prise en charge de propriétés paramétrées

6.3.5.1 Constat

Certaines propriétés doivent être vérifiées pour chaque valeur prise par une variable donnée du programme. Cela implique qu'il est nécessaire d'écrire autant de propriétés que de valeurs prises par la variable.

L'écriture répétée est d'une part fastidieuse et d'autre part source d'erreur, puisque l'oubli d'une valeur peut laisser échapper un bug très facilement.

6.3.5.2 Principe

Exemple 36 *Soit une application qui crée un certain nombre de ports pour communiquer avec d'autres applications sur un réseau donné. Pour pouvoir utiliser un port étiqueté par son identifiant id, il faut que celui-ci ait été préalablement créé. La propriété consiste donc à vérifier que pour tout port x :*

$\boxdot\ (utilisation(x) \Rightarrow \Diamond\ (creation(x)))$,

ce qui s'écrit avec le langage défini dans le chapitre 4 :

$\boxdot\ (utilisation = utilisation\$1 \Rightarrow (\Diamond\ (creation = utilisation\$1)))$

La vérification exhaustive pour toutes les valeurs possibles de x (par exemple si x est un entier, l'ensemble des entiers de 64 bits) n'est pas envisageable. Cependant,

il est possible de se restreindre à l'ensemble des valeurs rencontrées dans la trace, et vérifier la propriété temporelle pour chaque valeur de x prise au cours de l'exécution du programme à vérifier.

Une variable paramétrique x a un domaine de définition donné Δ_x et prend un ensemble de valeurs donné dans la trace D_x (dans le langage, le préfixe de la variable paramétrée correspond au domaine de variation de la variable référente). Cette approche implique qu'il ne sera pas possible de vérifier une propriété paramétrée définie sur l'ensemble $\Delta_x \backslash D_x$, puisque cet ensemble est trop grand pour être exploré.

Dans la suite, la formule paramétrée ϕ dispose de N variables paramétrées. L'approche va donc consister à calculer tous les N-uplets de valeurs possibles et à exécuter pour chaque N-uplet l'automate de Büchi correspondant à la formule ϕ, dans lequel les variables paramétrées ont été remplacées par le N-uplet.

6.3.5.3 Calcul de l'ensemble des valeurs d'une variable

Pour chaque variable v, l'ensemble des valeurs prises par v est calculé au moment de la lecture de la trace. Si $[\![v_i]\!]$ est la valeur de v à l'état σ_i, alors, l'ensemble D_v des valeurs de v est :

$$D_v = \bigcup_{i \in [\![0,|\sigma|-1]\!]} [\![v_i]\!]$$

6.3.5.4 Détermination des N-uplets de valeurs

Le calcul des ensembles de valeurs des variables paramétrées étant effectué, il faut générer tous les N-uplets de (variable,valeur) possibles.

Soit une propriété faisant intervenir N variables paramétrées $x_0, ..., x_{N-1}$ (deux variables x_i et x_j peuvent avoir un même ensemble de définition). Soient $D_0, ...D_{N-1}$ les ensembles de définition de ces variables, ν un N-uplet. Alors :

$$\nu \in \quad \times_{i=0}^{N-1} D_i \tag{6.16}$$

$$Nb_{N-uplet} = \quad \prod_{i=0}^{N-1} card(D_i) \tag{6.17}$$

Il est donc nécessaire de faire attention à ce que la multiplication du nombre d'automates ne soit pas rédhibitoire pour la vérification de la propriété temporelle. En effet, plus $Nb_{N-uplet}$ est élevé, plus le temps de vérification sera important.

6.3.5.5 Multiplication du nombre d'automates

La propriété temporelle étant paramétrée, Ltl2ba est utilisé pour générer un automate de Büchi générique également. Cette méthode ne fait appel à Ltl2ba qu'une seule fois. Ensuite, c'est l'automate de Büchi qui va être instancié pour chaque N-uplet existant.

Chaque automate est donc exécuté ensuite avec les données de son propre N-uplet. Les automates étant décorrélés, un résultat est donc donné pour chaque couple et une erreur détectée par un automate n'arrête pas l'exécution des autres automates.

6.3.6 Relancer l'automate après la détection d'une erreur

Lorsqu'une propriété est violée (confer section 6.4.2), le reste de la trace n'est pas exécuté. Pour certaines propriétés, il pourrait être intéressant de tenter de savoir si le reste de la trace présente des anomalies également.

Exemple 37 *Soit la propriété $\Box\, p$. Si la propriété p est violée dans un état donné de la trace, il est impossible de savoir ce qu'il en est pour le reste de la trace tant que le bug n'a pas été corrigé.*

La tentative d'exécution de la trace peut fournir des informations intéressantes au vérifieur, notamment si le bug ne se produit qu'une fois lors de l'exécution ou s'il est répétitif. Cela peut l'aider à cibler l'origine du bug.

Pour poursuivre la vérification malgré la détection d'une faute, à un état e donné, il est nécessaire de réinitialiser l'automate de Büchi, à savoir continuer l'exécution avec l'état initial de l'automate comme point d'entrée. En effet, l'automate peut se trouver dans un état où il n'y aura plus jamais de transition dont la formule est vraie. Il est donc préférable de le réinitialiser.

L'état e posant problème est également sauté et l'exécution de l'automate redémarre à l'état $e+1$, pour éviter de bloquer l'exécution indéfiniment à cet état. Cela revient donc à tronquer le début de la trace et à vérifier la propriété sur le reste de celle-ci.

Relancer l'automate après la détection d'une erreur sur une propriété de sûreté ($\Box\, p$) permet de voir, si la propriété est respectée sur le reste de la trace et d'isoler l'état où la propriété a été violée. Si l'erreur se répète dans la suite de la trace, et qu'il s'agit toujours d'un état collecté à partir d'un même point d'observation, cela permet alors de cibler précisément le problème. Il en est de même si l'erreur n'est pas répétée par la suite.

Pour une propriété de vivacité ($\Diamond\, p$), relancer l'exécution de l'automate de Büchi sur le reste de la trace n'a pas d'utilité, puisque si la propriété est violée, elle le sera au dernier état de la trace d'exécution.

L'algorithme de vérification étant défini pour la vérification de la trace jusqu'au dernier état, il faut maintenant définir l'algorithme de vérification pour la fin de la trace.

6.4 Traces finies

La logique temporelle linéaire est une logique sur des mots de longueur infinie. Or les traces étudiées sont issues de programmes qui se terminent ou sont stoppés et sont analysées a posteriori. Ce sont donc des traces finies.

Pour pallier ce problème, une solution classique, qui a été choisie ici, consiste à boucler sur le dernier état de la trace.

Définition 28 (Trace finie) *Une trace finie σ est une séquence finie d'états, de taille $|\sigma|$ considérée strictement positive. Il y a donc au minimum un état dans la*

trace. Le i^{me} état de cette trace est σ_i. Une sous-trace de σ pour $(i,j) \in [\![0, |\sigma| - 1]\!]^2, i \leq j$ s'écrit σ_i^j.

Exemple 38 *Soit la trace $\sigma = a_0a_1a_2a_3a_4$.*
Alors $\sigma_2 = a_2$ et $\sigma_1^3 = a_1a_2a_3$.

La première étape est de définir un algorithme traitant la fin d'une trace. Pour la seconde étape, il convient d'analyser les conséquences de cette solution sur l'évaluation des propriétés temporelles.

6.4.1 Algorithme de fin de trace

La gestion de fin de trace est nécessaire lorsqu'aucune erreur ne s'est produite durant l'exécution de l'automate avec la trace. À ce niveau de l'exécution, l'automate de Büchi possède un ensemble d'états courants $\mathcal{E}(|\sigma|-1)$. Pour rappel, il a été décidé de boucler sur le dernier état de la trace d'exécution (σ).

Il n'est pas possible de continuer l'exécution avec les algorithmes précédents, puisque la vérification ne terminerait jamais.

La définition 23 stipule qu'un mot est accepté si l'automate de Büchi $\mathcal{B}$ passe infiniment souvent par un état quelconque de l'ensemble des états finis de $\mathcal{B}$. En fin de trace, puisqu'il y a bouclage sur le dernier état de la trace, les formules de chaque transition de $\mathcal{B}$ ont toujours la même valeur (figure 6.19). La seule possibilité est que $\mathcal{B}$ décrive un cycle.

Définition 29 (Cycle dans un automate ou un graphe) *Un cycle dans un automate ou un graphe $\mathcal{G}$ est une séquence d'états de $\mathcal{G}$ se répétant infiniment souvent. Autrement dit, $\mathcal{G}$ dispose d'états fortement connexes.*

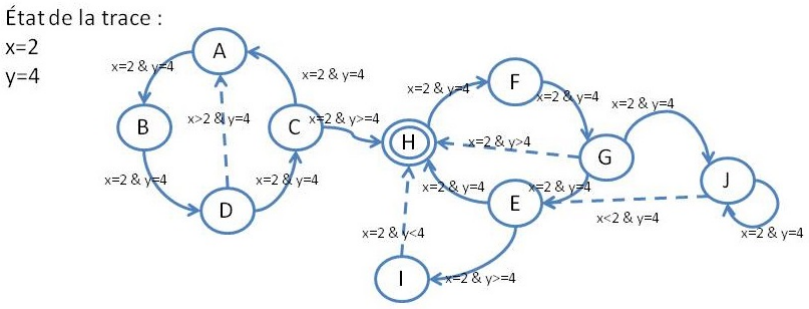

FIGURE 6.19 – Exemple d'automate de Büchi à la fin de la trace

La solution retenue consiste à faire appel à la théorie des graphes, pour s'assurer d'une part que l'algorithme de fin de trace est correct, et d'autre part que l'algorithme termine. L'automate de Büchi, en fin de trace est assimilable à un graphe orienté $\mathcal{A}$, en remplaçant les transitions dont la formule est vraie par des arcs orientés et en supprimant les transitions dont la formule est fausse.(figure 6.20).

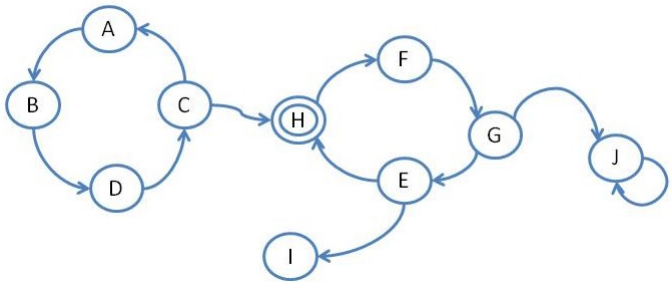

FIGURE 6.20 – Graphe issu de l'automate de Büchi de la figure précédente

Pour qu'au automate de Büchi reconnaisse un mot infini, il doit disposer d'un cycle contenant au moins un état acceptant. Donc, le graphe $\mathcal{A}$ issu de l'automate de Büchi doit disposer des mêmes cycles.

Il faut donc déterminer un graphe équivalent acyclique, à savoir : un DAG (direct acyclic graph). Pour cela, le calcul des composantes fortement connexes de $\mathcal{A}$ va générer un nouveau graphe $\mathcal{A}_\delta$. Chaque état contient un ensemble d'états fortement connexes de l'automate de Büchi.

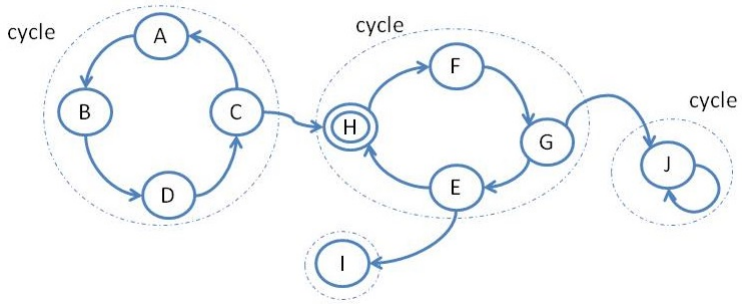

FIGURE 6.21 – Transformation d'un graphe en DAG par la méthode des composantes fortement connexes

Une fois $\mathcal{A}_\delta$ calculé, l'étape suivante consiste alors pour chaque élément ε de $\mathcal{E}(|\sigma|-1)$, de partir de l'état du dag contenant ε, de déterminer les états du dag accessibles, et de vérifier quels états du dag possèdent des états d'acceptation de l'automate de Büchi.

Si un état de $\mathcal{A}_\delta$ accessible depuis ε ne contient qu'un seul état de l'automate de Büchi, et si cet état est final, il est nécessaire de s'assurer que cet état boucle sur lui même. S'il ne boucle pas sur lui-même, il faut poursuivre le parcours de $\mathcal{A}_\delta$. Dans le cas contraire, un cycle avec un état final est trouvé, et la propriété est vraie.

Si un état de $\mathcal{A}_\delta$ est accessible à partir de l'état contenant ε, et s'il contient strictement plus d'un état, alors la propriété est vérifiée, puisque les états de l'automate de Büchi contenus dans l'état de $\mathcal{A}_\delta$ forment un cycle.

Dans le cas où, pour tout ε de $\mathcal{E}(|\sigma|-1)$ aucun état accessible du $\mathcal{A}_\delta$ ne contient

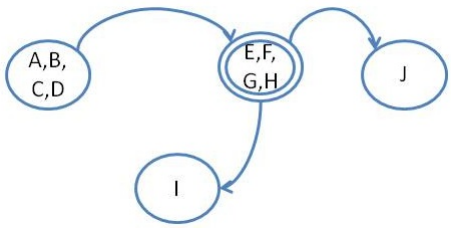

FIGURE 6.22 – DAG obtenu à partir du graphe précédent.

d'état final, (état de $\mathcal{A}_\delta$ contenant un état final de $\mathcal{A}$ bouclant sur lui même, ou plus d'un état strictement, dont l'un est final) la propriété n'est pas vérifiée. Le schéma 6.21 résume, à partir d'un exemple d'automate de Büchi (ici ne sont représentés que les transitions valides et les états atteignables), la méthode de transformation d'un graphe en dag. Le schéma 6.22 expose le graphe des composantes fortement connexes $\mathcal{A}_\delta$ obtenu à partir d'un graphe $\mathcal{A}$.

La démarche de l'algorithme se résume donc à :

1. déterminer les composantes fortement connexes de l'automate de Büchi de la propriété, assimilé à un graphe,
2. définir un dag, graphe équivalent à l'automate de Büchi, à partir du calcul des composantes fortement connexes,
3. parcourir le dag, pour chaque élément de $\mathcal{E}(|\sigma|-1)$,
4. déterminer pour chaque état du dag accessible depuis un élément ε de $\mathcal{E}(|\sigma|-1)$ s'il contient un état final de l'automate de Büchi,
5. si un état du dag accessible contient un état final alors la propriété est vérifiée. Dans le cas contraire, la propriété n'est pas vérifiée.

Proposition 6 *Cet algorithme termine.*

Idée de la preuve : L'algorithme consiste à définir le dag d'un graphe cyclique orienté, puis à parcourir ce dag en profondeur sans passer deux fois par un même état.

Proposition 7 *Cet algorithme est correct.*

Idée de la preuve : L'algorithme consiste à détecter des cycles dans l'automate de Büchi, et d'en trouver un disposant d'un état acceptant.

L'automate de Büchi est assimilable à un graphe $\mathcal{A}$. Les cycles sont détectés sur le graphe acyclique $\mathcal{A}_\delta$ déduit de $\mathcal{A}$ par calcul des composantes fortement connexes (ensemble d'états décrivant un cycle). Un état de $\mathcal{A}_\delta$ étant l'union de plusieurs états de $\mathcal{A}$, il suffit que cet ensemble contienne :

- soit un unique état, acceptant et bouclant sur lui-même,
- soit plus d'un état, dont l'un d'entre eux au moins soit acceptant.

Preuve : Soit $\mathcal{A}$ l'automate de Büchi non déterministe. Les états courants de l'automate sont réunis dans l'ensemble $\mathcal{E}(|\sigma|-1)$. Puisque la vérification d'une propriété temporelle consiste à boucler sur le dernier état de la trace, un sous-ensemble des transitions de $\mathcal{A}$ ont leur formule toujours évaluée à vrai. Le complémentaire de ce sous-ensemble sur l'ensemble des transitions de $\mathcal{A}$ ont leur formule toujours évaluée à faux. L'automate de Büchi est donc assimilable à un graphe orienté.

Soit $\mathcal{A}_\delta$ le graphe des composantes fortement connexes de $\mathcal{A}$. Par construction, chaque état de $\mathcal{A}_\delta$, contient un sous-ensemble des états de $\mathcal{A}$.

Soit la fonction $cfc_\mathcal{A}$ qui retourne pour chaque état de $\mathcal{A}$ l'état du dag associé. Soit la fonction $cfc^f_\mathcal{A}$ la fonction qui pour un état ε_d de $\mathcal{A}_\delta$ retourne *true* si un état de l'automate de Büchi contenu dans ε_d est final. Pour chaque élément de $\mathcal{E}(|\sigma|-1)$, les états accessibles de $\mathcal{A}_\delta$ à partir de $cfc_\mathcal{A}(\varepsilon)$ sont calculés. Soit $\mathcal{S}_\varepsilon$ cet ensemble d'états accessibles à partir de ε. Soit $\mathcal{B}_{nb}$ le nombre d'états de l'automate de Büchi compris dans l'état donné de $\mathcal{A}_\delta$ de l'automate de buchi. Un état e de l'automate de Büchi bouclant sur lui même s'écrit $loop(e)$

Si $\forall \varepsilon \in \mathcal{E}(|\sigma|-1), \forall \varepsilon_d \in \mathcal{S}_\varepsilon, cfc^f_\mathcal{A} = false$, alors la propriété est fausse, par construction du graphe des composantes fortement connexes.

Si $\forall \varepsilon \in \mathcal{E}(|\sigma|-1), \exists \varepsilon_d \in \mathcal{S}_\varepsilon, cfc^f_\mathcal{A} = true$, $\mathcal{B}_{nb}(\varepsilon_d) > 1$ alors la propriété est vraie, par construction du graphe des composantes fortement connexes.

Si $\forall \varepsilon \in \mathcal{E}(|\sigma|-1), \exists \varepsilon_d \in \mathcal{S}_\varepsilon, cfc^f_\mathcal{A} = true$, $\mathcal{B}_{nb}(\varepsilon_d) = 1$ et $\forall e \in \varepsilon_d, loop(e)$, alors la propriété est vérifiée.

Si $\forall \varepsilon \in \mathcal{E}(|\sigma|-1), \exists \varepsilon_d \in \mathcal{S}_\varepsilon, cfc^f_\mathcal{A} = true$, $\mathcal{B}_{nb}(\varepsilon_d) = 1$ et $\forall e \in \varepsilon_d, \neg\, loop(e)$, alors la propriété n'est pas vérifiée ■.

Un algorithme pour gérer la vérification d'une propriété en fin de trace est donc établi. Cependant, cet algorithme n'est pas suffisant.

6.4.2 Des formules qui n'en font qu'à leur tête

Le fait de vérifier une propriété temporelle sur une trace finie, rendue infinie en bouclant sur le dernier état, a des conséquences sur la satisfiabilité d'une propriété. Par exemple, pour des traces d'origine finie, certaines propriétés seront des tautologies, tandis que sur des traces d'origine infinie, ces propriétés peuvent être fausses.

6.4.2.1 Satisfiabilité différente selon la nature de la trace.

Les exemples suivants présentent des propriétés dont la satisfiabilité varie avec l'origine (finie ou infinie) de la trace.

Exemple 39 *Soit la propriété de vivacité* $\Diamond\ (\Box\ (A) \vee \Box\ (\neg\ A))$*, qui signifie qu'à partir d'un état futur la propriété A sera soit toujours vraie soit toujours fausse.*

- *Dans le cas des traces finies, cette formule est une tautologie du fait qu'on boucle sur le dernier état. (6.23 a)*
- *Dans le cas des traces infinies, un mot constitué à l'infini d'une alternance de A et de* $\neg\ A$ *invalide cette propriété. (6.23 b)*

Inversement, la négation de cette propriété ($\Box\ (\Diamond\ (\neg\ A)\ \wedge\Diamond\ (A))$) a également une satisfiabilité changeante.

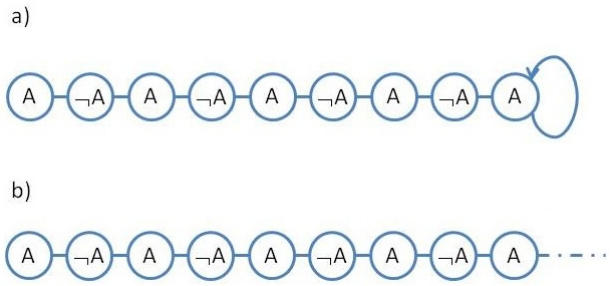

FIGURE 6.23 – Deux traces d'origines infinies et finies

Au final, à travers cet exemple, le constat est que selon la nature de la trace la satisfiabilité d'une propriété peut être variable.

6.4.2.2 Interprétation de l'évaluation d'une formule

Définition 30 (Évaluation d'une formule) *Soit une propriété ϕ à vérifier sur une trace d'exécution σ. $\mathcal{A}_\phi$ est l'automate de Büchi associé à ϕ. Soit un σ_i un état de la trace qui n'est pas le dernier état. Alors trois cas sont possibles.*

- *Si l'ensemble des états de $\mathcal{A}_\phi$ accessibles depuis σ_i est l'ensemble vide, alors la propriété est violée.*
- *Lorsque le dernier état de σ est atteint, il y a bouclage sur celui-ci. $\mathcal{A}_\phi$ est alors considéré comme un graphe. Les transitions dont l'évaluation de la formule est fausse sont retirées. Le dag de ce graphe est calculé et il est parcouru pour déterminer si un état acceptant est infiniment souvent accessible (voir définition 22). Deux cas sont possibles :*
  - *Un état acceptant de $\mathcal{A}_\phi$ est infiniment souvent accessible. La propriété est donc satisfaite jusque là. Elle pourrait cependant être violée par la suite (Propriétés de sûreté et de vivacité).*
  - *Aucun état acceptant de $\mathcal{A}_\phi$ n'est infiniment souvent accessible. La propriété n'est pas satisfaite. Elle pourrait cependant être satisfaite par la suite (Propriété de vivacité).*

Le choix dans cette thèse est de proposer une approche pragmatique qui consiste à fournir à l'utilisateur des informations statistiques qui l'aideront à interpréter le résultat de la vérification effectuée dans les deux derniers cas.

6.4.3 Approche pragmatique

Pour les propriétés évaluées à faux, une approche consiste à définir une métrique sur la trace pour savoir jusqu'à quel état la propriété s'est avérée vraie.

S'il existe un ensemble non vide de préfixes pour laquelle la propriété s'est avérée vraie, alors la taille du préfixe le plus long, noté σ_{max}, sera renvoyée. Si cet ensemble est vide (cas des propriétés de vivacité), alors la valeur 0 sera renvoyée.

Cette approche fournit une indication supplémentaire à l'utilisateur pour déterminer à partir de quand la propriété est évaluée à faux. Elle est d'autant plus utile que la trace est longue. De plus, à un état i donné de la trace σ, l'ensemble des variables modifiées $\mathcal{V}_m$ est connu, de part la construction de la trace. Si BL_X est l'ensemble des points du code source où la variable X est modifiée alors l'erreur est apparue dans l'un des points du code source de l'ensemble $\bigcap_{X \in \mathcal{V}_m} BL_X$.

6.4.4 Détection de patrons

La métrique définie dans la sous-section précédente apporte des informations à l'utilisateur lorsqu'une propriété est fausse. Cependant, dans le cas où une propriété est vraie, aucune information supplémentaire n'est fournie. Puisque donner davantage d'informations sur une propriété quelconque s'avère difficile, une approche sur des propriétés particulières a été étudiée.

Ainsi, pour une propriété particulière, il est possible de donner des informations statistiques. En généralisant, si le patron de propriété est connu, il devient possible de donner un certain nombre d'informations statistiques.

Exemple 40 *Soit la propriété* $\Box \Diamond (ressource > 0)$. *Elle respecte le patron générique :* $\Box \Diamond p$, *où p est une variable booléenne. Une information utile pour le vérifieur est le nombre de fois où la propriété p (ici* $ressource > 0$*) est vraie. En effet, cette propriété s'avère vraie à partir du moment ou p est vrai dans le dernier état de la trace, étant donné que la trace boucle sur le dernier état. Avec des informations statistiques données sur la trace, sans considérer l'état bouclant, l'utilisateur se rendra compte rapidement si la propriété p ne s'est révélée vraie que sur le dernier état de la trace ou non.*

Inversement, si le dernier état de la trace ne vérifie pas la propriété p, mais qu'elle se vérifie presque dans tout le reste de la trace, l'utilisateur aura ainsi plus de moyens pour analyser le résultat obtenu.

Avant de présenter les différents patrons étudiés, il est important de définir précisément un patron.

Définition 31 (Patron de propriété) *Un patron de propriété LTL est une formule de LTL liant les propositions* $p_0 ... p_n$, *où pour tout i de* $[\![0; n]\!]$, p_i *est une proposition de la logique propositionnelle.*

Exemple 41 *Soit le patron :* $\mathcal{P} = \Box (P \Rightarrow \Diamond Q)$ *:*

- $\Box (a > 0 \Rightarrow \Diamond (b < 0))$ *est conforme au patron* $\mathcal{P}$ *;*
- $\Box (a > 0 \Rightarrow \Diamond (\Box\, b < 0))$ *n'est pas conforme au patron* $\mathcal{P}$.

En d'autres termes, pour toute variable propositionnelle, il ne sera pas possible de la remplacer par une formule contenant des opérateurs temporels. Cette limitation est due au fait qu'une formule LTL est transformée en automate de Büchi et que dès qu'un opérateur temporel est ajouté, l'automate de Büchi s'en trouve différent.

6.4.4.1 Deux méthodes de calcul d'informations statistiques

Deux approches différentes sont possibles pour calculer les informations statistiques.

La première approche consiste à définir pour chaque patron une fonction de calcul qui s'intègre dans l'algorithme de vérification, à chaque fois qu'un pas de calcul est terminé. Selon les propriétés à étudier, des compteurs sont alors incrémentés.

Exemple 42 *Pour le patron* $\Box \Diamond p$*, il est utile d'avoir le nombre de fois où* p *est vrai. Dans ce cas, la fonction consistera à incrémenter un compteur à chaque pas de calcul où la propriété* p *est vraie.*

L'inconvénient de cette méthode est qu'il faut de ce fait implémenter pour chaque patron la fonction qui convient. Cela entraînera donc une modification de l'outil pour chaque nouveau patron. Dans le contexte industriel, cela nécessiterait de requalifier l'outil pour chaque nouveau patron, ce qui n'est pas envisageable.

La seconde approche consiste à définir un automate pour chaque patron. À chaque fois qu'une transition est activée (conditionnée par une formule booléenne), une série d'actions associée à cette transition est effectuée.

C'est l'approche qui a été privilégiée, puisqu'une fois l'automate clairement défini, il est possible d'écrire un automate pour chaque nouveau patron en dehors de l'outil. L'outil n'a besoin alors que d'un interpréteur d'automates pour fonctionner.

Cet automate déterministe est exécuté simultanément avec l'automate de Büchi. Après avoir calculé l'ensemble des états accessibles de l'automate de Büchi, l'état accessible de l'automate statistique est donc calculé et les opérations sur les compteurs sont appliquées si nécessaire.

Étant donné que c'est à l'utilisateur de définir cet automate, il est plus prudent d'exécuter l'automate de Büchi issu de Ltl2ba en parallèle. Ainsi, même si une erreur s'est glissée dans l'automate statistique, cela n'a de répercussions que sur les informations statistiques et pas la vérification qualitative. En outre, selon les informations que l'utilisateur souhaite calculer, il n'est pas nécessaire de disposer d'un automate statistique aussi complet que l'automate de Büchi associé.

Enfin cette approche est modulaire et incrémentale.

6.4.4.2 Automate de calcul d'informations statistiques

L'automate de calcul d'informations statistiques est défini de la manière suivante :

Définition 32 (Automate statistique) *Soit un alphabet Σ. Soit un ensemble $\mathcal{C}$ de variables entières. Ces variables sont des compteurs. Soit un ensemble d'actions $\Lambda_{\mathcal{C}}$ sur les variables de $\mathcal{C}$. Les actions sont des opérations liées à des compteurs :*

- *ne rien faire*
- *affectation d'une valeur de compteurs/constantes à un compteur. La valeur peut être :*
  - *somme de compteurs/constantes*
  - *soustraction de compteurs/constantes*
  - *multiplication de compteurs/constantes*
  - *division de compteurs/constantes*
  - *minimum de compteurs/constantes*
  - *maximum de compteurs/constantes*

Un automate de calcul statistique est un quintuplet $A = \{Q, \Sigma, \rightarrow, q_0, F, \mathcal{C}_0\}$ avec :

- *Q un ensemble d'états*
- *Σ un alphabet*
- *$\rightarrow \subseteq Q \times \Sigma \times \Lambda_{\mathcal{C}} \times Q$ une relation de transition*
- *q_0 un état initial*
- *$\mathcal{C}_0 : \mathcal{C} \rightarrow \mathbb{Z} \cup \varnothing$ l'ensemble des valeurs initiales pour chaque variable de $\mathcal{C}$.*

$\varnothing$ est une valeur neutre, notamment employée pour le calcul de minimum. $\forall n \in \mathbb{Z}$, $min(\varnothing, n) = n$

Pour chaque patron étudié, les automates de calcul d'informations statistiques associés aux patrons étudiés sont donnés ci-dessous.

6.4.4.3 Les patrons étudiés

Les patrons étudiés sont les suivants :

- $\Box\Diamond\, p$
- $\Box\, (p \Rightarrow q)$
- $\Box\, (p \Rightarrow \Diamond\, q)$

$\Box\Diamond\, p$: Cette propriété signifie que le prédicat p doit être vrai infiniment souvent, mais pas forcément tout le temps.

L'automate statistique de ce patron est donné par la figure 6.24

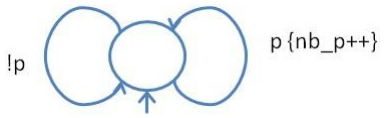

FIGURE 6.24 – Automate du patron $\Box\Diamond\, p$

Le compteur nb_p est initialisé à zéro et compte le nombre d'états dans la trace où la propriété p est vraie.

$\square\ (p \Rightarrow q)$: Cette propriété signifie que pour tout état de la trace, si p est vrai alors q est vrai. Le fait d'avoir ce patron plutôt que $\square\ (p)$ permet d'obtenir des informations sur pourquoi la proposition $p \Rightarrow q$ est vraie. Il y a en effet deux cas : p est faux ou p est vrai.

L'automate statistique de ce patron est donné par la figure 6.25. L'automate est le même que pour le patron $\square\Diamond\ p$ puisqu'il calcule également le nombre de fois où la propriété p est vraie. Ainsi, le nombre d'états dans la trace où $p \Rightarrow q$ avec p vrai est connu. Il n'est pas nécessaire d'ajouter d'information sur le fait que q soit vrai ou non, puisque si lorsque p est vrai, q n'est pas vrai, la vérification s'arrête.

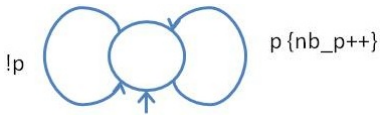

FIGURE 6.25 – Automate du patron $\square\ (p \Rightarrow q)$

Le compteur nb_p est initialisé à zéro et compte le nombre d'états dans la trace où la propriété p est vraie.

$\square\ (p \Rightarrow\Diamond\ q)$: Cette propriété signifie qu'à chaque fois que p survient alors nécessairement dans le futur, q se doit se produire.

Soit une trace vérifiant la propriété $\square\ (a \Rightarrow\Diamond\ b)$. Pour que la propriété soit vraie, à chaque fois que a se produit, b se produit un jour. Il peut cependant arriver que a se produise plusieurs fois avant que b se produise, dans ce cas pour une séquence de "a [^b]* b" le premier état où a se produit est appelé premier a. Dès que b s'est produit, une nouvelle séquence de "a [^b]* b" peut exister et donc un nouveau premier a également. La figure 6.26 illustre cette définition.

FIGURE 6.26 – Premier a, définition par l'exemple.

La figure 6.27 définit l'automate statistique.

Le tableau 6.4 présente les compteurs et leur valeur initiale.

La valeur initiale $\varnothing$ est utilisée pour initialiser les compteurs de minimaux. En effet, si la valeur zéro était utilisée, le minimum de n'importe quel nombre positif avec zéro étant toujours zéro, ce compteur ne calculerait rien.

Les cinq premiers compteurs sont issus de besoins industriels réels (nombre d'occurrences de p, de q, de $p \wedge q$ et de $p \Rightarrow\Diamond\ q$). Les autres sont des compteurs de démonstration, dont le but est de fournir un exemple de ce qui peut être calculé. Ils ont entre autres servi à donner le nombre de fois où un port a été utilisé lors d'expérimentations sur le logiciel 3 (confer chapitre 8).

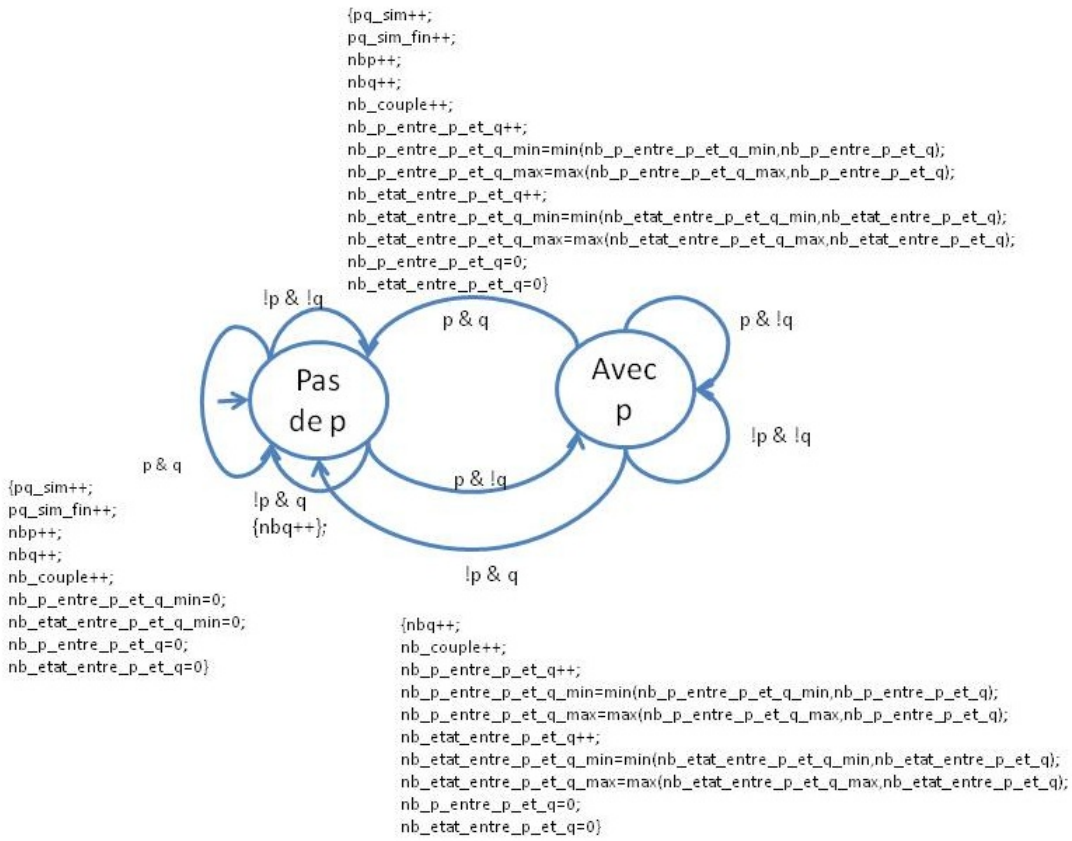

FIGURE 6.27 – Automate du patron $\Box\ (p \Rightarrow \Diamond\ q)$

6.4.4.4 Méthode de détection de patron

Pour détecter si une formule est conforme à un patron donné, la solution retenue est d'utiliser un arbre de recherche.

Chaque nœud de cet arbre correspond à un opérateur LTL ou à un opérateur booléen ($\neg$, $\vee$, $\wedge$, $\Rightarrow$), ou à une variable propositionnelle. Si l'opérateur est binaire, la différence est faite entre les fils du premier opérande et les fils du second opérande. Le nœud contient, en outre, l'ensemble des patrons encore atteignables à ce niveau de l'arbre.

La figure 6.28 présente l'arbre de recherche pour les trois patrons étudiés.

Il suffit alors de comparer cet arbre à celui de la formule. Si la formule possède un opérateur temporel qui n'est pas dans l'arbre de recherche, alors aucun patron n'est trouvé.

Si l'arbre de recherche est parcouru jusqu'aux feuilles, que l'intersection des ensembles de patron de chaque feuille conduit à un seul patron et que le reste de la formule à parcourir n'a pas d'opérateur temporel, alors le patron est celui de l'intersection des ensembles.

TABLEAU 6.4 – Utilité et valeur initiale des compteurs pour $\Box\,(p \Rightarrow \Diamond\, q)$

Compteur (valeur initiale)	Utilité
nbp (0)	nombre d'états vérifiant un premier p
nbq (0)	nombre d'états vérifiant q
pq_sim (0)	nombre d'états vérifiant p et q sans premier p
pq_sim_fin (0)	nombre d'états vérifiant p et q avec premier p
nb_couple (0)	nombre de $p \Rightarrow \Diamond\, q$ avec p vrai
$nb\_etat\_entre\_p\_et\_q$ (0)	nombre d'états entre un premier p et q
$nb\_etat\_entre\_p\_et\_q\_min$ ($\varnothing$)	nombre min. d'états entre un premier p et q
$nb\_etat\_entre\_p\_et\_q\_max$ (0)	nombre max. d'états entre un premier p et q
$nb\_p\_entre\_p\_et\_q$ (0)	nombre de p entre un premier p et q
$nb\_p\_entre\_p\_et\_q\_min$ ($\varnothing$)	nombre min. de p entre un premier p et q
$nb\_p\_entre\_p\_et\_q\_max$ (0)	nombre max. de p entre un premier p et q

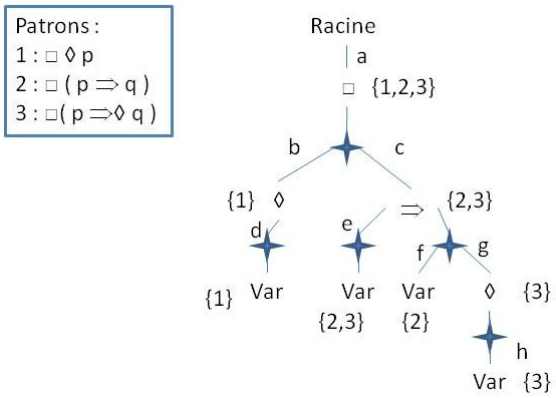

FIGURE 6.28 – Arbre de recherche des trois patrons étudiés

La recherche privilégie une recherche d'opérateur correspondant à celui de la formule, avant d'essayer une variable propositionnelle (branches (f,g) et e de l'arbre par exemple).

Exemple 43 *Soit la formule :* $\Box\Diamond\,(ressource > 0 \wedge temps\_calcul > 0)$.

Le parcours de l'arbre de recherche avec cette formule sera donc : a,b,d. Arrivé à la feuille var de d, il faut s'assurer que dans le reste de la formule : $ressource > 0 \wedge temps\_calcul > 0$, *il n'y a pas d'opérateur temporel, ce qui est le cas. La formule vérifie le patron* $\Box\Diamond\, p$, *avec* $p = (ressource > 0 \wedge temps\_calcul > 0)$

Exemple 44 *Soit la formule :* $\Box\,(ressource < 0 \Rightarrow (\Diamond\,(temps\_calcul = 0) \wedge \Diamond\,(temps\_calcul > 0)))$.

Le parcours de l'arbre de recherche avec cette formule sera donc : a,c,e,f. Arrivé à la feuille var de f, il faut s'assurer que dans le reste de la formule : $\Diamond\,(temps\_calcul = 0) \wedge \Diamond\,(temps\_calcul > 0)$, *il n'y a pas d'opérateur temporel, ce qui n'est pas le cas. Aucun patron n'a donc été détecté pour cette formule.*

Lorsqu'un patron est détecté, lors de la vérification d'une propriété temporelle, pour chaque itération de l'algorithme défini dans la section 6.3.1.6, un pas de calcul est effectué sur l'automate statistique :

La démarche du cas nominal est la suivante :

1. L'état suivant de la trace est chargé, s'il existe. Lors du chargement, les compteurs de modifications sont incrémentés à la volée, lorsqu'une variable change de valeur, ainsi que pour les temps de modifications. La variable globale temps τ est incrémentée systématiquement.
2. Si l'état suivant de la trace n'existe pas, l'algorithme nominal se termine et l'algorithme de fin de trace prend le relais.
3. La formule de chaque transition pour chaque état de l'automate appartenant à l'ensemble des états courants est évaluée. Si la formule est vraie, l'état pointé par la transition est ajouté au nouvel ensemble des états courants de l'automate.
4. La formule de chaque transition de l'état courant de l'automate statistique [7] est évaluée [8]. Les actions de la transition t, dont la formule est vraie, sont effectuées. L'état courant de l'automate statistique devient l'état pointé par t.
5. L'ensemble des états suivants devient l'ensemble des états courants. Dans cet ensemble, chaque état est unique, alors que deux transitions différentes peuvent conduire à un même état. Cela évite une explosion combinatoire du nombre d'états et donc du nombre de transitions à vérifier. La figure 6.7 schématise la simplification
6. Si l'état courant est vide, alors la propriété est fausse. L'algorithme se termine.
7. Sinon, l'algorithme recommence à l'étape 1.

Conclusion du chapitre

L'objectif de ce chapitre était de vérifier des traces d'exécution de programme construites en adéquation avec les hypothèses de la section 5.3, à savoir que les propriétés temporelles appartiennent au fragment $\mathcal{L}(\mathcal{U})$ ou sont des séquences faibles ou étendues.

Pour ce faire, après avoir rappelé les moyens et algorithmes classiques pour vérifier une propriété temporelle, il a été choisi, après avoir testé deux méthodes avec NuSMV, de transformer la formule LTL à vérifier en automate de Büchi à l'aide de Ltl2ba, puis d'exécuter ce dernier avec la trace d'exécution du programme à vérifier.

Ensuite, il a été décidé d'implémenter, en se basant sur Ltl2ba pour la transformation en automate de Büchi, un outil spécifique aux besoins industriels pour

7. Cet état est unique, l'automate statistique étant déterministe.
8. Une unique transition a sa formule évaluée à vraie.

pouvoir apporter des améliorations par rapport à l'outil NuSMV. Parmi ces améliorations, il y a la vérification d'une propriété temporelle purement orientée vers le passé par lecture de la trace d'exécution à l'envers, et la gestion de fin de trace.

La difficulté à établir un jugement concernant la satisfiabilité d'une propriété temporelle sur une trace finie, est compensée par l'ajout d'informations additionnelles, notamment par la donnée du numéro du dernier état où la propriété était vraie et par la définition d'informations spécifiques à des patrons de propriétés données.

Conclusion de la partie

L'objectif de cette partie était de présenter la démarche adoptée pour vérifier une propriété temporelle sur une trace d'exécution donnée.

Le chapitre 4 a permis de formaliser les propriétés temporelles à vérifier, en s'appuyant sur une étude basée sur quelques logiciels industriels. Pour des raisons de confidentialité, cette étude n'a pas pu être davantage détaillée. Une juxtaposition de LTL et d'un langage d'expressions régulières pour les séquences d'événements a été définie.

Le chapitre 5 a proposé une approche basée sur l'analyse statique de programmes pour générer les points d'observation nécessaires à la génération d'une trace d'exécution du programme étudié. La définition des points d'observation dépend de la propriété étudiée. A chaque affectation d'une variable de la propriété temporelle, un point d'observation est défini pour collecter la nouvelle valeur. Cette approche permet de minimiser la taille de la trace d'exécution.

L'analyse statique du programme avec Frama-C permet de trouver, pour une variable donnée, l'ensemble des points de programme où celle-ci est utilisée. Comme il s'agit d'une analyse sémantique de programme, les alias de la variable sont également détectés. Un filtre syntaxique permet de ne conserver que les points d'observation où la variable subit une affectation.

La trace générée à partir de ces points d'observation étant partielle, la correction de la vérification de la propriété temporelle sur cette trace a été faite et a montré que seules les propriétés appartenant au fragment $\mathcal{L}(\mathcal{U})$ peuvent être traitées (à savoir les propriétés utilisant les opérateurs temporels $\{\Box, \Diamond, \mathcal{U}\}$ et les séquences faibles et étendues, qui s'écrivent également à partir de ces mêmes opérateurs temporels).

Le chapitre 6 a proposé une approche permettant de vérifier une propriété temporelle sur une trace d'exécution finie. Pour disposer rapidement d'un outil fonctionnel, il a été décidé d'utiliser l'outil Ltl2ba. En conséquence, il s'est avéré nécessaire de boucler sur le dernier état de la trace pour la rendre infinie. Pour pallier cet inconvénient, une approche pragmatique consistant à donner des informations quantitatives sur la trace d'exécution a été choisie :

- lorsque la propriété est fausse : donner le plus grand préfixe de la trace où la propriété est vraie ;
- lorsque la propriété est un patron connu : donner des informations statistiques en fonction de ce patron.

Cette approche ne permet de vérifier une propriété que sur la trace donnée. Les limites de cette méthode sont donc celles du test classique, à savoir :

- une propriété violée en milieu de trace est fausse ;

- en fin de trace :
  - une propriété satisfaite peut éventuellement être violée par la suite ;
  - une propriété n'étant pas satisfaite pourrait l'être par la suite.

Les informations quantitatives, en plus de compenser le bouclage sur le dernier état, ont pour but d'aiguiller l'opérateur pour déterminer dans quel cas il se situe.

Troisième partie

Outil et expérimentations

Chapitre 7

Outillage

Sommaire

L'objectif de ce chapitre est de détailler l'architecture des prototypes implémentant l'approche définie dans la partie précédente. La section 7.1 présentera une vue globale de l'association entre l'outil de génération de scripts d'observation Breakpointer et l'outil de vérification AnTarES. La section 7.2 détaillera l'implémentation de Breakpointer l'outil de génération de scripts pour conduire une exécution de programme, afin de générer une trace d'exécution. Enfin, la section 7.3 sera consacrée à l'outil AnTarES, permettant d'analyser une trace d'exécution pour une propriété temporelle donnée.

7.1 La chaîne outil de vérification

Le schéma 7.1 présente le fonctionnement global de la chaîne d'outil.

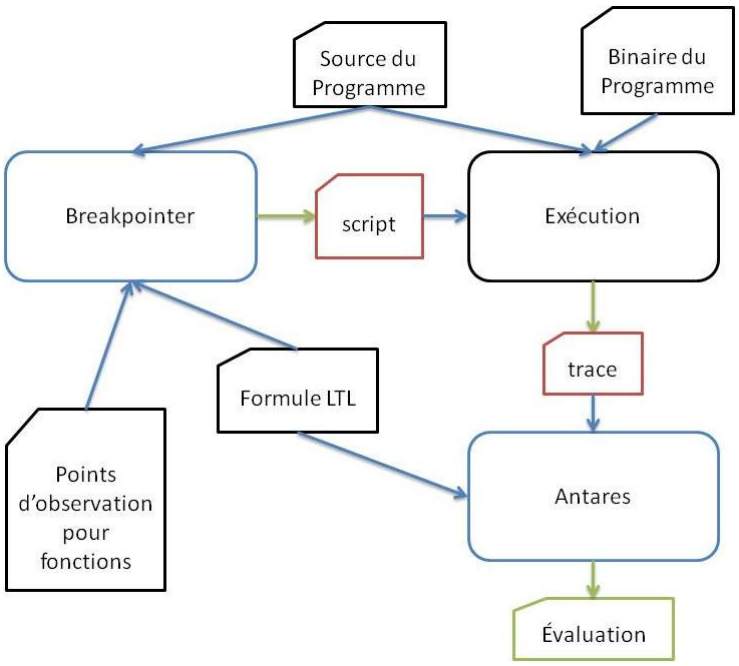

FIGURE 7.1 – Architecture globale de l'outil AnTarES

Breakpointer est un greffon de Frama-C fondé sur Occurrence et Value Analysis. À partir d'un ensemble de variables à étudier pour un programme donné, Breakpointer fournit un script de débogage pour conduire l'exécution du programme. À chaque modification de la valeur d'une des variables étudiées, un point d'observation permet de collecter cette nouvelle valeur et de l'intégrer dans une trace.

AnTarES est un outil d'analyse de trace d'exécution. Une propriété écrite en LTL est vérifiée sur une trace d'exécution donnée. AnTarES utilise un outil de transformation de formules LTL en automates de Büchi, Ltl2ba.

7.2 Génération de trace avec une analyse statique via Breakpointer

Breakpointer (5) génère un script d'exécution, après avoir analysé la formule à vérifier. Chaque occurrence de chaque variable intervenant dans la formule est recherchée dans l'ensemble du source du programme. Breakpointer fait une analyse sémantique de programme en calculant une sur-approximation des valeurs de chaque variable, ce qui permet de prendre en compte les alias de variables. Un filtre syntaxique utilisé sur le résultat de l'analyse sémantique permet de ne conserver que les points d'observation où la variable étudiée est modifiée. Les points d'observation où la variable est utilisée en lecture sont ainsi supprimés.

L'analyse effectuée par Breakpointer se fait en cinq étapes :

1. la lecture des entrées et options,
2. l'analyse syntaxique du programme étudié, pour les fonctions (5.2.3),
3. l'analyse sémantique du programme étudié, pour les variables (5.2.1),
4. le calcul des points d'observation avec le filtrage(5.2.2),
5. la génération du script.

Le schéma 7.2 résume les entrées et sorties nécessaires au fonctionnement de Breakpointer.

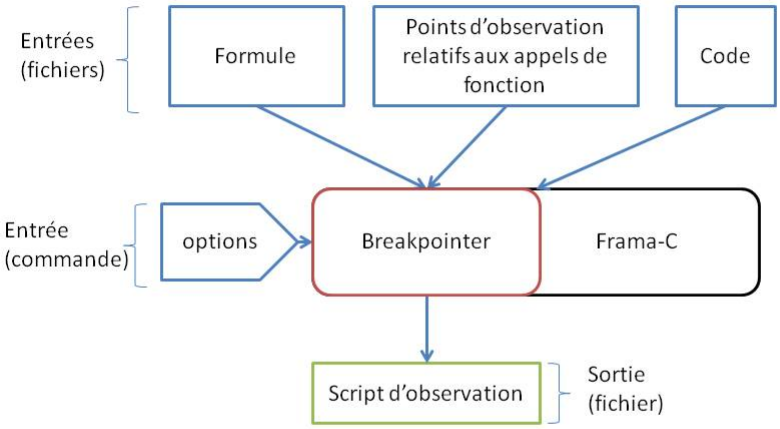

FIGURE 7.2 – Entrées et sorties de Breakpointer

7.2.1 Étape 1 : lecture des entrées et options

La première étape consiste à déterminer les variables à extraire et les options de Breakpointer. Concernant les variables, elles sont disponibles dans la formule LTL. Les variables ghost, définies pour l'appel de fonctions sont, quant à elles, définies dans un fichier à part. Les options de Breakpointer sont définies en ligne de commande, avec l'appel de l'analyseur statique.

7.2.1.1 Extraction des informations

Les variables Breakpointer analyse la propriété temporelle à vérifier pour en extraire l'ensemble des variables à étudier. L'extraction des données est basée sur la

grammaire du langage LTL étendu et défini dans le chapitre 4.

Les fonctions Concernant les appels de fonctions, un fichier spécifique contient la définition de chaque point d'observation relatif par rapport à l'appel d'une fonction donnée. Le langage de description de points d'observation relatifs à un appel de fonction est défini par la syntaxe suivante :

$Main$	$::= LFonc$	Liste des fonctions
$LFonc$	$::= Fonction$	Une fonction
	$\mid\ LFonc\ Fonction$	liste de fonction par définition récursive

Dans un fichier il peut donc y avoir une liste de définitions de points d'observation pour une ou plusieurs fonctions.

$Fonction$	$::= function\ Nom = EOL\ \{ListePoint\}$	Définition d'une fonction
$ListePoint$	$::= Point$	Un point d'observation
	$\mid\ ListePoint\ Point$	liste de point par définition récursive

Une fonction commence avec le mot-clef $function$ et le nom de la fonction concernée. EOL correspond à une fin de ligne. Pour chaque fonction, il est possible de définir une liste de points d'observation relatif.

$Point$	$::= make\_bp\ Ligne\ \#begin\ DATA\ \#end$	Définition d'un point d'observation

Un point d'observation commence par le mot-clef $make\_bp$ avec la ligne relative par rapport à l'appel de fonction. Un nombre positif place le point d'observation après l'appel de fonction tandis qu'un nombre négatif le place avant. Précédé par un $\#begin$, DATA n'est pas vérifié syntaxiquement. Il s'agit des commandes de débogage à effectuer. Elles sont écrites en gdb. Le mot-clef $\#end$ doit succéder à DATA pour signifier la fin des commandes à effectuer.

Pour simplifier l'écriture du fichier, un certain nombre de balises a été défini. Ces balises sont traitées lors de l'écriture du script de débogage dans un fichier.

Le tableau 7.1 liste ces différentes balises. Les balises de communication servent à lancer la communication avec le processus d'écriture d'informations dans la base de données. Les cinq balises suivantes font référence à la table et aux noms des différentes colonnes de la table de la base de données. Une définition du schéma de la base de données est disponible dans 5.1. Les trois dernières commandes permettent de mettre à jour le temps et l'identifiant de l'état.

TABLEAU 7.1 – Liste des balises d'aide à l'écriture de commande

Balise	Description
<opencom>	ouverture des communications avec la base de données
<closecom>	fermeture des communications avec la base de données
<tablename>	nom de la table
<idname>	nom de la colonne de l'identifiant de l'état
<tickname>	nom de la colonne temps
<tickcmd>	commande d'appel de la fonction temps de la plate-forme d'analyse dynamique
<tickupdate>	commande pour mettre à jour la valeur courante du temps
<idupdate>	commande pour incrémenter la valeur courante de l'identifiant d'un état

Exemple 45 *Cet exemple propose de définir un point d'observation lors de l'appel de la fonction foo. Le point d'observation est posé à la ligne de l'appel de la fonction foo. Les communications avec le processus d'écriture dans la base de données sont ouvertes. La variable gdb* $*sequence est mise à zéro, puis le temps est mis à jour. Le temps est calculé par la plate-forme d'analyse dynamique en fonction du nombre de cycles de calcul. La variable ghost sequence est écrite dans la base de données, avec pour valeur* $*sequence, soit* 0*. Enfin les communications avec le processus d'écriture dans la base de données sont fermées et l'identifiant de l'état est mis à jour.*

```
function foo =
{
make_bp 0 #begin <opencom>
set $sequence = 0
<tickupdate>
printf "<DYNA> INSERT INTO <tablename> VALUES (%d,%d, \"
    sequence\",\"%d\"); </DYNA>", $<idname>, $<tickname>,
    $sequence
<closecom>
<idupdate> #end
}
```

7.2.1.2 Détail des options

Il faut choisir le nom du fichier où se trouve la formule dont il faut extraire les variables à étudier. Il est possible d'effectuer une analyse sémantique de programme en changeant le point d'entrée. Par défaut, Value Analysis effectue une analyse sémantique avec la fonction *main* comme point d'entrée. Un ensemble de commandes personnalisées sont intégrables dans le script généré. La commande permettant d'extraire le temps d'exécution du programme est personnalisable. Le nom de l'exécutable ciblé est personnalisable. Enfin le nom de la base de données générée

à partir du script est paramétrable.

7.2.2 Étape 2 : Utilisateur d'un visiteur de Frama-C pour l'analyse syntaxique du programme

7.2.2.1 L'objet visiteur

Le programme est analysé via un arbre syntaxique abstrait, défini par la bibliothèque CIL [Necula 2002] (Une version OCaml de Cil est disponible dans [CIL 2002].) La bibliothèque de Frama-C dispose d'un objet appelé visiteur, qui hérite des visiteurs de CIL.

Il s'agit d'un système conçu pour parcourir et modifier efficacement l'arbre syntaxique abstrait. Chaque type d'élément présent dans l'arbre dispose d'une méthode pour détailler les actions à réaliser lorsque le type est rencontré durant le parcours.

7.2.2.2 Fonction du visiteur implémenté

Le visiteur implémenté a deux missions :

- Chaque variable rencontrée est collectée si elle fait partie des variables à étudier.
- Pour chaque appel de fonction, les positions de ces appels sont collectées si la fonction est dans la liste des fonctions à étudier.

Concernant la collecte des variables par analyse syntaxique, cette opération est nécessaire pour obtenir un ensemble d'objets variable correspondant aux noms donnés. En effet, Occurrence a besoin de ces objets pour effectuer ses calculs.

7.2.3 Étape 3 : appel de Value Analysis pour l'analyse sémantique

Avant de faire appel à Value Analysis, il est nécessaire de définir le point d'entrée des calculs. Par défaut, c'est la fonction *main* qui servira de point d'entrée. Sinon, c'est la fonction dont le nom a été passé en option lors du démarrage du greffon.

L'analyse sémantique est effectuée grâce à l'appel de $!Db.Value.compute$, une fonction de la bibliothèque des fonctions de Frama-C qui lance l'analyse statique du programme.

7.2.4 Étape 4 : filtrage syntaxique des données obtenues et génération du script

7.2.4.1 Filtrage

Lors de cette étape, on reprend l'ensemble des objets variables obtenus par l'analyse syntaxique via le visiteur et pour chaque élément de l'ensemble, on obtient l'ensemble des occurrences de cette variable, que ce soit directement ou via un pointeur.

Pour chaque occurrence, il reste à vérifier s'il s'agit d'un appel en lecture ou en écriture. Lors d'un appel en écriture d'une variable, la levée d'une exception empêche l'ajout de ce point d'observation à la liste des points à conserver.

7.2.4.2 Génération du script

La génération du script respecte le langage de script de débogage *gdb*. La génération se fait à l'aide de la fonction *fprintf* du module *Format* du langage ocaml.

Exemple 46 *En reprenant l'exemple 45, voici ce que donne la génération du script gdb.*

```
# fonction foo
b main.c:201
command
  printf "<DYNA> BEGIN TRANSACTION; </DYNA>"
set $sequence = 0
set $tick = $tick+1
printf "<DYNA> INSERT INTO automate VALUES (%d,%d, \"
    sequence\",\"%d\"); </DYNA>", $Timestamp, $tick,
    $sequence
printf "<DYNA> COMMIT TRANSACTION; </DYNA>"
set $Timestamp = $Timestamp+1
c
end
b main.c:428
command
  printf "<DYNA> BEGIN TRANSACTION; </DYNA>"
set $sequence = 0
set $tick = $tick+1
printf "<DYNA> INSERT INTO automate VALUES (%d,%d, \"
    sequence\",\"%d\"); </DYNA>", $Timestamp, $tick,
    $sequence
printf "<DYNA> COMMIT TRANSACTION; </DYNA>"
set $Timestamp = $Timestamp+1
c
end
```

Exemple 47 *En reprenant l'exemple 45, voici ce que donne la génération du script gdb pour un programme qui appelle la variable foo aux lignes 201 et 428 du fichier main.c :*

```
# fonction foo
b main.c:201
command
  printf "<DYNA> BEGIN TRANSACTION; </DYNA>"
set $sequence = 0
set $tick = $tick+1
```

```
printf "<DYNA> INSERT INTO automate VALUES (%d,%d, \"
    sequence\",\"%d\"); </DYNA>", $Timestamp, $tick,
    $sequence
printf "<DYNA> COMMIT TRANSACTION; </DYNA>"
set $Timestamp = $Timestamp+1
c
end
b main.c:428
command
 printf "<DYNA> BEGIN TRANSACTION; </DYNA>"
set $sequence = 0
set $tick = $tick+1
printf "<DYNA> INSERT INTO automate VALUES (%d,%d, \"
    sequence\",\"%d\"); </DYNA>", $Timestamp, $tick,
    $sequence
printf "<DYNA> COMMIT TRANSACTION; </DYNA>"
set $Timestamp = $Timestamp+1
c
end
```

Exemple 48 *Soit le programme suivant :*

```
int rr=1;

int opa(int r) {return r+1;}

void opb () {if(rr<4998) {rr+=2;}}

void opc () {rr=600;}

int main() {
   if (rr<5000) rr=opa(rr);
   opb();
   goto L6;
   opc();
 L6:
   return 1;


}
```

L'écoute de la variable rr génèrera les observateurs suivants :

```
b main.c:5
command
```

```
  printf "<DYNA> BEGIN TRANSACTION; </DYNA>"
  set $tick = $tick+1
  printf "<DYNA> INSERT INTO automate VALUES (%d,%d, \"rr
      \",\"%d\"); </DYNA>", $Timestamp, $tick, rr
printf "<DYNA> COMMIT TRANSACTION; </DYNA>"
set $Timestamp = $Timestamp+1
c
end
b main.c:10
command
  printf "<DYNA> BEGIN TRANSACTION; </DYNA>"
set $tick = $tick+1
printf "<DYNA> INSERT INTO automate VALUES (%d,%d, \"rr
    \",\"%d\"); </DYNA>", $Timestamp, $tick, rr
printf "<DYNA> COMMIT TRANSACTION; </DYNA>"
set $Timestamp = $Timestamp+1
c
end
```

7.3 Vérification de programmes sous AnTarES

AnTarES a été écrit en OCaml, un langage interprété, dont le système de typage fort permet d'éviter la majeure partie des bugs inhérents à la programmation d'un tel outil.

Cependant, un des inconvénients du langage OCaml est la difficulté à écrire des programmes parallèles. En effet, les tâches des programmes parallèles dans OCaml ne peuvent pas s'exécuter sur plusieurs cœurs. La raison principale est que le ramasse miette (ou garbage collector) n'est pas adapté aux programmes parallèles.

L'écriture de plusieurs programmes communiquant entre eux par des sockets pallie ce manque du langage. L'architecture du programme est donc distribuée, ce qui évitera des surcharges de mémoire vive pour de trop grandes traces. En effet, le travail pourra être réparti sur plusieurs machines différentes.

7.3.1 Architecture globale de l'outil AnTarES

L'architecture globale de l'outil de vérification AnTarES, dont le but est de vérifier une propriété temporelle sur une trace d'exécution donnée, a été réfléchie de manière à minimiser le temps nécessaire à la lecture de la trace d'exécution.

De fait, il possède une architecture distribuée en modules, dont le point fort est de pouvoir lire, simultanément, plusieurs parties différentes d'un même fichier. Chaque module de lecture lit la portion de trace qui lui incombe. Une portion de trace est un ensemble d'états successifs.

En outre, une fois que les informations ont été lues, elles restent en mémoire dans les modules de lecture jusqu'à l'arrêt de ces modules. Ainsi, en cas d'erreur

dans l'écriture d'une propriété temporelle, il est possible de relancer rapidement la vérification d'une propriété sur la trace mise en mémoire.

Le schéma 7.3 synthétise l'architecture de l'outil.

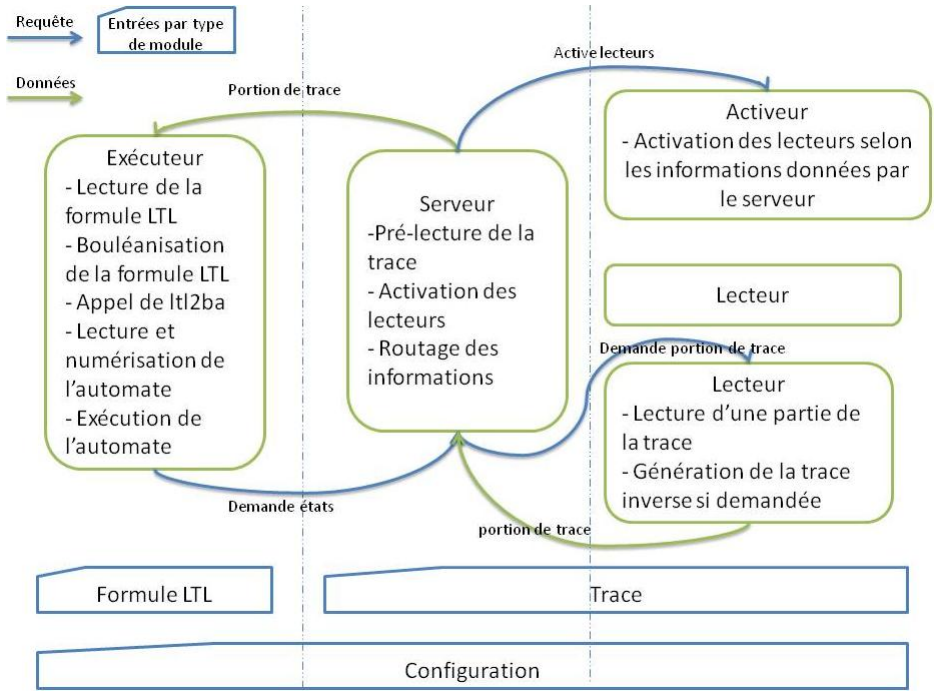

FIGURE 7.3 – Architecture globale de l'outil AnTarES

Deux programmes additionnels, de confort, ont également été implémentés. Le premier, appelé module d'activation, a été conçu pour activer automatiquement sur une machine donnée l'ensemble des modules de lecture alloués à la dite machine. Ce module doit donc être lancé en premier sur toutes les machines destinées à accueillir des modules de lecture. Le second programme, ou module extinction, provoque l'arrêt de l'ensemble de l'outil, sur toutes les machines. La libération des ports utilisés par les différents serveurs est, de plus, immédiate.

7.3.2 Le serveur de routage

7.3.2.1 Pré-lecture de la trace pour les modules de lecture

Une fois l'activateur de lecteur démarré sur chaque machine nécessaire à la lecture de la trace, le serveur de routage peut être lancé. Son premier rôle consiste à survoler la trace à lire pour en connaître le nombre d'états. Ceci fait, pour un nombre d'états $|\sigma|$, si n modules de lecture sont chargés de lire la trace, chaque module lira $|\sigma|/n$ états.

La pré-lecture d'une trace est régie par une classe adaptée au format de la trace. Dans le cas d'une trace dont le format(7.3.3.1) n'est pas une base de données, la classe fournit l'octet de début de lecture et le nombre d'états à lire. Pour une base

de données, le numéro du premier état et le nombre d'états à lire suffisent.

Une fois la pré-lecture terminée, le serveur envoie aux modules d'activation les informations de lecture. Ces derniers se chargent de démarrer correctement les différents modules de lecture. Un unique module d'activation par ordinateur est nécessaire pour qu'AnTarES fonctionne correctement.

7.3.2.2 Routage des informations

Lors de l'exécution de l'automate de Büchi, les états sont chargés par portion. Le nombre d'éléments dans une portion est définissable dans le fichier de configuration. Ce nombre d'éléments par portion influence considérablement la rapidité d'AnTarES. En effet, si un seul état transite par requête, le temps de réponse du réseau pour une requête est multiplié par le nombre d'états, ce qui ralentit considérablement la vérification. Faire transiter plusieurs états à la fois atténue cet effet.

À chaque fois qu'une nouvelle portion est nécessaire, une requête est faite au serveur de routage. Ce dernier, à partir du numéro d'état manquant pour l'exécution de l'automate de Büchi, détermine le numéro du module dans lequel se situe l'information.

Si le numéro de l'état dépasse les bornes, le serveur le signifie au module d'exécution qui lance la procédure de fin de trace. Sinon, il effectue une requête au module approprié pour collecter l'information. Une fois la réception de l'information effectuée, il la renvoie au module d'exécution de l'automate, qui peut poursuivre son travail.

7.3.3 Le module de lecture

7.3.3.1 Les formats

Le module de lecture accepte deux types de format de trace :
- une trace contenue dans une base de données sqlite3,
- une trace ASCI au format VCD (Value Change Dump).

L'organisation du module accepte la prise en charge de nouveaux formats de trace si nécessaire. Deux classes abstraites ont été écrites pour la gestion de lecture d'une classe : une pour la pré-lecture, et une pour la lecture effective. Pour étendre la portée d'AnTarES à un nouveau format, il suffit d'écrire deux nouvelles classes héritant des classes abstraites.

La trace en mémoire sous AnTarES possède un format générique qui ne dépend pas du format de la trace d'entrée. Elle contient, pour chaque état, la liste des variables modifiées uniquement, et une variable spéciale *time* qui contient le temps d'exécution de l'état donné.

La trace sqlite3 Pour lire une trace dans une base de données Sqlite3 [SQL], on dispose d'une librairie C avec l'installation de Sqlite3.

On utilisera l'interface OCaml/C pour Sqlite3 présente dans l'environnement Godi [GOD 2012], un environnement permettant de gérer efficacement un ensemble de packages pour OCaml.

La base de données Sqlite3 utilisée contient une table trace, de quatre colonnes :
- l'identifiant de l'état,
- le temps courant de l'état,
- le nom de la variable,
- la valeur de la variable, sous forme de chaîne de caractères.

La trace VCD Le format ASCII VCD a été spécifié dans le standard IEEE 1364-1995. Il est composé de deux parties, l'en-tête et le corps.

Dans l'en-tête, chaque variable de la trace est répertoriée, avec son type, et un identifiant unique lui est assigné.

Dans le corps, chaque état défini commence par l'écriture de son temps courant. Pour chaque variable modifiée, la nouvelle valeur est consignée, à l'aide de son identifiant unique. Selon le type de la variable, la définition peut être écrite avec un bit unique (événement), ou par la donnée du nombre de bits qu'occupe la variable et l'écriture ASCII (0 ou 1) de chaque bit (entier, flottant).

L'exemple 49 présente un exemple simplifié d'en-tête VCD, et l'exemple 50 deux états écrits avec le format VCD.

Exemple 49 *Entete de format VCD*

```
1  $date 2010.7.27 16:46:19 $end
2  $version TRACEGENERATION v1.0 $end
3  $timescale 1us $end
4  $scope module essai $end
5  $upscope $end
6  $scope module test1 $end
7  $var integer 32 P9 externe.compteur $end
8  $var event 1 P10 externe.compteur.revt $end
9  $var event 1 P11 externe.compteur.wevt $end
10 $upscope $end
11 $enddefinitions $end
```

L'entête contient la définition des variables de la trace (nom,type). Pour chaque variable, un identifiant lui est attribué. Cet identifiant est utilisé lors de la définition d'un état du programme.

Exemple 50 *Corps de format VCD*

```
1 #133
2 b00000000100000010000110100111110 P9
3 1P11
4 #754
5 1P10
```

Un état commence par la définition du temps courant d'exécution (entier précédé de "#"). Les modifications des variables sont ensuite définies. Une ligne correspond à la modification d'une variable (valeur, identifiant).

7.3.3.2 Lecture de la trace

Le module de lecture lit les informations de la trace, grâce aux informations données par le serveur de routage. La trace est stockée dans une table de hashage. La clef d'une donnée est le numéro de l'état. La donnée est la liste des variables modifiées et de leur nouvelle valeur.

Si l'utilisateur l'a spécifié, une fois la trace lue, l'outil effectue un traitement a postériori de la trace pour adapter la trace à une lecture inversée. Ce traitement est expliqué en 6.3.3.

Le problème de lecture inversée devient plus complexe lorsque la lecture de la trace est séparée en plusieurs modules.

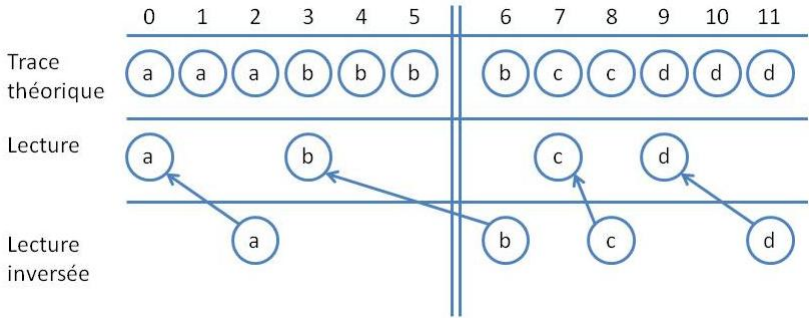

FIGURE 7.4 – Lecture inversée avec deux modules.

La figure 7.4 illustre le problème. La trace théorique sert de référence au raisonnement. La lecture s'effectue selon la trace étiquetée "Lecture". La trace est ensuite recalculée pour qu'une lecture inversée soit possible. Lors de ce calcul, quand une variable de la trace lecture change, il faut à l'état précédent récupérer la valeur de la variable pour la série d'états. Par exemple, lorsqu'à l'état 3, la variable x prend la valeur b, pour la trace inversée, il faut récupérer la valeur de x à l'état 2, a, qui se trouve à l'état 0. Or il n'est pas possible d'effectuer cette manipulation lorsque la valeur cherchée de la variable est lue par un autre module, comme c'est le cas de la valeur de x à l'état 6.

Pour pallier ce problème, il faut, pour chaque variable calculer la valeur de celle-ci au dernier état lu par chaque module. Par exemple, ici à l'état 5, la valeur de x est b. Ces données sont conservées dans une table de hashage prévue à cet effet. Si une variable, au dernier état d'un module donné, n'a pas de valeur, c'est parce que cette variable n'a pas été modifiée pendant toute la portion lue par ce module. La valeur de cette variable sera recherchée dans la table de hashage du module précédent, et ainsi de suite.

Pour conserver l'indépendance entre les différents modules, c'est le serveur central qui va ré-assembler la trace inverse. La figure 7.5 illustre le comportement du

serveur. La trace supérieure est une trace théorique de référence. La trace du milieu est celle lue par AnTarES. La trace inférieure est celle qui doit être calculée. Les tables de hashage sont modélisées par les rectangles contenant la valeur de x.

Dans le module 1, la dernière valeur de x est b. C'est ce que reçoit le serveur. Dans le module 2, x ne subit pas de modification. De ce fait, la table de hashage n'a pas d'information sur x en fin de lecture de cette portion de trace. Lors du ré-assemblage par le serveur, les tables des modules 1 et 2 fusionnent de sorte que la table 2 du serveur ait les valeurs les plus récentes de chaque variable. Dans cet exemple, la valeur la plus récente de x est b. Pour le module 3, la valeur de x est connue et vaut d. Cette table est fusionnée avec la table 2 du serveur, pour obtenir, une fois encore une table de hashage contenant pour chaque variable la valeur au dernier état.

Au final, chaque fois qu'une valeur de variable se trouve dans un module différent, le serveur utilise ces tables pour compléter la trace, avant de l'envoyer au module client.

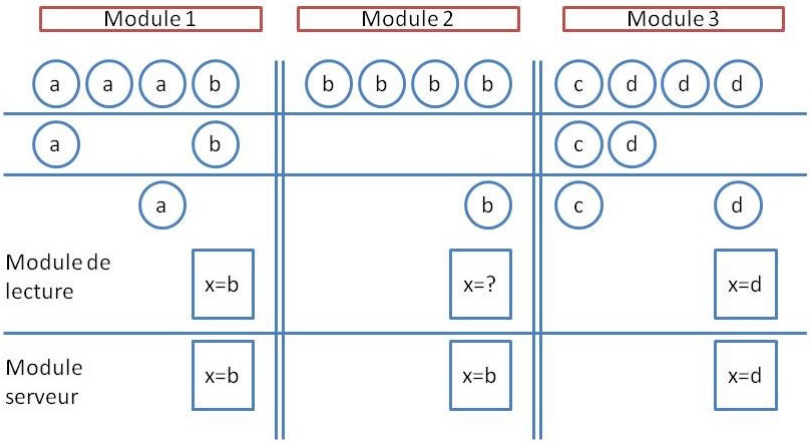

FIGURE 7.5 – Lecture inversée avec trois modules.

7.3.4 Exécution d'une trace

Le schéma 7.6 synthétise le fonctionnement du module d'exécution.

7.3.4.1 Cas normal

Lecture de la formule à vérifier La première étape consiste à lire le fichier contenant la propriété à vérifier. Par la suite, on appellera ce fichier *formule.ltl*. Le nom reste cependant entièrement personnalisable. Un parseur utilisant ocamlyacc et ocamllex a été écrit. Il prend en compte le langage défini dans la section 4.3.

Un arbre syntaxique abstrait en OCaml permet d'organiser les informations lues.

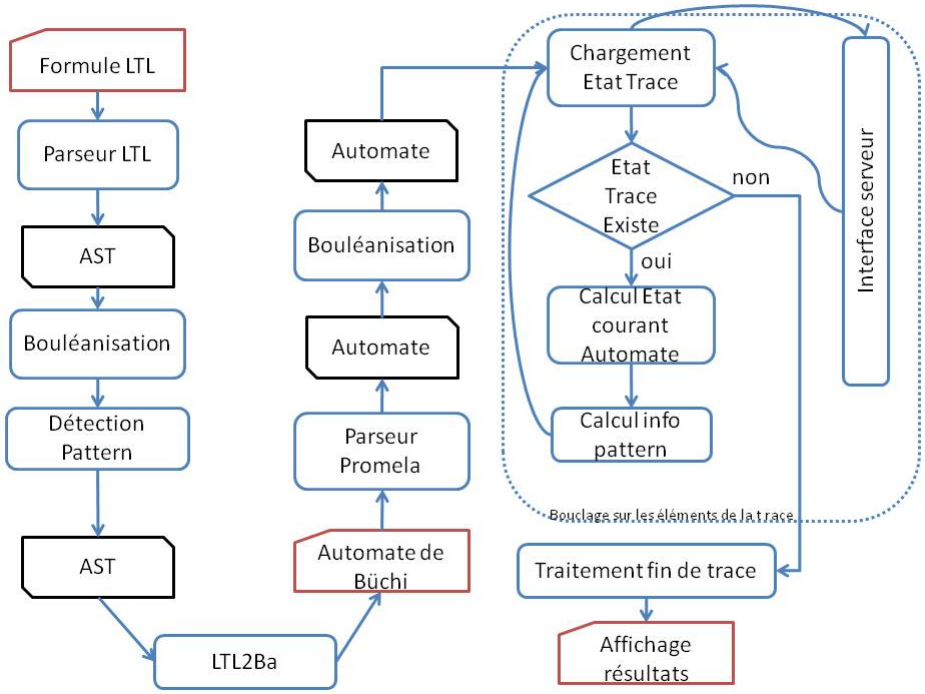

FIGURE 7.6 – Fonctionnement du module d'exécution

```
type varinst_t={var:string; mutable vale:valeur_t; mutable
    proch:varinst_t next_t; mutable modifcp:Int64.t ; mutable
     prof:int; mutable profmax:int}
```

```
type typop=
  | BOOLEEN
  | NUMINT
  | NUMFLOAT
  | NONDEF
```

Une *varinst_t* correspond à une variable avec un nom, une valeur et la *varin_inst* correspondant à la modification précédente de cette variable. Un *typop* est le type d'une opération ou d'une variable.

```
type vecteur_t=
  | VARIABLE of int
  | TEMPS of int
  | COMPTEUR of int
```

Le vecteur est une coordonnée, pour une variable donnée, pour savoir si on fait référence à la variable même, au temps de la variable ou à son numéro d'occurrence. L'entier fait référence à la n^e valeur précédente, avec n entier positif ou nul. Un entier nul fait référence à la valeur courante.

```
type nlitt=
  | FLOAT of float
  | INT of Int64.t
  | NVAR of varinst_t * ( typop ref) * vecteur_t
  | TODEF of string*string
```

Un *nlitt* modélise un littéral numérique (une variable ou constante numérique). Il peut également s'agir d'un élément à définir, mais assurément de type numérique.

```
type exp=
  | NLITT of nlitt
  | ET of exp*exp*(typop ref)
  | OU of exp*exp*(typop ref)
  | NON of exp*(typop ref)
  | XOU of exp*exp*(typop ref)
  | SHIFT_L of exp * exp*(typop ref)
  | SHIFT_R of exp*exp*(typop ref)
  | PLUS of exp list *(typop ref)
  | OPPOSE of exp *(typop ref)
  | INVERSE of exp *(typop ref)
  | Euclidiv of exp * exp *(typop ref)
  | MULTIPLIE of exp list *(typop ref)
  | NFUNC of string*exp list
```

Une *exp* est une expression numérique. Elle peut être un littéral numérique ou une opération entre une ou deux expressions numériques. Les opérateurs acceptés sont les opérateurs bit à bit et numériques classiques. Une expression numérique peut également être définie avec des parenthèses ou par une fonction.

```
type blitt=
  | BOOL of bool
  | BVAR of varinst_t*(typop ref)*vecteur_t

type boolexp=
  | SUP of exp * exp * (typop ref)
  | EGAL of exp * exp * (typop ref)
  | BLITT of blitt
```

Un *blitt* est un littéral booléen (une variable booléenne ou une constante booléenne). Une *boolexp* accepte un littéral booléen ou une opération de comparaison entre deux expressions numériques. Les opérations de comparaisons acceptées sont les égalités, les inégalités strictes et larges, et la relation différence.

```
type tempexp=
  | ALWAYS of tempexp
  | NEXT of tempexp
  | EVENTUALLY of tempexp
```

```
| UNTIL of tempexp*tempexp
| AND  of tempexp*tempexp
| NOT  of tempexp
| EXP of boolexp
| FUNCTION of string*tempexp list
| AUTOMATE of string* boolexp list
```

Une *tempexp* peut être écrite avec les opérateurs de la logique classique ($\wedge$, $\vee$, $\neg$, $\Rightarrow$) et les opérateurs de la logique temporelle linéaire ($\Box$, $\Diamond$, $\mathcal{U}$). Sont également acceptés des fonctions et des automates issus du langage d'expressions régulières pour les séquences.

Pendant la lecture des informations par le parseur, il n'est pas possible de vérifier le typage des éléments, notamment dans le cas de comparaison de deux variables comme l'exemple 51. Cette opération doit se faire a posteriori, avec les informations obtenues par la lecture de la trace.

Exemple 51 *Soit la propriété* $\Box\ ((a = b) \Rightarrow \Diamond\ (c = d))$. *Il est impossible de savoir si a et b sont des entiers ou des flottants. De même pour c et d.*

Atomisation Une fois les données dans l'arbre syntaxique abstrait, une première opération de transformation est menée : l'atomisation. Celle-ci consiste à remplacer l'ensemble des en-têtes prédicats et des fonctions prédéfinis par leur corps. En cas de définition de prédicats à partir d'autres prédicats, la transformation se poursuit jusqu'à n'obtenir que des formules avec les opérateurs de base. Il en est de même pour les fonctions. Cette étape est faite à l'aide d'un visiteur de formule LTL.

Booléanisation L'opération de booléanisation est l'étape suivante. Pour cette étape, la table de correspondance choisie est une table de hashage qui a pour clef le nom de la variable booléenne remplaçant la sous formule. Cette étape est faite à l'aide d'un visiteur de formule LTL.

Cette phase de booléanisation effectuée, la nouvelle formule est enregistrée dans le fichier $formule\_transformee.ltl$. La table de correspondance est conservée en mémoire.

Appel de Ltl2ba L'appel de Ltl2ba se fait directement par l'outil. Il est appelé par la fonction *Sys.command* de OCaml du module *Unix*. La table des correspondances est ainsi gardée en mémoire sans qu'il soit nécessaire de l'écrire.

En argument se trouve le nom du fichier transformé et l'option -F, signifiant que la formule se trouve dans ledit fichier. Il ne peut y avoir qu'une formule par fichier, sur une seule ligne.

Ltl2ba génère finalement l'automate, et l'écrit dans le fichier $formule.bua$.

Lecture de l'automate et unicité des objets variables. Un parseur permet de lire l'automate de Büchi une fois celui-ci généré. Ce parseur lit uniquement un

sous-ensemble du langage Promela. Lors de la phase de lecture de l'automate, à une variable correspond un unique objet variable. Chaque objet variable est stocké dans une table de hashage. Ce système permet de changer efficacement la valeur d'une variable donnée pendant l'exécution d'un automate.

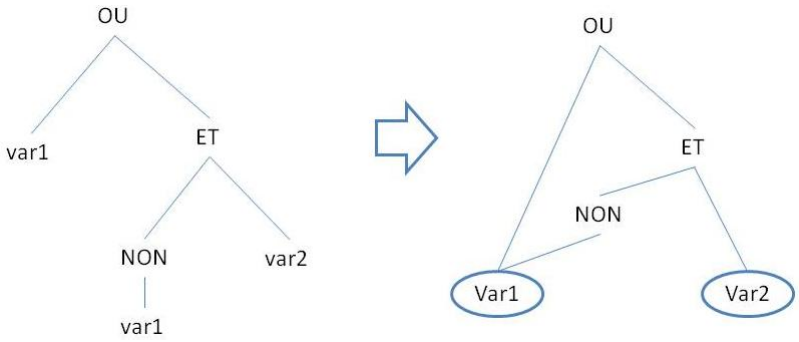

FIGURE 7.7 – Un unique objet variable par nom de variable

Numérisation. L'étape suivante est la numérisation de l'automate. Elle consiste à remplacer tout booléen appartenant à la table des correspondances par leur opération de comparaison correspondante dans toutes les transitions de l'automate. Chaque variable numérique est également modélisée par un unique objet. Cette étape est faite à l'aide d'un visiteur de formule LTL.

Inférence de type. Avant de lancer l'exécution de l'automate, une phase d'initialisation est nécessaire. En effet, lors de la lecture de l'automate, aucune vérification sur les types n'a été effectuée.

Pour ce faire, il est nécessaire d'avoir l'état initial du programme. C'est l'état zéro. Au minimum, cet état contient la valeur de chaque variable nécessaire à la vérification de la propriété. Cet état n'est pas utilisé lors de l'exécution. Le changement d'état se calcule après la mise à jour de l'état zéro vers l'état un.

Les données de cet état permettent d'inférer le type de chaque variable de la propriété et d'être en mesure d'évaluer les propriétés pour chaque transition de l'automate.

Cette inférence de type se fait sur les formules de chaque transition de l'automate, pour s'assurer de la cohérence de l'automate.

Exemple 52 *Soit la propriété* $(a > 10 \wedge b < 6,3)$ *L'inférence de type sans les données permet de dire que a est un entier et b un flottant (assimilé à un rationnel). Dans l'état zéro de la trace, on a :*

- $a = 0$
- $b = 2,3$

Il n'y a donc aucun problème détecté.

Exemple 53 *Soit la propriété* $(a < b)$ *L'inférence de type sans les données ne permet pas de déterminer le type des variables a et b. Il est juste possible de savoir que ces deux variables sont numériques. Soit les données de l'état zéro de la trace :*

- $a = 0$
- $b = 8,4$

L'inférence détecte une erreur de typage et la vérification s'arrête immédiatement.

Il n'est pas nécessaire d'effectuer l'inférence de type avant chaque évaluation, si on suppose que le typage est correctement effectué dans le programme C. Une erreur de typage dans le programme C serait détectée par le compilateur ou par Frama-C, le cas échéant.

Exécution L'exécution est basée sur l'algorithme défini dans la sous-section 6.3.1.6.

Une classe se charge de l'exécution d'un automate. Elle capte également les éventuelles exceptions caractérisant les différents problèmes pouvant survenir. Il peut s'agir d'un problème lié à la trace, à la formule, au typage des informations, à l'absence d'état atteignable ou à toute autre erreur invalidant la propriété. La fin de trace lève également une exception spécifique.

7.3.4.2 Les séquences avec des expressions régulières

Il est possible d'utiliser le langage d'expressions régulières défini en 4.3.3.

L'outil transforme une expression régulière en automate de Büchi en utilisant les algorithmes définis en 6.3.4. À partir d'une propriété de séquence écrite avec le langage d'expressions régulières, l'automate de Büchi est généré directement, sans utiliser Ltl2ba. La génération des automates est bien plus efficace avec cette méthode qu'avec une propriété de séquence écrite avec des opérateurs de la LTL. En effet, le temps de génération étant plus qu'exponentiel avec l'imbrication des opérateurs, il devient par exemple très vite impossible de générer en un temps raisonnable un automate à partir d'une propriété de séquence étendue écrite en LTL.

Exemple 54 *Soit la propriété suivante :*
$\Box\,((a \wedge cond) \Rightarrow ((a\ \mathcal{U}\ b) \wedge\Box\,(b \Rightarrow (b\ \mathcal{U}\ c)) \wedge\Box\,(c \Rightarrow (c\ \mathcal{U}\ d)) \wedge\Box\,(d \Rightarrow (d\ \mathcal{U}\ e)) \wedge$
$\Box\,(e \Rightarrow (e\ \mathcal{U}\ f)) \wedge\Box\,(f \Rightarrow (f\ \mathcal{U}\ g)) \wedge\Box\,(g \Rightarrow (g\ \mathcal{U}\ h)) \wedge\Box\,(h \Rightarrow (h\ \mathcal{U}\ i)) \wedge$
$\Box\,(i \Rightarrow (i\ \mathcal{U}\ j)) \wedge\Box\,(j \Rightarrow (j\ \mathcal{U}\ k))))$

Il s'agit d'une séquence étendue de 11 éléments. Le temps de génération de l'automate de Büchi correspondant est de plus de 1700 *secondes (pour un ordinateur avec un processeur de deux cœurs à 2,61Ghz, 16Go de RAM, sachant qu'environ 20Mo sont utilisés par Ltl2ba) et le résultat est un automate de plus de 550 états. La même propriété écrite avec le langage d'expressions régulières génère un automate de Büchi d'une douzaine d'états en moins d'une seconde.*

7.3.4.3 Détection éventuelle d'un patron de propriété

La détection des patrons de propriétés repose sur les algorithmes définis en 6.4.4.

Dans le cas de propriétés ne découlant pas d'une expression régulière, il est possible de détecter un patron de propriété, dans le but de fournir des informations additionnelles sur la trace d'exécution.

Un patron ne tient pas compte de l'ordre de lecture de la trace d'exécution. Une propriété LTL du passé ou du futur peut donc lui correspondre.

Arbre de recherche :

```
type patron=
  | TOUJOURS
  | SUIVANT
  | SERA
  | JUSQUE
  | ET
  | NON
  | TERMINAL

type 'a arbre =
  | Node of 'a * 'a arbre list * StringSet.t
  | BiNode of 'a *'a arbre list * 'a arbre list * StringSet.
    t
  | NNode of 'a * 'a arbre list array * StringSet.t
  | Feuille of StringSet.t

type patron_connu=
    patron arbre list * StringSet.t
```

Le type *patron_connu* modélise un arbre de patrons connus. Il repose sur le type générique 'a arbre. Le Node est utilisé pour les opérateurs unaires ($TOUJOURS$, $SUIVANT$, $SERA$, NON), le BiNode pour les opérateurs binaires ($JUSQUE$) et le NNode pour les opérateurs n-aires (ET). La feuille est réservée à $TERMINAL$.

Les types modélisant un patron :

```
type pattern={infos:info_t;variables:variables_t;check:
   check_t;automate:automate_t; mutable journal:journal_t}
```

Le type *pattern* contient toutes les informations nécessaires pour calculer un patron donné.

```
type info_t ={nom:string;idinfo:string;description:string;
   pattern:Ast.tempexp}

type variable_t={idvar:string; mutable valeur:Int64.t;
   inivaleur:Int64.t; resetable:bool}
```

```
type variables_t = (string,variable_t)Hashtbl.t

type check_t ={nbetat:int; nbetatfinal:int option}
```

Le type *infos_t* correspond aux informations de base du patron : nom, identifiant, description et formule. La table de hashage *variables_t* contient l'ensemble des compteurs (*variable_t*) utilisés par l'automate statistique. Le type *check_t* est utilisé pour vérifier que l'automate de Büchi généré à partir de la formule temporelle correspond à celui attendu par le patron. Des vérifications du nombre d'états de l'automate de Büchi (nombre d'états au total et nombre d'états finaux) sont faites.

```
type 'a transition_t ={origine:'a; arrivee:'a; mutable
    formule:Ast.tempexp; mutable action:action_t list}

type etat_t={id:string; init:bool; mutable transitions:
    etat_t transition_t list}

type automate_t={table:(string,etat_t)Hashtbl.t}
```

La table de hashage *automate_t* contient l'ensemble des états (*etat_t*) de l'automate statistique. Chaque état possède des transitions *transition_t*.

```
type calcul_t =
  | Elem of varconst_t
  | Additionner of calcul_t*calcul_t
  | Soustraire of calcul_t*calcul_t
  | Diviser of calcul_t*calcul_t
  | Multiplier of calcul_t*calcul_t
  | Oppose of calcul_t
  | ParCalc of calcul_t
  | Min of calcul_t*calcul_t
  | Max of calcul_t*calcul_t

type action_t =
  | Set of variable_t * calcul_t
```

Lorsque la formule contenue par une transition est évaluée à vraie, les actions *action_t* de la transition doivent être exécutées. Une *action_t* est une affectation d'une valeur à une variable. Cette valeur est déterminée par une expression arithmétique *calcul_t*. Seuls des entiers et des constantes entières *varconst_t* peuvent être manipulés.

```
type journal_t=block_t list
```

Enfin, un patron dispose d'un système de mise en forme du journal d'impression *journal_t*. Un journal est une liste de blocs *block_t* à imprimer.

```
type toprint_t=
  | Phrase of string
```

```
  | AvecNum of string* varconst_t list

type block_t =
  | Instruction of toprint_t
  | If of condition_t * block_t list * block_t list
```

Un *block_t* est soit une instruction, soit une structure conditionnelle contenant des listes de *block_t*. Selon que *condition_t* est évalué à vrai ou faux, alors la première, respectivement la seconde, liste de blocs est imprimée.

```
type compare_t=
  | Sup of calcul_t*calcul_t
  | Inf of calcul_t*calcul_t
  | SupEg of calcul_t*calcul_t
  | InfEg of calcul_t*calcul_t
  | Egal of calcul_t*calcul_t
  | Diff of calcul_t*calcul_t

type condition_t=
  | Et of condition_t*condition_t
  | Ou of condition_t*condition_t
  | Non of condition_t
  | ParCond of condition_t
  | Comp of compare_t
```

Une *condition_t* est un prédicat sur des opérations de comparaison *compare_t*. Aucune variable booléenne n'est tolérée.

Lecture d'un patron : syntaxe

$Main$	:= *blank LesInfos eols Variables eols Check eols Automate eols Journal blank EOF*	Point d'entrée

Un *blank* correspond à une liste de sauts de ligne ou au caractère vide. *eols* correspond à une liste de sauts avec un saut de ligne au minimum. $Informations$ correspond aux informations de bases du pattern. $Variables$ permet de déclarer l'ensemble des compteurs utilisés dans l'automate statistique. *Automate* est l'automate statistique. *Journal* détaille le formatage de l'affichage des compteurs. *EOF* est le caractère de fin de fichier.

$LesInfos$	:= *LesInfos eols* { *eols infos eols* }	Les informations
$infos$	:= *infos eols info*	Liste d'info séparées par saut de ligne
	| *info*	une information

$info$	$:= NOM = INFO;$	Nom du pattern
	$\mid ID = INFO;$	Id du pattern
	$\mid DESCRIPTION = STRING;$	Description du pattern
	$\mid FORMULE = formule;$	Formule du pattern

Les informations nécessaires sont le nom du pattern, son identifiant, une description du pattern et la formule LTL associée au pattern. Ces informations peuvent être données dans le désordre.

$Variables$	:= Variables $eols$ { $eols\ variables\ eols$ }	Déclaration et initialisation de variables
$variables$	$:= variables\ eols\ variable$	Liste de variables séparées par saut de ligne
	$\mid variable$	une variable
$variable$	$:= NOMVAR = VALEUR;$	une variable avec sa valeur initiale

Les compteurs sont déclarés et initialisés à cet endroit. Les compteurs sont de type entier.

$Check$	:= Check $eols$ { $eols\ checks\ eols$ }	Les éléments check
$checks$	$:= checks\ eols\ check$	Liste de checks séparés par saut de ligne
	$\mid check$	un check
$check$	$:= NBETAT = VALEUR;$	Nombre d'états dans l'automate de Büchi
	$\mid NBETATFINAUX = VALEUR;$	Nombre d'états finaux dans l'automate de Büchi

Deux types d'éléments sont vérifiés : le nombre d'états au total et le nombre d'états finaux de l'automate de Büchi attendu, correspondant au pattern défini.

$Automate$	:= Automate $eols$ { $eols\ etats\ eols$ }	l'automate statistique
$etats$	$:= etats\ eols\ etat$	Liste d'états séparés par saut de ligne
	$\mid etat$	un état

etat	:=	*ETAT eols* { *eols transis eols* }	un état

Un état de l'automate possède un identifiant *ETAT* et une liste de transitions.

transis	:=	*transis eols transi*	Liste de transitions séparées par saut de ligne
	|	*transi*	une transition
transi	:=	transition (*formule* GOTO *ETAT*) ;	une transition sans action
	|	transition (*formule* GOTO *ETAT*) *eols* { *eols actions eols* }	une transition avec action(s)

Une transition contient une formule LTL *formule* et pointe vers un état dont l'identifiant est *ETAT*. Une transition peut avoir une liste d'actions *actions* ou non.

actions	:=	*actions eols action*	Liste d'actions séparées par saut de ligne
	|	*action*	une action
action	:=	*NOMVAR* = *calcul* ;	affectation d'une valeur calculée à un compteur

Une action est une affectation d'une valeur à une variable donnée. La valeur est calculée par une expression *calcul*.

calcul	:=	*varconst*	compteur ou constante
	|	*calcul* + *calcul*	addition
	|	*calcul* - *calcul*	soustraction
	|	*calcul* / *calcul*	division
	|	*calcul* * *calcul*	multiplication
	|	- *calcul*	opposé
	|	(*calcul*)	parenthèse
	|	min (*calcul* , *calcul*)	minimum
	|	max (*calcul* , *calcul*)	maximum
varconst	:=	*NOMVAR*	variable
	|	*VALEUR*	constante

Un calcul est une opération mathématique entre des variables et constantes. Les opérateurs arithmétiques de base sont reconnus. Les calculs de minimum et maximum de valeurs sont également reconnus.

formule	::= *Booleen*	constante booléenne
	| *NOMVAR*	variable booléenne
	| (*Lt*)	parenthèses
	| $\neg$ *Lt*	opérateur non
	| *Lt* $\vee$ *Lt*	opérateur ou
	| *Lt* $\wedge$ *Lt*	opérateur et
	| *Lt* $\Rightarrow$ *Lt*	opérateur implique
	| $\circ$ *Lt*	opérateur temporel suivant
	| $\Box$ *Lt*	opérateur temporel toujours
	| $\Diamond$ *Lt*	opérateur temporel futur
	| *Lt* $\mathcal{U}$ *Lt*	opérateur temporel jusque

La formule est utilisée pour définir d'une part le pattern, dans les informations, et d'autre part dans l'automate statistique. Pour l'automate statistique, aucun opérateur temporel n'est autorisé. La vérification de cette règle se fait après la lecture du pattern.

Journal	:= Journal *eols* { *eols blocks eols* }	Mise en forme des compteurs pour affichage

Le journal décrit la mise en forme des données à afficher.

blocks	:= *blocks eols block*	Liste de blocks séparés par saut de ligne
	| *block*	un block
block	:= *inst*	une instruction d'affichage
	| If (*condition*) *eols* { *eols blocks eols* } *eols* else *eols* { *eols blocks eols* }	Structure if then else
	| If (*condition*) *eols* { *eols blocks eols* } ;	Structure if then

Un *block* est un bloc d'information à afficher. Il peut s'agir d'une instruction ou d'une structure conditionnelle contenant elle-même une ou deux listes de *block*.

inst	:= [*INSTRUCTION*] ;	une phrase sans données
	| [*INSTRUCTION*] *varconsts* ;	une phrase avec données numériques
varconsts	:= *varconsts* , *varconst*	liste de variables séparées par virgule
	| *varconst*	une variable

Une instruction est soit une phrase à afficher sans aucune donnée numérique, soit une phrase contenant des données numériques. Dans le second cas, les données numériques sont définies dans une liste. Pour faire référence à une variable numérique de cette liste dans $INSTRUCTION$, il faut utiliser une expression $\$ < x >$ où x est la position de la valeur dans la liste (le premier élément de la liste possédant l'indice 0).

condition :=	*compa*	comparaison entre deux expressions numériques
|	*condition* & *condition*	opérateur Et
|	*condition* || *condition*	opérateur Ou
|	! *condition*	opérateur Non
|	(*condition*)	Parenthèses

Une condition est une expression booléenne (et, ou, non) ou une opération de comparaison entre valeurs numériques. Les constantes et variables booléennes ne sont pas admises.

compa :=	*calcul* > *calcul*	opérateur supérieur
|	*calcul* < *calcul*	opérateur inférieur
|	*calcul* >= *calcul*	opérateur supérieur ou égal
|	*calcul* <= *calcul*	opérateur inférieur ou égal
|	*calcul* = *calcul*	opérateur égal
|	*calcul* != *calcul*	opérateur différent

Une opération de comparaison compare deux expressions numériques *calcul*.

Préparation d'un patron Lorsqu'un patron a été détecté, trois étapes sont nécessaires à sa préparation :

1. la mise en correspondance des variables du patron avec les expressions de la formule,
2. la génération de l'automate statistique en fonction de ces correspondances,
3. le remplacement des objets variable_t par l'objet variable correspondant de la table pattern.variables.

La première étape consiste à faire le lien entre une variable du patron et son expression dans la formule LTL correspondante.

Exemple 55 *Soit la formule REV* $\Box\ ((port\_utilise = 1) \Rightarrow \Diamond\ (port\_cree = 1))$. *Pour le patron* $\Box\ (P \Rightarrow \Diamond\ Q)$, *la correspondance est donc :*

- $P := port\_utilise = 1$
- $Q := port\_cree = 1$

La seconde étape consiste à remplacer dans l'automate statistique contenu dans pattern.automate, pour chaque transition, les variables des formules par l'expression correspondante.

La troisième étape est basée sur le même principe que dans 7.3.4.1. (Figure 7.7). Cette méthode permet d'évaluer efficacement les actions, ainsi que les conditions contenues dans le journal.

Calcul des informations avec le pattern Pour calculer les informations liées au pattern, la classe chargée de l'exécution de l'automate de Büchi exécute l'automate statistique simultanément. Á chaque pas d'exécution de l'automate de Büchi, un pas d'exécution de l'automate statistique est effectué.

L'automate statistique est déterministe, contrairement à l'automate de Büchi. De ce fait, une seule transition doit être activable. La conjonction des formules de toutes les transitions partant d'un même état doit donc être fausse, quelles que soient les valeurs des variables intervenant dans ces propriétés.

De plus, la disjonction des formules de toutes les transitions partant d'un même état doit être vraie, pour que l'automate soit capable de s'exécuter quels que soient les états rencontrés.

Á chaque pas de calcul de l'automate statistique, la liste des actions correspondant à la transition dont la propriété est vraie est effectuée.

7.3.4.4 Les propriétés paramétrées

Au niveau implémentation, le domaine de variation de chaque variable est calculé au moment de la lecture de la trace. Chaque module de lecture calcule le domaine de définition correspondant à sa portion de trace à lire. Le domaine complet est simplement assemblé par le serveur de routage, à partir des sous-domaines de chaque module de lecture, avant d'être transmis au module d'exécution.

L'automate de Büchi n'est calculé qu'une fois. Il est ensuite répliqué pour chaque combinaison de valeurs des variables paramétrées. Pour chaque automate répliqué, les variables paramétrées sont remplacées par leurs valeurs. Un visiteur permet de traiter cette étape.

Le nombre d'automates dépend du nombre de combinaisons. Il est important de limiter le nombre de variables paramétrées différentes dans une même formule pour que le temps de vérification reste raisonnable.

Lors de cette étape de vérification, plutôt que d'exécuter un automate, l'ensemble des automates est exécuté. Ils sont exécutés en même temps, à savoir que pour chaque état de la trace, l'ensemble des automates est mis à jour.

Conclusion du chapitre

Dans ce chapitre, les architectures des deux prototypes Breakpointer et AnTarES ont été explicités. La prochaine étape consiste à expérimenter ces prototypes sur des applications Airbus.

Chapitre 8

Expérimentations

Sommaire

Ce chapitre présente les expérimentations menées sur des applications Airbus. Deux évaluations concernent la plate-forme d'analyse dynamique elle-même. Les deux ciblent des logiciels spécifiques. Pour des raisons de confidentialité, les logiciels considérés ne seront ni nommés ni décrits. L'objectif de ce chapitre est multiple. Dans un premier temps, il permet de s'assurer sur des exemples que les propriétés à vérifier sont formalisables. L'expressivité et la facilité d'utilisation du langage sont donc testées. Concernant la trace d'exécution, la pose de points d'observation automatique doit être testée. Enfin, concernant la vérification, il est nécessaire de s'assurer des performances sur une trace de grande taille. Il faut également s'assurer de l'utilité des réponses fournies par le prototype sur les traces finies.

8.1 Logiciel 1

Cette expérimentation a été menée sur un modèle de la plate-forme d'analyse dynamique. Il s'agit d'un modèle comprenant entre autres, un calculateur, un système de controle d'interruption, une mémoire EEPROM, un émetteur et un watchdog. Ce modèle est écrit en 675 lignes de C. Le logiciel n'est pas multitâche. Il doit être arrêté volontairement sinon il tournerait indéfiniment. Ce logiciel portant sur un modèle de la plate-forme d'analyse dynamique, Breakpointer n'est donc pas utilisé.

L'objectif de l'étude consiste à vérifier le comportement du watchdog. En effet, celui-ci surveille l'intervalle de temps entre deux écritures de données dans la mémoire EEPROM. Si la différence de temps entre deux écritures dépasse un temps limite T_0 donné, alors une interruption est envoyée au contrôleur et à l'émetteur.
Pour cette section, chaque propriété à vérifier sera traitée en deux temps, avec une phase de formalisation et une phase de vérification. Une écriture sera modélisée par la variable d'état *ecrire* (1 pour écriture, 0 sinon). Une interruption envoyée au contrôleur sera modélisée par *it.controleur* (1 pour envoyé, 0 sinon). Il en est de même avec *it.emetteur* pour l'émetteur.

8.1.1 Cas du dépassement du watchdog, une première tentative

8.1.1.1 Formalisation

La première intuition consiste à mesurer la différence de temps entre deux écritures et de la comparer au seuil. Si le seuil est dépassé, l'interruption doit être envoyée.

La propriété est donc :

$$\Box\ ((ecrire?T - ecrire?1?T) > T_0 \Rightarrow \Diamond\ (it.controleur = 1 \wedge it.emetteur = 1)) \tag{8.1}$$

Par ailleurs, il s'agit d'une propriété dont le patron est connu. Des informations additionnelles sont donc disponibles.

Finalement, sous AnTarES on écrira :

$$G((ecrire?T - ecrire?1?T) > T_0 -> F(it.controleur = 1 \&\& it.emetteur = 1)) \tag{8.2}$$

8.1.1.2 Résultats

Cette propriété s'est révélée fausse. L'analyse des informations additionnelles a permis de relever des incohérences par rapport à des vérifications antérieures du watchdog. En effet, une lecture de la trace VCD avec un outil graphique a montré que le watchdog fonctionnait correctement. Or, les résultats de cet essai ont déclaré trois détections de la propriété $(ecrire?T - ecrire?1?T) > T_0 \Rightarrow \Diamond\ (it.controleur = 1)$, soit une de moins que ce qui avait été détecté par lecture graphique de la trace.

Une analyse de la trace a permis d'identifier le problème. En effet, le watchdog lance une requête d'interruption dès que le temps entre la dernière écriture et le

temps courant est plus grand que T_0, et non pas dès que le temps entre deux écritures est plus grand que T_0.

Sachant que la trace est de 1848633 états, les temps de calcul sont consignés dans le tableau 8.1

Module	Temps (s)
Lecteur (min-max)	9.21-10.09
Serveur	3.04
Client	25.85

TABLEAU 8.1 – Temps de calcul, avec 5 lecteurs

8.1.2 Cas du dépassement du watchdog, formalisation définitive

8.1.2.1 Formalisation

Après avoir pris en compte le résultat précédent, qui consiste à mesurer l'écart entre le temps courant et la dernière écriture dans la mémoire EEPROM, on obtient donc la formulation suivante :

$$\Box\ ((\$time - ecrire?T) > T_0 \Rightarrow (it.controleur = 1 \wedge it.emetteur = 1)) \qquad (8.3)$$

$\$time$ est la variable temps. Cette propriété suit également le patron $\Box\ (P \Rightarrow Q)$. Sous AnTarES, la propriété s'écrit :

$$G((\$time - ecrire?T) > T_0 -> (it.controleur = 1 \&\& it.emetteur = 1)) \qquad (8.4)$$

8.1.2.2 Résultats

La propriété s'avère vraie. Les informations additionnelles précisent en outre que la propriété $(\$time - ecrire?T) > T_0$ s'est révélée vraie 4 fois.

Les temps de calcul de l'outil AnTarES sont consignés dans le tableau 8.2, sachant que la trace possède 1848633 états.

Module	Temps (s)
Lecteur (min-max)	9.21-10.09
Serveur	3.04
Client	35.84

TABLEAU 8.2 – Temps de calcul, avec 5 lecteurs

8.1.3 Cas nominal du watchdog

8.1.3.1 Formalisation

Le cas nominal est lorsque le temps entre deux écritures n'excède pas T_0. Dans ce cas, il n'y a pas d'interruption envoyée. Pour cette propriété, il suffit d'adapter le formalisme adopté en 8.1.2

$$\Box\ ((\$time - ecrire?T) < T_0 \Rightarrow (it.controleur = 0 \wedge it.emetteur = 0)) \tag{8.5}$$

À noter, que pour une vérification plus précise, il est mieux de séparer la propriété précédente en deux :

$$\Box\ ((\$time - ecrire?T) < T_0 \Rightarrow (it.controleur = 0)) \tag{8.6}$$

$$\Box\ ((\$time - ecrire?T) < T_0 \Rightarrow (it.emetteur = 0)) \tag{8.7}$$

Ainsi, avec deux propriétés, si l'une est fausse, on peut vérifier la seconde, car il se peut qu'un bug empêche l'envoi d'une interruption à l'émetteur mais pas au calculateur par exemple.

Sous AnTarES on a donc :

$$G((\$time - ecrire?T) < T_0 -> (it.controleur = 0)) \tag{8.8}$$

$$G((\$time - ecrire?T) < T_0 -> (it.emetteur = 0)) \tag{8.9}$$

8.1.3.2 Résultats

Les deux propriétés sont vérifiées. D'après les données additionnelles, $(\$time - ecrire?T) < T_0$ est vrai 1848629 fois.

Les temps de calculs de l'outil AnTarES sont consignés dans le tableau 8.3, sachant que la trace possède 1848633 états.

Module	Temps (s)
Lecteur (min-max)	9.21-10.09
Serveur	3.04
Client (8.8)	25.84
Client (8.9)	25.77

TABLEAU 8.3 – Temps de calcul, avec 5 lecteurs

8.2 Logiciel 2

Cette expérimentation a été menée sur un modèle de la plate-forme d'analyse dynamique, similaire à celui de la section 8.1. Ce modèle comporte 750 lignes de code et tout comme le logiciel 1, n'est pas multitâche. De même, l'arrêt du logiciel

est volontaire à la fin du test. Cette expérimentation concerne le mode d'écriture dans la mémoire EPPROM. Lorsque le mode programmation est activé, c'est à dire lorsqu'une donnée doit être écrite, il y a un temps de latence entre le début et la fin de l'écriture de la donnée. Tant que l'écriture n'est pas achevée, le dernier bit du bloc à écrire est complémenté par rapport à la donnée à écrire.

Dans un premier temps, il faudra formaliser la propriété à vérifier. Dans un second temps, AnTarES permettra de vérifier cette propriété sur différentes tailles de traces, ce qui permettra d'étudier l'influence du nombre d'états dans la trace sur les temps de calcul.

8.2.1 Formalisation

Le mode d'écriture est défini par la variable *mode*. Un événement de lecture est défini par la variable *lecture.evt*. $*time* correspond au temps courant. *lecture* et *ecriture* sont respectivement les données à lire et à écrire. *lecture.addr* et *ecriture.addr* sont les adresses des données respectivement à lire et à écrire.

$$\begin{aligned} G(mode = 2 -> (lecture.evt -> \\ (((\$time - ecriture?T) < tprog \ \& \ ecriture.addr = lecture.addr) -> \\ (lecture = ((ecriture \ \& \ 0xFFFFFFFFFFFFF7F) \mid (! \ ecriture \ \& \ 0x80)) \\ )))) \end{aligned} \tag{8.10}$$

Cette propriété doit être vérifiée lorsque le mode d'écriture est le mode 2, lorsqu'un événement de lecture survient. Si l'adresse mémoire lue correspond à l'adresse mémoire où une écriture est en cours, alors la valeur lue doit vérifier la propriété suivante : ((*ecriture* & 0xFFFFFFFFFFFFF7F) | (! *ecriture* & 0x80)).

8.2.2 Résultats

La propriété est vérifiée. Bien qu'on dispose d'un patron pour cette propriété, dans ce cas, les informations additionnnelles ne permettent ici que de vérifier que $mode = 2$ se produit dans la trace.

En terme de performance, le graphique 8.1 résume les résultats obtenus pour des traces dont le fichier est compris entre 5Mo et 1.1Go. Les lecteurs et le serveur se situent tous sur une machine huit cœurs (avec 512ko de cache). Le vérifieur est sur une autre machine de même type. Le temps de calcul de vérification, Ltl2ba compris, est linéaire en fonction du nombre d'états dans la trace.

8.3 Logiciel 3

Le logiciel étudié est basé sur la norme Arinc653. La partie vérifiée du logiciel est de 550 lignes de code C (Le logiciel complet en fait 128000 lignes). La bibliothèque Arinc653 est de 2600 lignes de code C. Il s'agit en outre d'un logiciel multitâche

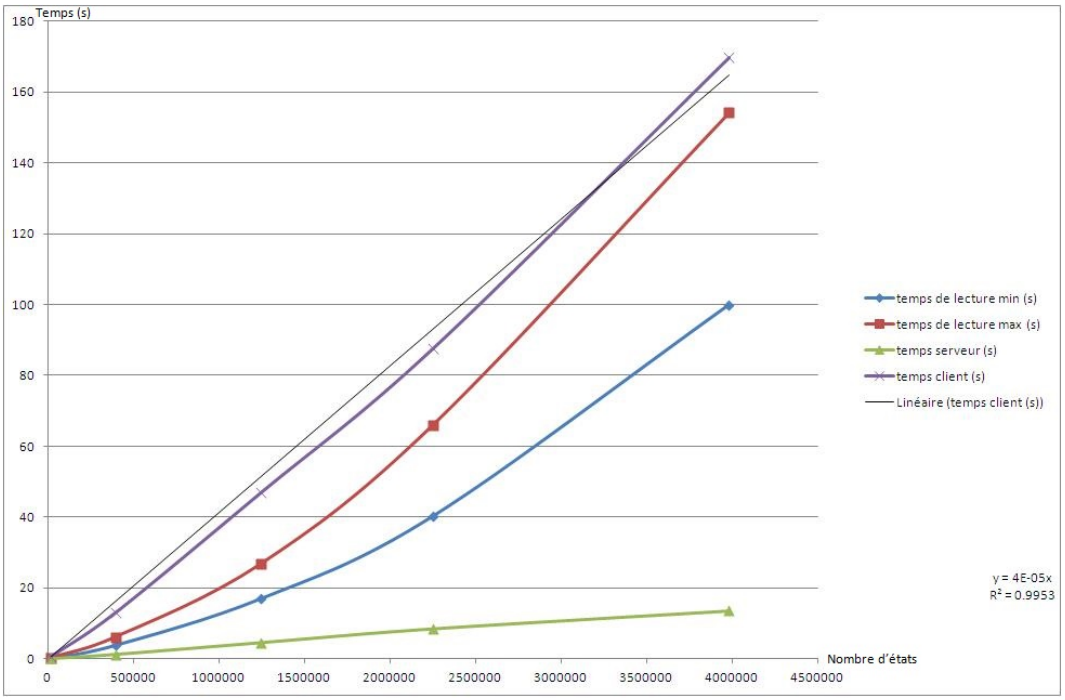

FIGURE 8.1 – Temps des différents modules, pour 10 lecteurs

qui, pour les besoins de la vérification est arrêté après un nombre de boucles fini. En temps normal, le logiciel ne s'arrête pas.

Pour son exécution, le programme utilise des ports créés dynamiquement à son initialisation. Il est donc nécessaire de s'assurer que tous les ports utilisés ont été créés au préalable.
Dans un premier temps, cette propriété sera formalisée. Puis l'utilisation de Breakpointer permettra de générer le script pour définir les points d'observation. Enfin, la propriété sera vérifiée par AnTarES.

Pour cette section, on modélisera par la variable entière $port\_utilise$ l'identifiant du port utilisé et par la variable $port\_cree$ l'identifiant du port créé.

8.3.1 Formalisation

En langage naturel, la propriété à vérifier est que tout port utilisé a été créé au préalable. Il semble que cette propriété soit plus simple à écrire en logique du passé, qu'en logique du futur.

En effet, une solution en logique du futur consisterait à dire que tout numéro de port non attribué ne peut être ensuite utilisé, soit, avec x un numéro de port quelconque :

$$\Box\ (\neg\ (port\_cree = x) \Rightarrow \Diamond\ (\neg\ port\_utilise = x)) \tag{8.11}$$

Cependant, cela signifie qu'il faudrait explorer tous les numéros de ports possibles, ce qui nécessite un temps considérable. Hors, AnTarES ne peut utiliser que des variables paramétrées dont le domaine de définition correspond au domaine de variation d'une variable présente dans la trace. Il n'est donc pas possible de vérifier cette propriété autrement qu'en écrivant la propriété temporelle pour chaque valeur possible de x.
En logique du passé, la propriété devient :

$$\boxdot ((port\_utilise = x) \Rightarrow \Diamond (port\_cree = x)) \tag{8.12}$$

Cette propriété nécessite toujours une variable paramétrée. Cependant, il s'agit d'une variable dont le domaine de définition correspond au domaine de variation de la variable modélisant l'ensemble des ports créés. Cette formalisation est donc utilisable. Elle permet en outre d'expérimenter la vérification d'une propriété par lecture inversée de la trace d'exécution.

Finalement, la propriété écrite pour AnTarES est :

$$\begin{aligned} & REVERSE\#G(\\ & port\_utilise = port\_utilise\$0- > \\ & F(port\_cree = port\_utilise\$0)) \end{aligned} \tag{8.13}$$

La prochaine étape consiste maintenant à générer les points d'observation.

8.3.2 Les points d'observation

Pour vérifier cette propriété, il est nécessaire de connaître les identifiants des ports créés. Ces derniers sont générés par la fonction $CREATE\_QUEUING\_PORT$. Cette fonction prend en paramètre le numéro du port. Cependant, il s'agit d'un pointeur. Il est donc nécessaire de récupérer la valeur de ce port après une éventuelle modification.

```
void CREATE_QUEUING_PORT  (
        QUEUING_PORT_NAME_TYPE    QUEUING_PORT_NAME,
        MESSAGE_SIZE_TYPE         MAX_MESSAGE_SIZE,
        MESSAGE_RANGE_TYPE        MAX_NB_MESSAGE,
        PORT_DIRECTION_TYPE       PORT_DIRECTION,
        QUEUING_DISCIPLINE_TYPE   QUEUING_DISCIPLINE,
        QUEUING_PORT_ID_TYPE      *QUEUING_PORT_ID,
        RETURN_CODE_TYPE          *RETURN_CODE  )  ;

void SEND_QUEUING_MESSAGE  (
        QUEUING_PORT_ID_TYPE      QUEUING_PORT_ID,
        MESSAGE_ADDR_TYPE         MESSAGE_ADDR,
        MESSAGE_SIZE_TYPE         LENGTH,
        SYSTEM_TIME_TYPE          TIME_OUT,
        RETURN_CODE_TYPE          *RETURN_CODE) ;
```

Cependant, si la valeur de cette variable est collectée à chaque modification, elle sera collectée en début de fonction, et après, au sein de la fonction, faussant la vérification de la propriété. En effet, cette variable est un pointeur et donc la valeur de cette variable n'a de sens qu'après l'appel de $CREATE\_QUEUING\_PORT$. Pour cette étude, les points d'observation seront donc posés manuellement, car il faudrait poser un point d'observation après la fonction ($CREATE\_QUEUING\_PORT$ par exemple), tout en connaissant le nom du paramètre correspondant à l'identifiant pour chaque utilisation de la fonction, ou poser un point d'observation avant chaque return de la fonction.

Pour l'utilisation d'un port, la fonction $SEND\_QUEUING\_MESSAGE$ est employée. Il suffit donc de connaître la valeur de l'identifiant de port utilisé.

La plate-forme d'analyse dynamique génère la trace d'exécution à partir du script de débogage. Pour cette étude, un programme de test d'intégration a été utilisé, afin d'avoir suffisamment de stimuli lors de l'exécution du programme.

8.3.3 Vérification de la propriété avec AnTarES

La trace générée contient 129 éléments. L'ensemble de variation de la variable des ports ($port\_utilise$), utilisés pendant l'exécution du programme, est $\{0, 1, 2, 5\}$.

Etant donné que la propriété a un pattern connu ($\Box\ (P \Rightarrow\Diamond\ Q)$), des informations additionnelles sont données dans le tableau 8.4.

Port	Nombre d'utilisations
0	1
1	107
2	2
5	20

TABLEAU 8.4 – Informations additionnelles

À noter que le port 0 n'est pas employé en pratique. Il s'agit d'une initialisation pour que l'outil AnTarES soit capable de déterminer le type des variables $port\_cree$ et $port\_utilise$.

Les temps de chaque module sont répertoriés dans le tableau 8.5

Module	Temps (s)
Lecteur	0.015
Serveur	0.024
Client	0.07

TABLEAU 8.5 – Temps de calcul

8.4 Logiciel 4

Le logiciel étudié, d'environ 20000 lignes de code C, suit une phase d'initialisation pendant laquelle chaque composant est activé. L'objectif de cette expérience consiste à vérifier que la séquence d'activation de chaque composant suit un ordre précis.
La première étape consistera à définir les points d'observation nécessaires à la vérification de la formule. Pour la seconde étape, deux méthodes seront expérimentées pour formaliser et vérifier la propriété. Pour la première méthode, des formules écrites en logique temporelle, à l'aide des opérateurs $\Box$, $\Diamond$ et $\mathcal{U}$ seront utilisés. La seconde méthode, quant à elle, utilisera le langage d'expression de séquence.

8.4.1 Les points d'observation

Pour la génération automatique des points d'observation, Breakpointer a été utilisé. En effet, chaque élément de la séquence d'activation est identifiable par une fonction. On utilise donc la capacité de Breakpointer à définir un point d'observation en fonction de la position d'appel d'une fonction définie.

Pour ces points d'observation, on définit une variable entière *sequence* qui prend une valeur spécifique à chaque activation d'une fonction d'initialisation.

Il y a 11 fonctions au total qui sont nommées $f_1, ..., f_{11}$. Pour f_i, $sequence = i$.

8.4.2 Vérification à l'aide de la logique temporelle

8.4.2.1 Formalisation

Pour rappel, pour que la correction de la vérification soit juste, il n'est pas possible d'employer l'opérateur $\circ$. Pour cet exemple, *sequence* peut prendre la même valeur pendant plusieurs états successifs. L'opérateur $\mathcal{U}$ va donc être utilisé.

Soit $i \in [\![2; 10]\!]$. La propriété se formalise de la façon suivante.

Cela se traduit donc par :

$$\Box\left((sequence = i) \Rightarrow ((sequence = i)\ \mathcal{U}\ (sequence = i + 1))\right) \tag{8.14}$$

Il faut cependant englober chaque sous-formule 8.14 pour qu'elles ne soient "activées" que lorsque $sequence = 1$ s'est produit. Dans le cas contraire, chaque suffixe de la séquence serait autorisé.

Exemple 56 *sequence=10 ; sequence=11 serait un suffixe autorisé de la séquence complète.*

$$\Box\left(sequence = 1 \Rightarrow (sequence = 1 U sequence = 2 \wedge (\wedge_{i=2}^{10}\ sous{-}formule\ 8.14(i)))\right) \tag{8.15}$$

La formalisation s'écrit donc, sous AnTarES :

$$\begin{aligned} G((sequence = 1) -> ((sequence = 1 U sequence = 2) \& \\ G(sequence = 2 -> (sequence = 2 U sequence = 3)) \& \\ G(sequence = 3 -> (sequence = 3 U sequence = 4)) \& \\ G(sequence = 4 -> (sequence = 4 U sequence = 5)) \& \\ G(sequence = 5 -> (sequence = 5 U sequence = 6)) \& \\ G(sequence = 6 -> (sequence = 6 U sequence = 7)) \& \\ G(sequence = 7 -> (sequence = 7 U sequence = 8)) \& \\ G(sequence = 8 -> (sequence = 8 U sequence = 9)) \& \\ G(sequence = 9 -> (sequence = 9 U sequence = 10)) \& \\ G(sequence = 10 -> (sequence = 10 U sequence = 11)))) \end{aligned} \tag{8.16}$$

8.4.2.2 Résultat

La propriété se révèle fausse. Après une analyse de la trace, qui ne fait que 19 éléments, on constate qu'une séquence commence à l'état 1 (sequence=1) et s'interrompt à l'état 4 (sequence=4), car à l'état 5, une nouvelle séquence redémarre à 1.

Cette erreur donne l'opportunité de tester une séquence avec une condition d'activation supplémentaire, en remplaçant la première occurrence de $sequence = 1$ par $sequence = 1 \wedge sequence?T >= 5$

Après modification de la propriété, le résultat est positif.

En terme d'efficacité, cette solution n'est pas satisfaisante. D'une part, l'automate généré dépasse les 500 états. D'autre part, le temps de calcul de Ltl2ba dépasse les 1500 secondes.

Des essais de performances ont été menés pour vérifier des propriétés de séquence sur cette trace de 19 états. Le but de cette manipulation est de voir l'influence du nombre d'éléments dans la séquence sur le nombre d'état et le temps de calcul. Les schémas 8.2 et 8.3 résument les résultats en terme de performance. Le temps nécessaire à la génération de l'automate avec Ltl2ba est plus qu'exponentiel avec le nombre d'éléments dans la séquence. En effet, le coefficient de corrélation est très éloigné de 1.

Cette explosion du nombre d'états (figure 8.3) s'explique en partie par le fait que Ltl2ba ne travaille qu'avec des variables booléennes. En effet, dans les étiquettes des transitions de l'automate, en prenant en compte des liens tels que si $(sequence = 1)$ alors $(sequence! = 2)$, il est possible de simplifier voire supprimer des transitions, parce qu'elles ont des formules insatisfiables.

Ces résultats nous ont poussé à la définition du langage d'expressions régulières pour définir des automates optimisés.

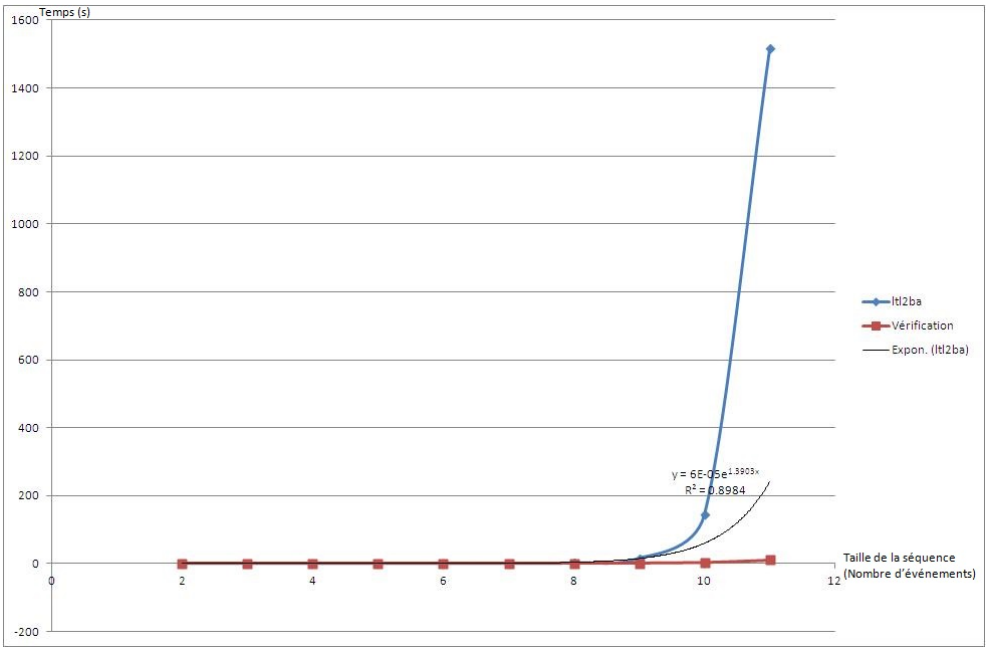

FIGURE 8.2 – Temps pris par Ltl2ba et la vérification en fonction de la taille de la séquence

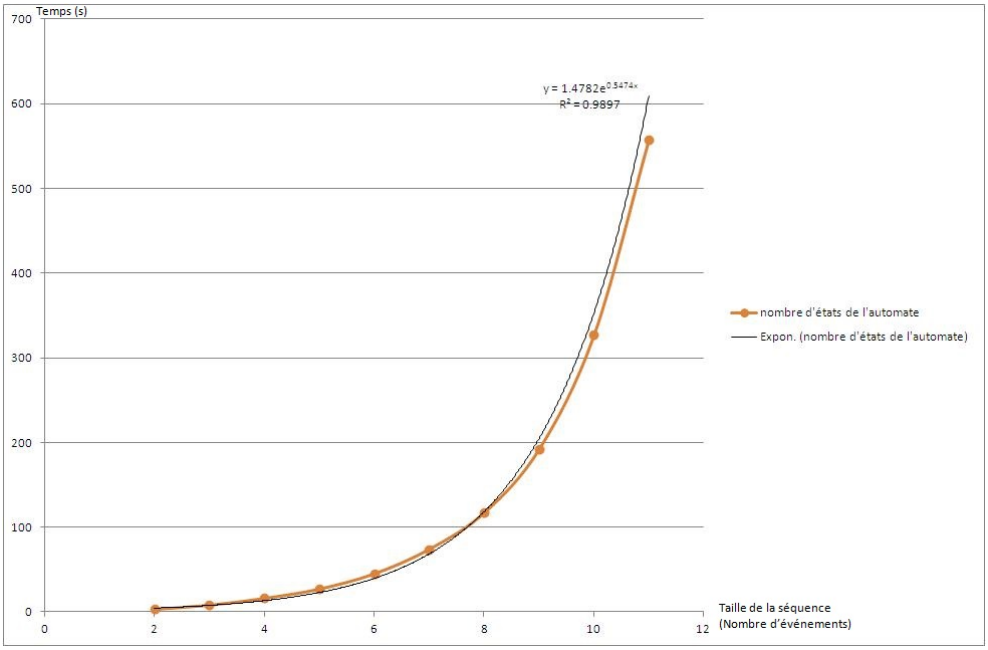

FIGURE 8.3 – Nombre d'états dans l'automate en fonction de la taille de la séquence

8.4.3 Vérification à l'aide du langage d'expressions régulières

8.4.3.1 Formalisation

Avec le langage d'expressions régulières, il suffit de déclarer que la condition d'activation est $sequence?T >= 5$, puis d'utiliser une séquence étendue (confer 4.3.3.1). La propriété est donc la suivante :

$$[sequence?T >= 5][sequence = 1; sequence = 2; sequence = 3; sequence = 4; sequence = 5; sequence = 6; sequence = 7; sequence = 8; sequence = 9; sequence = 10; sequence = 11]* \tag{8.17}$$

8.4.3.2 Résultats

La propriété est vérifiée. L'avantage de cette méthode est qu'il n'y a pas d'appel à Ltl2ba. De ce fait, le temps de calcul est considérablement réduit : seulement 0.02 secondes. Le temps de génération d'un automate est négligeable : 0.005 seconde. En outre, le nombre d'états de l'automate est très limité : seulement 12 états pour une séquence à 11 éléments.

Conclusion du chapitre

L'objectif de ce chapitre était d'expérimenter la démarche employée sur des exemples industriels concrets.

Concernant la formalisation des propriétés temporelles, le langage est suffisamment expressif. Cependant, il n'est pas toujours simple de manipuler de telles propriétés. Le logiciel 1 est un exemple, puisque la première propriété formalisée n'était pas correcte, par rapport à ce qui devait être vérifié.

La génération automatique de traces, elle, n'a pu être testée que sur les logiciels embarqués et non pas sur les modèles de la plate-forme d'analyse dynamique. En effet, cette dernière dispose de son propre système de génération de trace VCD. D'autre part, pour les cas rencontrés, ce sont principalement des propriétés sur des appels de fonctions ou sur des valeurs de paramètres passés à des fonctions qui ont été utilisés. Des efforts d'automatisation de production des scripts conduisant la génération des traces doivent être faits.

En terme de performances, les résultats sont satisfaisants. Cependant, le temps de vérification pourrait être davantage réduit, si le programme pouvait être parallélisé. En outre, la transformation des propriétés temporelles en automate de Büchi peuvent consommer beaucoup de temps, et générer des automates non optimisés. L'utilisation du langage d'expressions régulières en guise d'alternative a cependant grandement amélioré les problèmes de génération d'automate de Büchi, lorsque la propriété temporelle était une séquence. En effet, la définition d'une propriété de

séquence à l'aide d'opérateurs temporels nécessite que ceux-ci soient imbriqués. Or, le temps de transformation d'une propriété LTL en automate de Büchi est plus qu'exponentiel avec l'imbrication. L'utilisation des expressions régulières permet de générer des automates simples (en comparaison de ceux générés par Ltl2ba) en un temps très court (de l'ordre de la seconde dans les cas étudiés).

Enfin, les informations additionnelles se sont révélées importantes pour comprendre d'où provenait le problème de formalisation du logiciel 1. Il permet en outre d'avoir des informations quantitative sur les propriétés. Cela s'est révélé également utile avec le logiciel 3 pour voir quels ports étaient créés, et combien de fois ils étaient utilisés.

Quatrième partie

Conclusion

Chapitre 9

Bilan

Cette thèse CIFRE a permis d'apporter plusieurs contributions à la vérification de propriétés temporelles par analyse dynamique sur des traces d'exécution.

Un langage de formalisation des propriétés temporelles a été proposé. Ce langage repose principalement sur un fragment de la logique temporelle linéaire, qui suffit pour exprimer les propriétés à vérifier. Pour traiter le cas des propriétés de séquence, un langage d'expressions régulières a été proposé car il facilite l'expression de ces propriétés.

Il est également possible d'utiliser des propriétés paramétrées pour compacter l'écriture de propriétés temporelles qui ne diffèrent que par une valeur d'une variable du programme. D'une part, cela minimise le nombre de propriétés à écrire, et d'autre part, cela évite l'oubli d'une propriété.

Des variables spécifiques ont aussi été définies, notamment pour obtenir la n^e valeur précédente d'une variable, ou la n^e date à laquelle une variable a été modifiée, ou enfin le compteur des modifications d'une variable du programme.

La génération de la trace d'exécution du programme, quant à elle, s'effectue par l'intermédiaire d'un script généré par analyse statique. La trace obtenue est ainsi minimisée, puisque seuls les points de programme, où une variable de la propriété à vérifier est modifiée, sont collectés. La correction de la vérification est démontrée, à condition de ne pas utiliser l'opérateur temporel "suivant" ou des séquences fortes.

Pour la vérification d'une propriété temporelle sur une trace d'exécution, quatre contributions sont à noter.

La génération directe de l'automate de Büchi correspondant à une expression régulière augmente sensiblement la vitesse de vérification d'une propriété de séquence. En effet, d'une part la génération d'un automate de Büchi pour une propriété de séquence par expression régulière est bien plus rapide, mais en outre, elle fournit un automate optimisé en nombre d'états et de transitions, ce qui accélère la vitesse d'exécution de cet automate sur une trace d'exécution.

Il est possible de vérifier des propriétés temporelles du passé grâce à la lecture inverse de la trace. Plutôt que de générer un automate de Büchi à partir d'une propriété orientée vers le passé, il suffit d'écrire la propriété temporelle en remplaçant les opérateurs du passé par leur équivalent futur, de générer l'automate de Büchi correspondant et de l'exécuter sur la trace d'exécution du programme lue à l'envers.

Les propriétés paramétrées reposent sur des variables du programme dont le domaine de variation est calculé sur toute la trace. Plutôt que d'écrire une propriété par valeur possible de cette variable, et de dérouler toute l'approche proposée, l'automate de Büchi est répliqué par autant d'automates qu'il y a de valeurs dans le produit des domaines de variations des variables paramétriques. Les différents automates de Büchi sont alors exécutés simultanément.

Pour pallier le problème des traces finies, une approche pragmatique a été proposée. Celle-ci consiste à donner des informations statistiques sur la trace d'exécution pour aider l'utilisateur à conclure sur le résultat de vérification fourni par l'outil. Ces informations statistiques dépendent de la propriété (patron de propriété). En cas de propriété violée, le préfixe de la trace, où la propriété est vraie, est retourné.

Du point de vue industriel, deux prototypes ont été implémentés. Breakpointer est un greffon de Frama-C dont le but est de générer un script d'exécution contenant les points d'observation utiles à la vérification d'une propriété donnée. AnTarES implémente l'approche de vérification proposée avec une architecture distribuée, pour améliorer l'efficacité de la vérification d'une propriété temporelle.

Des expérimentations ont également été menées sur différents logiciels d'Airbus et ont donné des résultats satisfaisants.

Chapitre 10

Perspectives

Le langage de spécification tel qu'il est défini actuellement, est suffisant pour exprimer les propriétés temporelles extraites du contexte industriel de cette thèse. Cependant, ce langage pourrait être amélioré. D'une part des expérimentations avec des utilisateurs industriels sont nécessaires pour apporter des pistes de simplification de la syntaxe. D'autre part, à plus long terme, une modification du langage dans le but de l'utiliser avec d'autres techniques de vérification pourrait être envisageable, comme le propose [Delahaye 2013]. Nous pourrions disposer alors d'un langage exprimant des objectifs de vérification, où pour chaque objectif, il serait par exemple possible de recourir au test, à l'analyse statique, à l'analyse dynamique, voire même à une combinaison de ces analyses. Un tel langage pourrait en outre simplifier les procédures de certification, puisque toutes les propriétés à vérifier sur une fonctionnalité d'un programme pourraient être rassemblées, apportant une vue d'ensemble sur les moyens mis en œuvre pour s'assurer du bon fonctionnement de cette fonctionnalité.

La génération de la trace repose sur Breakpointer. Des améliorations sont probablement encore possibles du point de vue de l'optimisation. Par exemple, il serait intéressant d'utiliser des points d'observation conditionnels pour ne récupérer la valeur d'une variable dans un tableau que lorsque cette variable est modifiée. En outre, il serait intéressant d'utiliser la version multitâche de Value Analysis pour prendre en compte les alias croisés notamment.

En ce qui concerne la partie vérification, un problème qui pourrait se révéler important sur le temps de vérification d'une propriété est la fonction de booléanisation. En effet, les automates de Büchi générés après cette booléanisation ne sont pas optimisés, car lors de la booléanisation d'une formule, des liens implicites entre opérations de comparaison sont perdus. Rétablis à la numérisation dans les transitions de l'automate, il arrive que ces transitions aient des formules insatisfiables. Supprimer de telles transitions augmenterait la rapidité de la vérification, car en plus de ne pas évaluer ces transitions, si un état de l'automate ne dispose plus d'aucune transition conduisant à lui, alors il peut également être éliminé [1]. Pour optimiser les automates, deux pistes sont envisageables :

- utiliser un solveur SMT,
- développer un autre outil que Ltl2ba prenant en compte nativement les propriétés avec des variables numériques. Cette seconde approche pourrait également permettre de mixer les propriétés LTL avec les expressions régulières.

1. sauf s'il s'agit de l'état initial.

Une autre piste pour améliorer la vitesse de vérification d'une propriété serait de trouver une solution pour paralléliser l'algorithme de vérification, même s'il s'avère nécessaire, pour cela de pré-processer la trace à vérifier.

Pour une industrialisation du prototype AnTarES, il peut être envisagé de faire des investigations approfondies sur les bibliothèques OCaml existantes. Notamment, une étude de la dernière version du langage OCaml pourrait apporter de nouvelles optimisations. Une autre piste consiste à réécrire AnTarES dans un langage différent tel que le C++, qui est plus rapide d'exécution. Cependant, un avantage d'OCaml est qu'il s'agit d'un langage fortement typé. Cela limite ainsi fortement l'apparition de bug. L'écriture d'une interface graphique avec un atelier d'écriture de propriétés temporelles et affichage des informations additionnelles pourrait également être très utile. En effet, l'atelier d'écriture pourrait ainsi regrouper les patrons de vérification fréquemment utilisés, aider le spécifieur dans sa tâche notamment à l'aide de coloration syntaxique par exemple. L'affichage des résultats de vérification peuvent également aider à voir efficacement quels sont les problèmes liés à la vérification d'une propriété plus efficacement que la lecture d'un journal contenant tous les résultats.

Bibliographie

[Ait Ameur 2010] Yamine Ait Ameur, Frédéric Boniol et Virginie Wiels. *Toward a wider use of formal methods for aerospace systems design and verification.* International Journal on Software Tools for Technology Transfer, vol. 12, no. 1, pages 1–7, 2010. (Cité en page 6.)

[Barringer 2003] Howard Barringer, Allen Goldberg, Klaus Havelund et Koushik Sen. *EAGLE does Space Efficient LTL Monitoring.* Rapport technique, Nasa, 2003. (Cité en pages 12, 16, 17 et 18.)

[Barringer 2004] H. Barringer, A. Goldberg, K. Havelund et K. Sen. *Program monitoring with LTL in EAGLE.* In Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, page 264, april 2004. (Cité en page 12.)

[Barringer 2010] Howard Barringer, Alex Groce, Klaus Havelund et Margaret Smith. *Formal analysis of log files.* Journal of aerospace computing, information, and communication, vol. 7, no. 11, pages 365–390, 2010. (Cité en page 16.)

[Baudin 2002] P. Baudin, A. Pacalet, J. Raguideau, D. Schoen et N. Williams. *Caveat : a tool for software validation.* In Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on, pages 537–, 2002. (Cité en page 8.)

[Baudin 2008] Patrick Baudin, Pascal Cuoq, Jean C. Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy et Virgile Prevosto. *ACSL : ANSI C Specification Language (version 1.6)*, 2008. (Cité en pages 12 et 53.)

[Bauer 2006a] Andreas Bauer, Martin Leucker et Christian Schallhart. *Monitoring of Real-Time Properties.* In S. Arun-Kumar et Naveen Garg, editeurs, FSTTCS 2006 : Foundations of Software Technology and Theoretical Computer Science, volume 4337 of *Lecture Notes in Computer Science*, pages 260–272. Springer Berlin Heidelberg, 2006. (Cité en page 18.)

[Bauer 2006b] Andreas Bauer, Martin Leucker et Jonathan Streit. *SALT—Structured Assertion Language for Temporal Logic.* In Zhiming Liu et Jifeng He, editeurs, Formal Methods and Software Engineering, volume 4260 of *Lecture Notes in Computer Science*, pages 757–775. Springer Berlin Heidelberg, 2006. (Cité en page 13.)

[Bauer 2010] Andreas Bauer, Martin Leucker et Christian Schallhart. *Comparing LTL Semantics for Runtime Verification.* J. Log. and Comput., vol. 20, no. 3, pages 651–674, Juin 2010. (Cité en page 18.)

[Beer 2001] Ilan Beer, Shoham Ben-David, Cindy Eisner, Dana Fisman, Anna Gringauze et Yoav Rodeh. *The Temporal Logic Sugar.* In Gérard Berry, Hubert Comon et Alain Finkel, editeurs, Computer Aided Verification, volume 2102

of *Lecture Notes in Computer Science*, pages 363–367. Springer Berlin Heidelberg, 2001. (Cité en page 13.)

[Blanc 2006] Benjamin Blanc, Guy Durrieu, Abdesselam Lakehal, Odile Laurent, Bruno Marre, Ioannis Parissis, Christel Seguin et Virginie Wiels. *Automated functional test case generation from data flow specifications using structural coverage criteria.* In 3rd European Congress on Embedded Real Time Software (ERTS2006), Toulouse, France, volume 1. Citeseer, 2006. (Cité en page 14.)

[Blanchet 2003] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux et Xavier Rival. *A Static Analyzer for Large Safety-Critical Software.* In In PLDI 2003 -ACM SIGPLAN SIGSOFT Conference on Programming Language Design and Implementation, 2003 Federated Computing Research Conference, pages 196–207, San Diego, California, USA, June 7-14 2003. http://www.astree.ens.fr/. (Cité en page 8.)

[CIL 2002] *CIL : C Intermediate Language*, 2002. (Cité en page 116.)

[Cimatti 2000] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia et Marco Roveri. *NUSMV : a new symbolic model checker.* International Journal on Software Tools for Technology Transfer, vol. 2, pages 410–425, 2000. (Cité en page 70.)

[Corbett 1998] James C. Corbett. *Constructing Compact Models of Concurrent Java Programs.* In In Proceedings of the ACM Sigsoft Symposium on Software Testing and Analysis, pages 1–10. ACM Press, 1998. (Cité en page 9.)

[Cousot 1977] Patrick Cousot et Radhia Cousot. *Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints.* In Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, 1977. (Cité en page 8.)

[Cuoq 2012] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles et Boris Yakobowski. *Frama-C : a software analysis perspective.* In Proceedings of the 10th international conference on Software Engineering and Formal Methods, SEFM'12, pages 233–247, Berlin, Heidelberg, 2012. Springer-Verlag. www.frama-c.com. (Cité en pages 8, 12 et 53.)

[d'Amorim 2005a] Marcelo d'Amorim et Klaus Havelund. *Event-based runtime verification of java programs.* SIGSOFT Softw. Eng. Notes, vol. 30, no. 4, pages 1–7, Mai 2005. (Cité en page 16.)

[d'Amorim 2005b] Marcelo d'Amorim et Grigore Rosu. *Efficient Monitoring of omega-Languages.* In CAV'05, pages 364–378, 2005. (Cité en page 12.)

[Delahaye 2013] Mickaël Delahaye, Nikolai Kosmatov et Julien Signoles. *Common specification language for static and dynamic analysis of C programs.* In Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, pages 1230–1235, New York, NY, USA, 2013. ACM. (Cité en page 157.)

[Dhaussy 2009] Philippe Dhaussy, Pierre-Yves Pillain, Stephen Creff, Amine Raji, Yves Le Traon et Benoit Baudry. *Evaluating context descriptions and property definition patterns for software formal validation.* In Model driven engineering languages and systems, pages 438–452. Springer, 2009. (Cité en page 13.)

[Dijkstra 1976] E.W. Dijkstra. A discipline of programming ; automatic computation. Prentice Hall Int., 1976. (Cité en page 8.)

[Drusinsky 2000] Doron Drusinsky. *The Temporal Rover and the ATG Rover.* In Klaus Havelund, John Penix et Willem Visser, editeurs, SPIN Model Checking and Software Verification, volume 1885 of *Lecture Notes in Computer Science*, pages 323–330. Springer Berlin Heidelberg, 2000. (Cité en page 15.)

[Dwyer 1998] Matthew B. Dwyer, George S. Avrunin et James C. Corbett. *Property specification patterns for finite-state verification.* In Proceedings of the second workshop on Formal methods in software practice, FMSP '98, pages 7–15, New York, NY, USA, 1998. ACM. (Cité en pages 28, 29 et 30.)

[Eisner 2003] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac et David Campenhout. *Reasoning with Temporal Logic on Truncated Paths.* In Jr. Hunt WarrenA. et Fabio Somenzi, editeurs, Computer Aided Verification, volume 2725 of *Lecture Notes in Computer Science*, pages 27–39. Springer Berlin Heidelberg, 2003. (Cité en page 18.)

[Ferlin 2012a] A. Ferlin et V. Wiels. *Combination of Static and Dynamic Analyses for the Certification of Avionics Software.* In Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium on, pages 331–336, 2012. (Cité en page 23.)

[Ferlin 2012b] Antoine Ferlin et Virginie Wiels. *Calcul de points d'observation pour l'analyse dynamique de programmes.* In AFADL, 2012. (Cité en page 23.)

[Foster] Harry Foster, Erich Marschner et Yaron Wolfsthal. *IEEE 1850 PSL : The Next Generation.* (Cité en pages 13 et 43.)

[Gastin 2001] Paul Gastin et Denis Oddoux. *Fast LTL to Büchi Automata Translation.* In Gérard Berry, Hubert Comon et Alain Finkel, editeurs, Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01), volume 2102 of *Lecture Notes in Computer Science*, pages 53–65, Paris, France, Juillet 2001. Springer. (Cité en page 72.)

[Gerth 1997] Rob Gerth. *Concise Promela Reference*, 1997. (Cité en page 72.)

[Giannakopoulou 2001] D. Giannakopoulou et K. Havelund. *Automata-based verification of temporal properties on running programs.* In Automated Software Engineering, 2001. (ASE 2001). Proceedings. 16th Annual International Conference on, pages 412 – 416, nov. 2001. (Cité en page 17.)

[GOD 2012] *GODI - OCaml for Everybody*, 2012. (Cité en page 122.)

[Havelund 2001a] K. Havelund et G. Rosu. *Monitoring programs using rewriting.* In Automated Software Engineering, 2001. (ASE 2001). Proceedings. 16th

Annual International Conference on, pages 135 – 143, nov. 2001. (Cité en page 14.)

[Havelund 2001b] Klaus Havelund et Grigore Roşu. *Monitoring Java Programs with Java PathExplorer*. Electronic Notes in Theoretical Computer Science, vol. 55, no. 2, pages 200 – 217, 2001. RV'2001, Runtime Verification (in connection with CAV '01). (Cité en page 18.)

[Havelund 2002a] Klaus Havelund et Grigore Rosu. *Synthesizing Monitors for Safety Properties*. In In Tools and Algorithms for Construction and Analysis of Systems (TACAS'02, pages 342–356. Springer, 2002. (Cité en page 17.)

[Havelund 2002b] Klaus Havelund et Kestrel Technology. *A Rewriting-based Approach to Trace Analysis*. Automated Software Engineering, vol. 12, page 2005, 2002. (Cité en page 14.)

[Heckmann 2005] Reinhold Heckmann et Christian Ferdinand. *Verifying Safety-Critical Timing and Memory-Usage Properties of Embedded Software by Abstract Interpretation*. In Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, 2005. http://www.absint.com/. (Cité en page 8.)

[Henzinger 2003] ThomasA. Henzinger, Ranjit Jhala, Rupak Majumdar et Grégoire Sutre. *Software Verification with BLAST*. In Thomas Ball et SriramK. Rajamani, editeurs, Model Checking Software, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. Springer Berlin Heidelberg, 2003. (Cité en page 9.)

[Hoare 1969] C. A. R. Hoare. *An axiomatic basis for computer programming*. Commun. ACM, vol. 12, no. 10, pages 576–580, Octobre 1969. (Cité en page 53.)

[Julliand 2009] Jacques Julliand, Pierre-Alain Masson et Emilie Oudot. *Partitioned PLTL Model-Checking for Refined Transition Systems*. Information and Computation, vol. 207, no. 6, pages 681–698, Juin 2009. (Cité en page 8.)

[Kleene 1956] S C Kleene. *Representation of events in nerve nets and finite automata*. In In Automata Studies. Princeton University Press : Princeton, 1956. (Cité en page 68.)

[Manna 1995] Zohar Manna et Amir Pnueli. Temporal verification of reactive systems : safety. Springer-Verlag New York, Inc., New York, NY, USA, 1995. (Cité en pages 17 et 18.)

[Markey 2003] N. Markey. Logiques temporelles pour la vérification : expressivité, complexité, algorithmes. 2003. (Cité en page 64.)

[Meredith 2012] PatrickO'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen et Grigore Roşu. *An overview of the MOP runtime verification framework*. International Journal on Software Tools for Technology Transfer, vol. 14, pages 249–289, 2012. (Cité en page 15.)

[Miller 2010] Steven P. Miller, Michael W. Whalen et Darren D. Cofer. *Software model checking takes off*. Commun. ACM, vol. 53, no. 2, pages 58 – 64, feb 2010. (Cité en page 8.)

[Necula 2002] GeorgeC. Necula, Scott McPeak, ShreeP. Rahul et Westley Weimer. *CIL : Intermediate Language and Tools for Analysis and Transformation of C Programs.* In R.Nigel Horspool, editeur, Compiler Construction, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228. Springer Berlin Heidelberg, 2002. http ://www.cs.berkeley.edu/ necula/cil/. (Cité en pages 53 et 116.)

[Oddoux 2003] Denis Oddoux. Utilisation des automates alternants pour un model-checking efficace des logiques temporelles linéaires. 2003. (Cité en page 72.)

[PSL 2004] *The Accelera PSL Language Reference Manual.* Rapport technique, 2004. (Cité en page 13.)

[Randimbivololona 1999] Famantanantsoa Randimbivololona, Jean Souyris, Patrick Baudin, Anne Pacalet, Jacques Raguideau et Dominique Schoen. *Applying formal proof techniques to avionics software : a pragmatic approach.* In Jeannette Wing, Jim Woodcock et Jim Davies, editeurs, FM'99 — Formal Methods, volume 1709 of *Lecture Notes in Computer Science*, pages 719–719. Springer Berlin / Heidelberg, 1999. 10.1007/3-540-48118-4_45. (Cité en page 8.)

[RTCA 2011] Special C. RTCA. *DO-178C, Software Considerations in Airborne Systems and Equipment Certification*, 2011. (Cité en page 5.)

[Sakarovitch 2003] J. Sakarovitch. Éléments de théorie des automates. Les Classiques de l'informatique. Vuibert, 2003. (Cité en pages 69 et 80.)

[Schnoebelen 1999] P. Schnoebelen. Vérification de logiciels : Techniques et outils du model-checking. Vuibert informatique. Vuibert, 1999. (Cité en pages 8 et 69.)

[Souyris 2005] Jean Souyris, Erwan Le Pavec, Guillaume Himbert, Victor Jégu, Guillaume Borios et Reinhold Heckmann. *Computing the worst case execution time of an anvionics program by abstract interpretation.* In 5th Intl Worshp on Worst-Case Execution Time (WCET) Analysis, pages 21–24, 2005. (Cité en page 8.)

[Souyris 2007] Jean Souyris et David Delmas. *Experimental Assessment of Astrée on Safety-Critical Avionics Software.* In Francesca Saglietti et Norbert Oster, editeurs, Computer Safety, Reliability, and Security, volume 4680 of *Lecture Notes in Computer Science*, pages 479–490. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-75101-4_45. (Cité en page 8.)

[Souyris 2009] Jean Souyris, Virginie Wiels, David Delmas et Hervé Delseny. *Formal Verification of Avionics Software Products.* In Formal Methods, Lecture Notes in Computer Science, volume 5850, page 532, 2009. (Cité en page 8.)

[SQL] *sqlite3.* (Cité en page 121.)

[Stolz 2006] Volker Stolz et Eric Bodden. *Temporal Assertions using AspectJ.* Electronic Notes in Theoretical Computer Science, vol. 144, no. 4, pages 109 – 124, 2006. Proceedings of the Fifth Workshop on Runtime Verification (RV 2005). (Cité en pages 15 et 18.)

[Stous 2013] Nicolas Stous et Virgile Prevosto. *Aoraï Plugin Tutorial*. Rapport technique, Frama-C, 2013. (Cité en page 57.)

[Vorbehalten 2005] Alle Rechte Vorbehalten, Technischen Universität München, Oliver Arafat, Oliver Arafat, Andreas Bauer, Andreas Bauer, Martin Leucker, Martin Leucker et Christian Schallhart. *Runtime Verification Revisited*. Rapport technique, 2005. (Cité en page 18.)

Vérification de propriétés temporelles sur des logiciels avioniques par analyse dynamique formelle

Résumé : La vérification de logiciels est une activité dont l'importance est cruciale pour les logiciels embarqués critiques. Les différentes approches envisageables peuvent être classées en quatre catégories : les méthodes d'analyse statique non formelles, les méthodes d'analyse statique formelles, les méthodes d'analyse dynamique non formelles et les méthodes d'analyse dynamique formelles. L'objectif de cette thèse est de vérifier des propriétés temporelles dans un cadre industriel, par analyse dynamique formelle.

La contribution comporte trois parties. Un langage adapté à l'expression des propriétés à vérifier, tirées du contexte industriel d'Airbus, a été défini. Il repose notamment sur la logique temporelle linéaire mais également sur un langage d'expressions régulières.

La vérification d'une propriété temporelle s'effectue sur une trace d'exécution d'un logiciel, générée à partir d'un cas de test pré-existant. L'analyse statique est utilisée pour générer la trace en fonction des informations nécessaires à la vérification de la propriété temporelle formalisée.

Cette approche de vérification propose une solution pragmatique au problème posé par le caractère fini des traces considérées. Des adaptations et des optimisations ont également été mises en œuvre pour améliorer l'efficacité de l'approche et faciliter son utilisation dans un contexte industriel. Deux prototypes ont été implémentés, des expérimentations ont été menées sur différents logiciels d'Airbus.

Mots clés : Logique Temporelle Linéaire, Analyse Dynamique, Analyse Statique, Logiciels critiques

Verification of temporal properties on avionics software using formal dynamic analysis

Abstract : Software Verification is decisive for embedded software. The different verification approaches can be classified in four categories : non formal static analysis, formal static analysis, non formal dynamic analysis and formal dynamic analysis. The main goal of this thesis is to verify temporal properties on real industrial applications, with the help of formal dynamic analysis.

There are three parts for this contribution. A language, which is well adapted to the properties we want to verify in the Airbus context was defined. This language is grounded on linear temporal logic and also on a regular expression language.

Verification of a temporal property is done on an execution trace, generated from an existing test case. Generation also depends on required information to verify the formalized property. Static analysis is used to generate the trace depending on the formalized property.

The thesis also proposes a pragmatic solution to the end of trace problem. In addition, specific adaptations and optimisations were defined to improve efficiency and user-friendliness and thus allow an industrial use of this approach. Two applications were implemented. Some experiments were led on different Airbus software.

Keywords : Linear Temporal Logic, Dynamic analysis, Static analysis, critical software