Mini Mémoire
Le Model-Checking de CTL

BUI QUANG PHUONG Linh – 000427796
Promoteur : Prof. GEERAERTS Gilles
Année 2017-2018

Université Libre de Bruxelles

Résumé La vérification de modèles, plus communément appelée via son appellation anglaise *Model-Checking*, est un système de vérification automatique qu'un système informatique ou électronique satisfasse une certaine propriété. Celle-ci est généralement utilisé afin de prouver la bonne fonctionnalité du système ou dans le cas contraire, de détecter des bugs ou des dysfonctionnements. Plusieurs types d'algorithmes et de logiques permettent d'effectuer un Model-Checking. Nous allons principalement nous intéresser au Model-Checking de CTL mis en place durant les années 80.

Table des matières

1 Introduction

1.1 Contexte et point de vue global

Les dernières décennies entrainant l'évolution exponentielle de la technologie et de l'informatique, la société actuelle fait régulièrement usage de systèmes automatisés afin d'assurer le bon fonctionnement et la fiabilité des programmes, machines et autres divers systèmes informatiques. Afin de vérifier ces systèmes, l'utilisation d'une technique algorithmique appelée *Model-Checking* va être mise en place.
Le principe de base du model-checking est la vérification qu'un certain système satisfasse une certaine propriété. Le système sera représenté par un modèle $\mathcal{M}$ tandis que la propriété fera office d'une formule Φ. Dans un premier temps, la modélisation du comportement dynamique du système est nécessaire avant d'être suivie par la vérification de la formule de la propriété grâce à l'algorithme de model-checking qui déterminera si le modèle satisfait bien la formule.

$$\boxed{\mathcal{M} \vDash \Phi}$$

Lorsque le système ne satisfait pas la propriété évaluée, le model-checking a donc repéré un bug ou un dysfonctionnement. Chaque année, le coût des bugs s'élève à environ 59 milliards de dollars [11] rien qu'aux Etats-Unis. La création et la mise en place de technique de vérification est donc impérative pour réduire le taux de bugs afin de minimiser ces coûts.

Illustrons cela par un exemple plus concret : un distributeur de boissons. Dans ce cas, il est indispensable que le montant demandé pour une certaine boisson soit mis dans la machine afin que le client puisse récupérer la boisson. La vérification de ce prérequis va être effectuée à l'aide d'un système de model-checking. Un autre exemple illustrant l'importance du model-checking est la fermeture des barrières sur une voie ferrée lors du passage d'un train. Celle-ci doit être réalisée automatiquement lorsqu'un train est sur le point de traverser une voie ferrée. Dans ce cas, une petite erreur de timing peut avoir une très grande ampleur concernant la sécurité des passants. Il est donc impératif de vérifier la bonne fonctionnalité du système gérant la fermeture des barrières grâce au model-checking.

1.2 Les phases du model-checking

Le processus de base d'un model-checking est divisé en 3 phases :

1. **La phase de modélisation** : permet de représenter le comportement du système. Dans le cas du model-checking, cette phase de modélisation est réalisée à l'aide d'automates d'états finis [1] et de logique temporelle [2]. Celle-ci permet une formalisation du système ainsi que de la propriété à vérifier. De plus, cette modélisation permet de réaliser les premières vérifications grâce à différentes simulations réalisées sur le modèle. Cette phase de modélisation est l'objet de la section 2 *"Algorithme de model checking"* et subsection 3.1 *"Introduction à la logique temporelle"*.
2. **La phase d'exécution** : exécution du model-checker vérifiant la validité de la propriété à satisfaire. Le model checking trouve principalement son intérêt dans sa vérification **automatique** et offre donc un réel avantage. (cfr. section 1.3)
3. **La phase d'analyse** : lorsque les résultats sont obtenus après exécution, différents cas potentiels existent :
   — Si la propriété est **satisfaite** : vérification de la propriété suivante (s'il y en a).
   — Si la propriété n'est **pas satisfaite** : soit un contre-exemple est produit qui décrit un scénario possible d'erreur (violation de la propriété), soit il y a correction du modèle et réexécution du model-checker, soit il y a réexécution de toute la procédure.
   — **Manque de mémoire** (cfr. *State explosion problem* [3]) : tentative de réduction du modèle notamment grâce aux diagrammes de décision binaire ou une réduction partielle de la commande et réexécution le model-checker.

1. cfr. section 2 "Algorithme de model checking"
2. cfr. subsection 3.1 "Introduction à la logique temporelle"
3. section 1.3 - 1er désavantage

Un schéma récapitulatif reprenant les différentes phases citées ci-dessus est présentée dans la Figure 1.

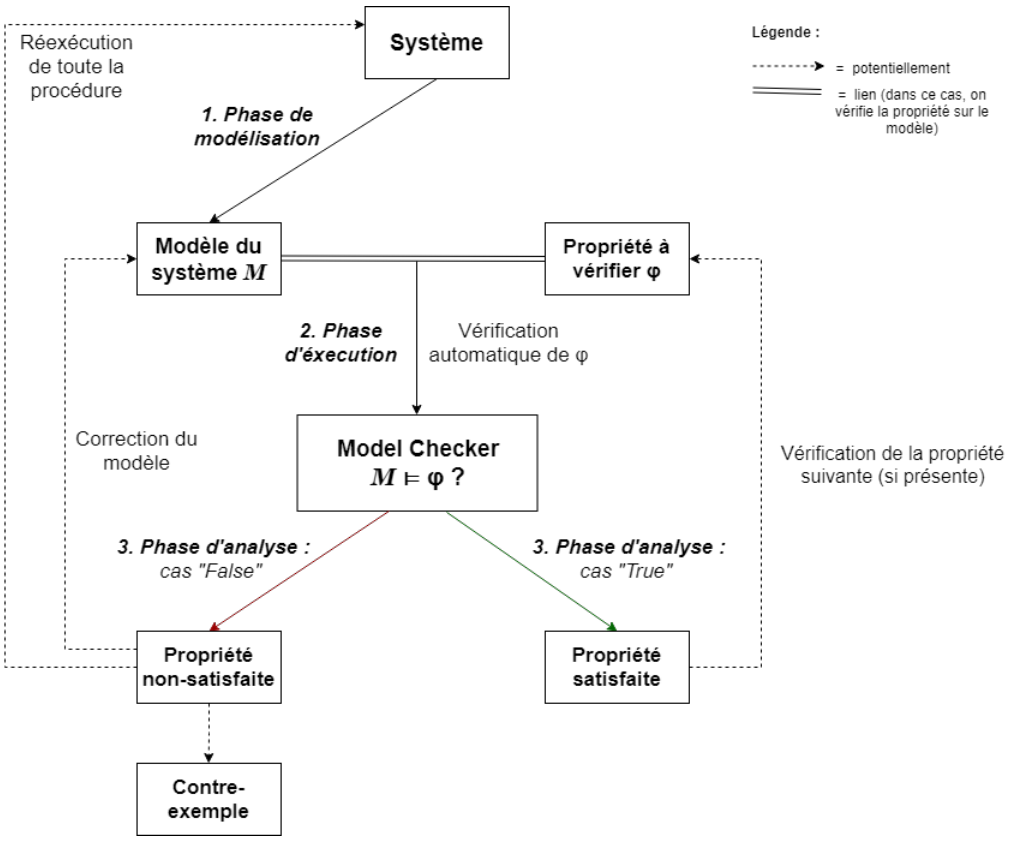

FIGURE 1: Schéma récapitulatif de l'approche du model-checking

1.3 Avantages & désavantages

Dans cette section, plusieurs avantages et inconvénients seront présentés sur base des recherches effectuées en 2008 par *Clarke* [8] ainsi que du livre de référence "Principles of Model Checking" [3].

Avantages du model-checking Le model-checking possède plusieurs avantages par rapport à l'utilisation d'éventuelles autres techniques de vérifications et de détection d'erreur. Voici une liste non exhaustive de ces avantages :

— Il s'agit d'une approche générale de vérification applicable dans tous les domaines tels que les systèmes embarqués ou ingénierie logicielle.
— La phase de vérification est automatique. Suivant la phase de modélisation, la seule action nécessaire de l'utilisateur afin de faire fonctionner le model-checking est de l'activer. Le programme s'occupe du reste. Niveau performance, cet automatisme permet un gain de temps conséquent.
— Contrairement aux simples tests, le model-checking va également prouver la bonne fonctionnalité du système et dans le cas contraire, nous renvoyer un contre-exemple d'une exécution du système qui falsifie la propriété et ainsi, détecter le(s) bug(s). Le taux d'erreurs non repérées est donc moindre par rapport aux tests simples.
— Le model-checking agit sur toute la modélisation du système et non qu'une seule partie, il s'agit donc d'une méthode exhaustive. Néanmoins, si l'utilisateur le souhaite, il est tout de

même possible d'utiliser le model-checking pour une vérification partielle et ainsi l'appliquer à des parties de la modélisation.
— Utilisation de la logique temporelle (*cfr. subsection 3.1 "Introduction à la logique temporelle"*) qui permet d'exprimer facilement les diverses propriétés de manière non ambigüe et universelle.

Désavantages du model-checking Néanmoins, le model-checking est une solution discutable sur certains points :

— Lorsqu'un problème est traité par la technique de model checking, plus un problème est grand, plus la taille du model checking va croître. Le modèle réalisé va donc grandir exponentiellement au nombre de processus actifs (appelés états). Pour N variables avec un domaine de k valeurs possibles, le nombre d'états grandit de k^N. Par exemple, pour 20 variables booléennes, il y aurait déjà 2^{20} soit 1048576 états. Le modèle devient donc très vite surchargé et surpasse la capacité de la mémoire. Ce problème est appelé *State explosion problem* [9]. Par conséquent, le model-checking ne pourrait donc pas croître. Cependant, plusieurs recherches [4] ont été établies dans le but d'établir une solution à ce problème tels que les *diagrammes de décisions binaires* permettant de représenter plusieurs états en un diagramme.
— Le model-checking s'applique généralement sur des systèmes finis et n'est pas adapté aux systèmes non-finis de part sa modélisation grâce à des automates d'états finis. Hors, la possibilité de tomber sur un système non-fini (ayant donc une infinité de valeurs possibles pour les variables) n'est pas écartée.
— Comme son nom l'indique, le model-checking exécute une vérification du modèle du système et non du système en lui-même. Le modèle étant limité à un système de transitions (cfr section 2 *"Algorithme de model checking"*), ce n'est donc pas une représentation exacte du système, certains éléments pourraient différés et/ou manqués, que ce soit au niveau du hardware tel que des défauts de fabrication ou bien au niveau software tel que des erreurs de codage. Pour palier à ce problème, des tests supplémentaires sont nécessaires.
— Conséquence du dernier point : le résultat n'est donc pas garanti à 100% et pourrait donc contenir des erreurs.

2 Algorithme de model checking

2.1 Graphes et principes d'états

Le model-checking base sa représentation sur un système de graphe orienté, plus précisément sur un *système d'états finis* où chaque nœud du graphe est appelé état tel que les arcs entre ces états sont appelés transitions. Cet ensemble d'états et de transitions décrit le comportement du système réactif et forme le **modèle** de ce système. Dans l'exemple du problème de distributeur de boissons abordé précédemment, un tel modèle se présenterait comme tel :

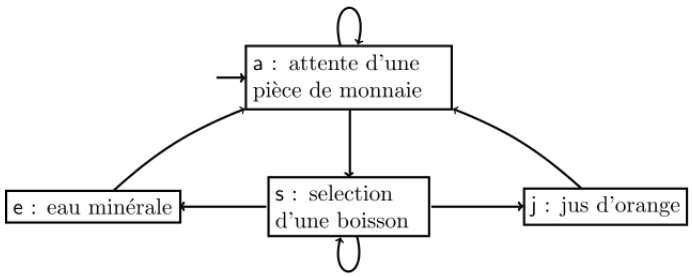

FIGURE 2: Modèle du problème d'un distributeur de boissons [16]

4. La section 5 "State Explosion Problem" de l'article de Clarke [9] reprend les principales recherches de solutions à ce problème.

où chaque état est défini par un nom (un alias). Par exemple, "a" définit l'attente de l'entrée d'une pièce de monnaie dans la machine.

Un modèle de ce type constitue la base d'une *structure de Kripke*. La section suivante présente une telle structure ainsi que l'adaptation du modèle de la Figure 2 en *modèle de Kripke*.

2.2 Structure de Kripke

Définition Une structure de *Kripke* [12] est un tuple :

$$\boxed{\mathcal{K} = (S, I, A, AP, \longrightarrow, \lambda)}$$

tel que :

- **S** est un ensemble fini d'états,
- $\boldsymbol{I \subseteq S}$ est l'ensemble des états initiaux,
- **A** est un ensemble d'actions,
- **AP** est un ensemble de propositions atomiques,
- $\boldsymbol{\longrightarrow \subseteq S \times A \times S}$ est une relation de transitions entre états,
- $\boldsymbol{\lambda : S \rightarrow 2^{AP}}$ est une fonction de labelisation (étiquetage) des états qui définit pour chaque état $\boldsymbol{s \in S}$ l'ensemble $\boldsymbol{\lambda(s)}$ de toutes les propositions atomiques $\boldsymbol{AP}$ qui sont valides dans cet état.

Exemple complet d'une structure de Kripke Toujours sur base du même problème de distributeur de boissons, le modèle de la Figure 2 peut-être repris et complété en se voyant attribuer des noms d'actions afin de pouvoir spécifier les transitions et par conséquent de respecter la structure de Kripke.

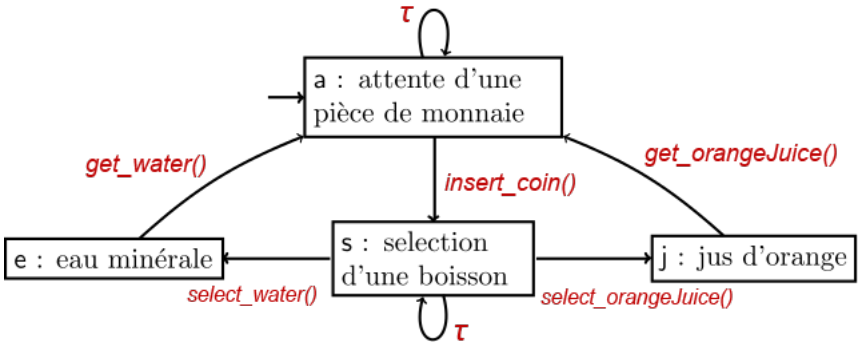

FIGURE 3: Modèle de Kripke complet du problème d'un distributeur de boissons

Dans ce cas, l'ensemble des états est $S = \{a, e, s, j\}$. L'ensemble des états initiaux $I = \{a\}$ ici ne comporte qu'un seul état initial et est représenté dans le modèle par un arc d'entrée vers l'état initial a.

Les labels associés à chaque transition correspond à une action et font donc partie de l'ensemble d'action A. Le symbole τ représente une action silencieuse qui désigne une action non observable mais qui n'est néanmoins pas à négliger tels que l'attente de la pièce ou une non-sélection de boisson.

L'ensemble d'action est donc $A = \{select\_water, select\_orangeJuice, insert\_coin, get\_orangeJuice, \tau\}$.

Les états et actions maintenant définis, des exemples de relations de transitions peuvent être présentés. Voici les transitions d'états lors de l'achat d'un jus d'orange dans le distributeur :

1. $\boldsymbol{a} \xrightarrow{\text{insert\_coin()}} \boldsymbol{s}$: représente la transition de l'état *attente d'une pièce de monnaie* à l'état *sélection d'une boisson* grâce à l'action d'insertion de pièce *insert_ coin*.
2. $\boldsymbol{s} \xrightarrow{\text{select\_orangeJuice()}} \boldsymbol{j}$: représente la transition de l'état *sélection d'une boisson* à l'état *jus d'orange* grâce à l'action *select_ orangeJuice*.
3. $\boldsymbol{j} \xrightarrow{\text{get\_orangeJuice()}} \boldsymbol{a}$: représente la transition de l'état *jus d'orange* à l'état *attente d'une pièce de monnaie* grâce à l'action *get_ orangeJuice*.

Ces 3 exemples de transitions font partie des 7 transitions d'états possibles du modèle.

Finalement, concernant le choix des propositions atomiques, le choix le plus simple est d'utiliser le nom des états comme propositions atomiques.
Pour un état s, la fonction de labelisation serait donc $\lambda(s) = \{s\} \cap AP$.

2.3 Arbre d'exécution d'une structure de Kripke

Définition Un *arbre d'exécution d'une struture de Kripke* correspond au "dépliage" du modèle de Kripke où les nœuds correspondent aux états tel que la racine est l'état initial du modèle de Kripke. Au niveau i, les fils d'un nœud sont les états successeurs au niveau $i+1$, c'est-à-dire les états subissant une transition à partir du nœud du niveau i.

Pourquoi utiliser un arbre ? L'utilisation d'un arbre permet de représenter tous les chemins de la structure de Kripke. Lorsque le graphe de Kripke est cyclique, l'arbre résultant est un arbre infini. C'est notamment de cette transformation en arbre que **le model checking CTL** (Computation Tree Logic) tient son nom vu qu'il se base sur la structure en arbre du système.

Illustration : d'un graphe à un arbre. Afin d'illustrer la transformation de la structure de Kripke à un arbre, reprenons une nouvelle fois l'exemple du distributeur de boissons dont le modèle de Kripke est présentée à la Figure 3 et transformons celui-ci en arbre.

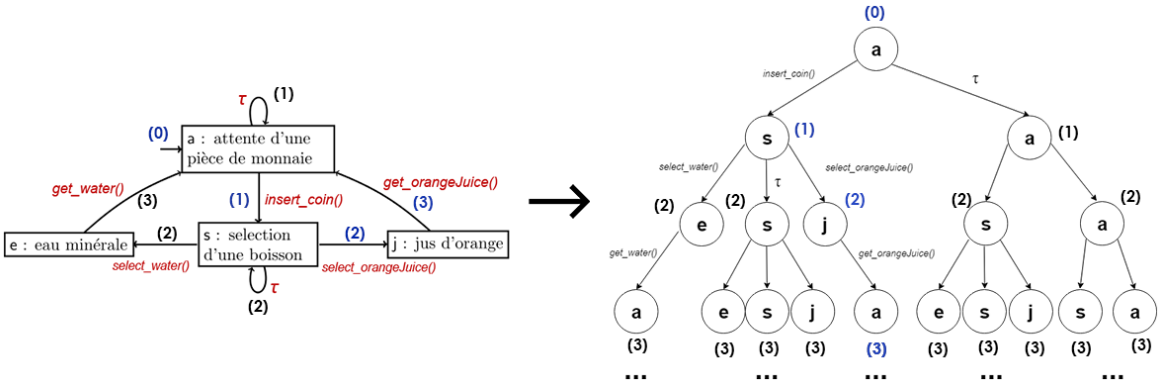

FIGURE 4: Etapes de construction de l'arbre à partir du graphe

La Figure 4 présente les différentes étapes de construction de l'arbre d'exécution de la structure de Kripke du problème du distributeur de boissons vu précédemment. La construction débute par l'état initial du système et va donc représenter la racine de l'arbre (Niveau 0). Le niveau suivant (Niveau 1) représente les états successeurs potentiels de l'état précédent, à savoir l'état intial 0.

Dans la même logique, le niveau 2 contient les états potentiels successeurs de ceux du niveau 1, et ainsi de suite pour tous les niveaux. En guise d'exemple d'exécution, l'achat d'un jus d'orange est représenté par les états dont leurs numéros sont bleus dans la Figure 4.

Tous les éléments présents dans le modèle de Kripke se retrouvent bien dans l'arbre. Dans ce cas, il s'agit bien d'un arbre infini résultant d'un graphe cyclique dont les 4 premiers niveaux sont présentés dans la Figure 4.

3 Le model-checking de CTL : généralités et logiques

3.1 Introduction à la logique temporelle

Les différentes propriétés du système à vérifier sont exprimés via la notion de *logique temporelle*. Celle-ci forme le deuxième élément complémentaire aux systèmes d'états finis (modèle de Kripke) permettant la réalisation de l'étape de modélisation du model checking.

La logique temporelle définit les propriétés temporelles à l'aide de connecteurs temporels [5] ("until", "next", "always", ...) et de quantificateurs sur les états ou les chemins de son graphe. Cette logique permet donc d'exprimer et vérifier des propriété de sûreté (absence de bugs), d'absence de blocages de l'exécution, d'invariance (tous les états satisfont une propriété) ou d'équité (fonctionnel et répétable infiniment) et remplit donc parfaitement le rôle de vérification du model checking.

Les deux principaux types de logiques temporelles qui seront abordés prochainement sont **la logique temporelle linéaire** (LTL) et **la logique temporelle arborescente** (CTL). Cependant, il existe également d'autres types de logiques temporelles tels que CTL* (fusion entre LTL et CTL), mucalcul, ForSpec, PSL, ...

3.2 Un premier aperçu de LTL et CTL

Linear Temporal Logic - LTL La logique temporelle linéaire ne quantifie pas sur les états et considère donc seulement des traces linéaires. Elle permet donc de décrire le comportement des systèmes réactifs au moyen des propriétés du système pour lesquels le temps se déroule linérairement. Elle étend la logique classique par des modalités temporelles dont certaines sont présentées dans la Figure 5. Celles-ci référencent vers différents moments dans le temps. Une bulle grisée exprime une propriété vérifiée à ce moment. Cet ensemble d'opérateurs consitue une partie de la sémantique de LTL. Cette logique ne sera pas abordée en détails dans ce document.

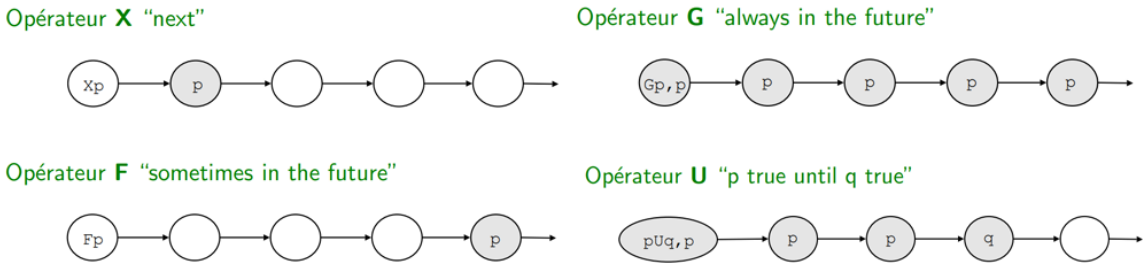

FIGURE 5: Modalités temporelles de LTL [4]

5. N.B : Il s'agit bien d'une logique temporelle et non temporisée, cette logique ne quantifie donc pas l'écoulement du temps mais décrit bien l'ordre des événements sans introduire la notion de temps explicitement.

Computation Tree Logic - CTL CTL est une logique temporelle basée sur la logique propositionnelle avec une notion discrète du temps dont le futur n'est pas déterminé. En effet, CTL exprime tous les chemins possibles d'un état à un autre grâce à sa structure d'arbre (notamment les arbres résultant du modèle de Kripke). En d'autres termes, la logique temporelle arborescente CTL peut être définie à partir d'un ensemble de variables propositionnelles AP, des connecteurs classiques de la logique propositionnelle tels que l'implication, la conjonction ou la disjonction, de quantificateurs de chemins (E ou A) et de connecteurs temporels (X,F,G,U). La syntaxe et sémantique de CTL fera l'objet des subsection 3.3 et subsection 3.4.

3.3 Syntaxe de CTL

Avant de décrire la syntaxe de CTL, la notion de *chemin* dans un système de transition (en particulier, celle de Kripke) est à définir :

Un **chemin** π dans un système de Kripke $\mathcal{K} = (S, I, A, AP, \delta, \lambda)$ est une séquence infinie d'états $s_0, s_1, \cdots \in S$ tel que $\forall i \geq 0 : s_i \rightarrow s_{i+1}$.

Comme expliqué précédemment, la logique temporelle arborescente CTL est basée sur la logique propositionnelle et définit donc chaque connecteur de cette logique ainsi que d'autres connecteurs temporels propres à CTL. Cet ensemble forme la syntaxe de la logique temporelle arborescente CTL et est présentée comme tel :

Soit :
$\mathbb{V}$ un ensemble de variables propositionnelles,
C un ensemble fini de connecteurs propositionnels : $C = \{\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow\}$,
T un ensemble fini de connecteurs temporels : $T = \{X, F, G, U, A, E\}$

Tel que :
$\mathbb{V} \subseteq CTL$ (variables propositionnelles sont donc des formules CTL),
ϕ, ϕ_1, ϕ_2 des propriétés quelconques $\in CTL$

Alors :

Syntaxe	Notation française	Signification
$\neg\phi_1$	NON ϕ_1	Négation de ϕ_1
$\phi_1 \wedge \phi_2$	ϕ_1 ET ϕ_2	Conjonction de ϕ_1 et ϕ_2
$\phi_1 \vee \phi_2$	ϕ_1 OU ϕ_2	Disjonction de ϕ_1 et ϕ_2
$\phi_1 \Rightarrow \phi_2$	ϕ_1 IMPLIQUE ϕ_2	Implication de ϕ_1 vers ϕ_2
$\phi_1 \Leftrightarrow \phi_2$	ϕ_1 EQUIVAUT ϕ_2	Double implication (équivalence) de ϕ_1 et ϕ_2

TABLE 1: Syntaxe de CTL - Logique propositionnelle

Syntaxe	Signification
$AX\phi$	Dans tous les chemins (A), ϕ est satisfait pour le prochain état du chemin. (X)
$AF\phi$	Dans tous les chemins (A), ϕ est satisfait pour au moins un état du chemin. (F)
$AG\phi$	Dans tous les chemins (A), ϕ est satisfait pour tous les états du chemin. (G)
$A(\phi_1 \cup \phi_2)$	Dans tous les chemins (A), ϕ_1 est satisfait jusqu'à ce que ϕ_2 devient valide $(\cup)$
$EX\phi$	Il existe un chemin (E) où ϕ est satisfait pour le prochain état du chemin. (X)
$EF\phi$	Il existe un chemin (E) où ϕ est satisfait pour au moins un état du chemin. (F)
$EG\phi$	Il existe un chemin (E) où ϕ est satisfait pour tous les états du chemin. (G)
$E(\phi_1 \cup \phi_2)$	Il existe un chemin (E) où ϕ_1 est satisfait jusqu'à ce que ϕ_2 devienne satisfait $(\cup)$

TABLE 2: Syntaxe de CTL - Logique temporelle

La Table 1 reprend les connecteurs principaux de la logique propositionnelle tandis que la Table 2 correspondant à de la logique temporelle. Cette dernière est divisée selon les quantificateurs de chemins en deux parties :

— **A** pour "**A**ll" : la propriété est satisfaite sur tous les chemins possibles (= inévitable).
— **E** pour "**E**xists" : la propriété est satisfaite sur au moins un chemin (= possible).
Chacune de ses parties possède 4 types de connecteurs :
— **X** pour "ne**X**t" : la propriété doit être satisfaite pour le prochain état du chemin.
— **F** pour "**F**inally" : la propriété est satisfaite pour au moins un état du chemin.
— **G** pour "**G**lobally" : la propriété est satisfaite pour tous les états du chemin.
— U pour "**U**ntil" : la propriété doit être satisfaite jusqu'à ce que la deuxième est "déclenchée" et devient donc satisfaite.

La logique temporelle présentée à la Table 2 est illustrée dans la Figure 6. Les propositions de l'état initial de chaque système sont considérés comme étant vraies.

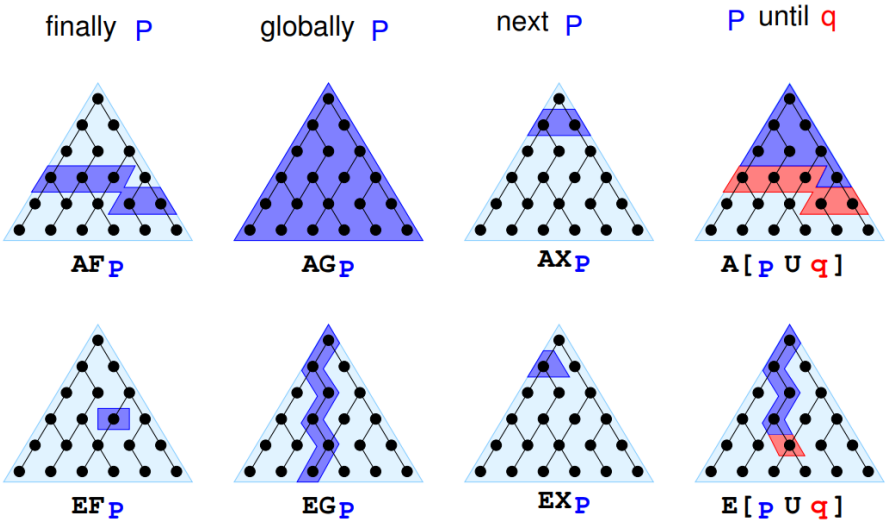

FIGURE 6: Illustration de la logique temporelle de CTL [2]

3.4 Sémantique de CTL

Les formules CTL sont interprétées par rapport aux états et aux chemins d'un système de transition (qui suit la structure de Kripke [6]). Généralement, la sémantique de CTL aborde donc une formule CTL (propriété) ϕ par deux relations de satisfactions, une pour les états s et l'autre pour les chemins [7] $\pi \in \mathbb{P}$, où $\mathbb{P}$ définit l'ensemble des chemins $\{\pi_0, \pi_1, \ldots, \pi_i\}$ d'une structure de Kripke :

1. Pour un état : $(\mathcal{K}, s) \vDash \phi \Leftrightarrow s$ satisfait ϕ
2. Pour un chemin : $(\mathcal{K}, \pi) \vDash \phi \Leftrightarrow \pi$ satisfait ϕ

où $\mathcal{K} = (S, I, A, AP, \delta, \lambda)$ est une structure de Kripke.

Ces relations de satisfactions sont notamment définies par la logique propositionnelle :

$$\begin{array}{lcl} (\mathcal{K}, s) \vDash \neg\phi & \Leftrightarrow & \neg s \vDash \phi \\ (\mathcal{K}, s) \vDash \phi_1 \wedge \phi_2 & \Leftrightarrow & (s \vDash \phi_1) \wedge (s \vDash \phi_2) \\ (\mathcal{K}, s) \vDash \phi_1 \vee \phi_2 & \Leftrightarrow & (s \vDash \phi_1) \vee (s \vDash \phi_2) \end{array}$$

De plus, pour vérifier la satisfaction d'une propriété ϕ d'une structure de Kripke $\mathcal{K}$, la logique temporelle caractérise également la sémantique :

6. cfr. subsection 2.2
7. Rappel : la notion de chemin est définie dans la subsection 3.3

$$
\begin{array}{lcl}
(\mathcal{K}, s) \vDash AX\phi & \Leftrightarrow & \forall \pi \in \mathbb{P} \text{ tel que } s_1 \vDash \phi \\
(\mathcal{K}, s) \vDash AF\phi & \Leftrightarrow & \forall \pi \exists i \geq 0 \text{ tel que } \pi_i \vDash \phi \text{ avec } \pi_i \in \mathbb{P} \\
(\mathcal{K}, s) \vDash AG\phi & \Leftrightarrow & \forall \pi \forall i \geq 0 : \pi_i \vDash \phi \text{ avec } \pi_i \in \mathbb{P} \\
(\mathcal{K}, s) \vDash A(\phi_1 \cup \phi_2) & \Leftrightarrow & (\forall \pi \exists i \geq 0 \text{ tel que } \pi_i \vDash \phi_2) \wedge (\forall j \in 0 \leq j < i : \pi_j \vDash \phi_1) \text{ avec } \pi_{i,j} \in \mathbb{P} \\
(\mathcal{K}, s) \vDash EX\phi & \Leftrightarrow & \exists \pi \in \mathbb{P} \text{ tel que } s_1 \vDash \phi \\
(\mathcal{K}, s) \vDash EF\phi & \Leftrightarrow & \exists \pi \text{ tel que } \exists i \geq 0 \text{ tel que } \pi_i \vDash \phi \text{ avec } \pi_i \in \mathbb{P} \\
(\mathcal{K}, s) \vDash EG\phi & \Leftrightarrow & \exists \pi \text{ tel que } \forall i \geq 0 : \pi_i \vDash \phi \text{ avec } \pi_i \in \mathbb{P} \\
(\mathcal{K}, s) \vDash E(\phi_1 \cup \phi_2) & \Leftrightarrow & (\exists \pi \text{ tel que } \exists i \geq 0 \text{ tel que } \pi_i \vDash \phi_2) \wedge (\forall j \in 0 \leq j < i : \pi_j \vDash \phi_1) \text{ avec } \pi_{i,j} \in \mathbb{P}
\end{array}
$$

Equivalence sémantique Deux formules CTL ϕ_1 et ϕ_2 sont équivalents, dénoté $\phi_1 \equiv \phi_2$, lorsqu'ils sont sémantiquements identiques, c'est-à-dire que pour chaque état s : $s \vDash \phi_1 \Leftrightarrow s \vDash \phi_2$.

Exemples d'équivalence sémantique

— Les lois de De Morgan formulés en CTL :

$$
\begin{array}{c}
\neg AF\phi \equiv EG\neg\phi \\
\neg EF\phi \equiv AG\neg\phi \\
\neg AX\phi \equiv EX\neg\phi
\end{array}
$$

— Les lois d'expansions :

$$
\begin{array}{c}
AG\phi \equiv \phi \wedge AXAG\phi \\
EG\phi \equiv \phi \wedge EXEG\phi \\
AF\phi \equiv \phi \vee AXAF\phi \\
EF\phi \equiv \phi \vee EXEF\phi \\
A(\phi_1 \cup \phi_2) \equiv \phi_2 \vee (\phi_1 \wedge AXA(\phi_1 \cup \phi_2) \\
E(\phi_1 \cup \phi_2) \equiv \phi_2 \vee (\phi_1 \wedge EXE(\phi_1 \cup \phi_2)
\end{array}
$$

Sûreté et vivacité d'un système Deux définitions importantes concernant le bon fonctionnement d'un système peuvent maintenant être exprimées :

1. Un système est dit **sûr** lorsqu'il est garanti que quelque chose de "mauvais" ne va jamais se produire et peut être exprimer par la formule CTL $AG\neg bad$. Si ce n'est pas le cas alors il existe un nombre fini de contre-exemples.
2. Un système est dit **vivant** lorsqu'il est garanti que quelque chose de "bon" va toujours se produire et peut être exprimer par la formule CTL $AGAFgood$. Si ce n'est pas le cas alors il existe un nombre fini de contre-exemples.

3.5 Comparaison LTL/CTL

Caractéristiques générales

— LTL est limité à une représentation linéaire du système mais ses connecteurs temporels permettent d'exprimer le futur ainsi que le passé notamment grâce à ses extensions de formules. Par exemple, l'opérateur G "Always in the future" devient G^{-1} "Always in the past". Chacun de ses opérateurs possède sa variante exprimant le passé.

— CTL offre une représentation complète en arbre présentant tous les chemins possibles et donc une grand nombre de propriétés potentiels exprimés, ce qui fait donc de lui un model-checking efficace. Néanmoins, les connecteurs temporels utilisés ne permettent d'exprimer uniquement le temps futur.

Expressivité

— LTL décrit les propriétés d'une exécution à la fois. LTL est donc très expressif sur un chemin mais il n'y a aucune expressivité sur les futures possibles.

— CTL raisonne sur un comportement arborescent en considérant plusieurs exécutions possible en même temps. CTL pourrait donc manquer d'expressivité linéaire mais est très expressif concernant les futures potentiels.

De ce fait, certaines propriétés CTL (LTL) ne sont pas expressibles en LTL (CTL) ou est considéré d'une autre façon d'une logique à une autre. Pour citer des exemples :

- $AGEF\phi$ (appelé *"reset property"*) est une formule CTL qui n'est pas expressible en LTL.
- La formule CTL $AFAXp$ distingue les deux systèmes présentés dans la Figure 7 tandis que la formule LTL FXp ne le distingue pas.

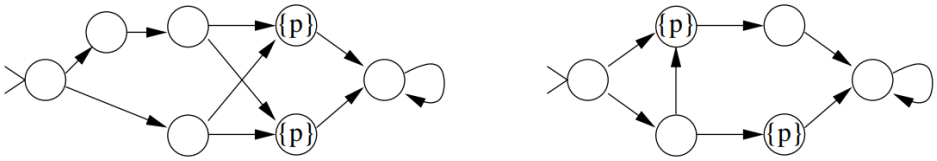

FIGURE 7: Exemple 2 - Expressivité CTL/LTL

- La formule LTL FGp n'est pas expressible en CTL comme illustré dans la Figure 8.

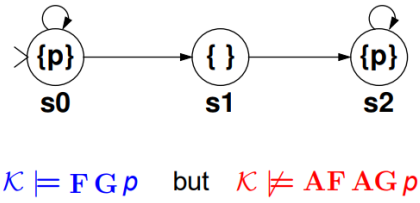

FIGURE 8: Exemple 3 - Expressivité CTL/LTL

L'utilisation des deux types de logiques temporelles LTL et CTL est donc nécessaire ; l'expressivité de ces deux logiques n'est pas comparable, elles sont complémentaires : chaque logique possède des propriétés que l'autre ne peut pas exprimer.

Complexité théorique

⚠*Pour rappel, les notions de théorie de complexité abordés dans cette section sont présentées dans l'annexe section A.*

— D'une part, la complexité en espace mémoire des problèmes de model checking utilisant LTL est PSPACE-complet. D'autre part, la complexité en temps de l'algorithme du model checking de LTL est en $\mathcal{O}(|TS| \cdot exp(|\phi|))$ où $|TS|$ correspond à la taille du système de transition et $|\phi|$ la taille de la propriété à vérifier.

— La complexité en temps des problèmes de model checking utilisant CTL est PTIME. Plus particulièrement, la complexité de l'algorithme du model checking de CTL est en $\mathcal{O}(|TS| \cdot |\phi|)$ où $|TS|$ correspond à la taille du système de transition et $|\phi|$ la taille de la propriété à vérifier.

En pratique

— LTL (étendu avec des expressions régulières) est généralement utilisé dans l'industrie des processeurs. Il est notamment très intuitif au niveau des contres-exemples et de l'interface et permet la vérification au run-time par tests. LTL vérifie donc des propriétés compliquées mais uniquement sur des parties de système.

— CTL est utilisé pour une vérification de propriétés simples sur un système entier et donc en d'autres termes, effectue une vérification du modèle.

4 Le model-checking CTL : algorithme et implémentation

4.1 Présentation et analyse de l'algorithme

L'algorithme de model checking de CTL qui va être présenté est le premier algorithme à avoir été développé, à savoir le model checking de CTL par labelling.

En quoi consiste l'algorithme par labelling ? Premièrement, une structure de Kripke $\mathcal{K} = (S, I, A, AP, \delta, \lambda)$ ainsi qu'une certaine propriété ϕ à vérifier (formule CTL) sont prises en entrée. Comme son nom l'indique, l'algorithme par labelling se base sur des marquages tout au long de son exécution. Il est impossible de vérifier la propriété directement (sauf s'il s'agit d'une formule banale tels que $AF\phi$, $AX\phi$, ...), il est donc nécessaire de traiter les sous-formules ϕ' de ϕ. Le marquage va donc s'opérer sur les états s de $\mathcal{K}$ pour lesquels la sous-formule ϕ' est vérifiée. Cette opération de marquage est caractérisée par la fonction `Marking(`$\mathcal{K}$`,`ϕ'`)` qui sera détaillée plus tard. S'en suit la même opération qui va être procédée récursivement pour les autres sous-formules. Finalement, $\mathcal{K}$ satisfait ϕ si et seulement si l'état initial s_0 est marqué par ϕ.

Formules CTL à implémenter Afin d'implémenter l'algorithme de model checking CTL par labelling, certaines formules CTL particulières sont à implémenter. Rappelons la liste des formules CTL [8] vues précédemment : $AX\phi$, $AF\phi$, $AG\phi$, $A(\phi_1 \cup \phi_2)$, $EX\phi$, $EF\phi$, $EG\phi$ et $E(\phi_1 \cup \phi_2)$ ainsi que les opérateurs de base en logique propositionnelle $\neg\phi$, $\phi_1 \wedge \phi_2$ et $\phi_1 \vee \phi_2$. Fort heureusement, uniquement les formules $EX\phi$, $E(\phi_1 \cup \phi_2)$ et $A(\phi_1 \cup \phi_2)$ sont essentielles à exprimer, du fait que les autres en découlent directement grâce aux relations d'équivalence.

$$AX\phi \equiv \neg EX\neg\phi$$
$$AF\phi \equiv A(True \cup \phi)$$
$$AG\phi \equiv \neg EF\neg\phi$$
$$EF\phi \equiv E(True \cup \phi)$$
$$EG\phi \equiv \neg AF\neg\phi$$

Un algorithme de marquage pour les opérateurs de la logique propositionnelle ainsi que pour les trois formules CTL essentielles citées ci-dessus sera donc présenté dans les sections qui suivent. Les algorithmes présentées sont adaptés sur base du document de Sébastien Bardin : "Introduction au Model Checking", réalisé en 2008. [5]

⚠ ***Remarque :*** Afin de clarifier les différents cas, la procédure de marquage a été divisée en différentes procédures mais il s'agit bien de la même fonction de marquage `Marking` qui regroupe les différents cas `MarkingNOT`, `MarkingAND`, `MarkingOR` ainsi que les autres procédures de marking concernant les formules CTL `MarkingEX`, `MarkingEU` et `MarkingAU`.

Opérateurs de la logique propositionnelle L'opération principale des différents algorithmes est évidemment le marquage des états qui vérifient la propriété ϕ. Dans ce cas, $\phi = \neg\phi'$, $\phi = \phi_1 \wedge \phi_2$ et $\phi = \phi_1 \vee \phi_2$. Pour rappel, la procédure prend en input la structure de Kripke $\mathcal{K} = (S, I, A, AP, \delta, \lambda)$ et la formule à vérifier ϕ. La notation $s.\phi$ fait le lien entre l'état s et la formule ϕ. Dans ce cas, l'état s est étiquetté par le label ϕ. En d'autres termes, si $s.\phi = True$ alors s vérifie la formule ϕ et inversément.

Algorithm 1 Fonction de marquage : cas 1 $\phi = \neg\phi'$

```
1: procedure MarkingNOT(K, φ)                          ▷ Cas 1 : φ = ¬φ'
2:     MarkingNOT(K, φ')   ▷ Appel récursif de la fonction de marquage sur la sous-formule φ'
3:     for each s in S do :
4:         s.φ := not(s.φ')          ▷ On redéfinit s.φ en l'assignant à la sous-formule ¬φ'
```

8. Rappel : cfr subsection 3.3 pour leurs significations

Algorithm 2 Fonction de marquage : cas 2 $\phi = \phi_1 \wedge \phi_2$

```
1: procedure MarkingAND(𝒦, φ)                          ▷ Cas 2 : φ = φ1 ∧ φ2
2:     MarkingAND(𝒦, φ1) ;        ▷ Appel récursif de la fonction de marquage sur la sous-formule φ1
3:     MarkingAND(𝒦, φ2) ;        ▷ Appel récursif de la fonction de marquage sur la sous-formule φ2
4:     for each s in S do :
5:         s.φ := and(s.φ1, s.φ2)   ▷ On redéfinit s.φ en l'assignant à la sous-formule φ1 ∧ φ2
```

Algorithm 3 Fonction de marquage : cas 3 $\phi = \phi_1 \vee \phi_2$

```
1: procedure MarkingOR(𝒦, φ)                           ▷ Cas 3 : φ = φ1 ∧ φ2
2:     MarkingOR(𝒦, φ1) ;         ▷ Appel récursif de la fonction de marquage sur la sous-formule φ1
3:     MarkingOR(𝒦, φ2) ;         ▷ Appel récursif de la fonction de marquage sur la sous-formule φ2
4:     for each s in S do :
5:         s.φ := or(s.φ1, s.φ2)    ▷ On redéfinit s.φ en l'assignant à la sous-formule φ1 ∨ φ2
```

***EXφ* : Exists Next** Concernant la formule CTL $EX\phi$, l'état qui nous intéresse est l'état s' suivant l'état s. Comme toutes les autres formules, l'appel récursif de la fonction de marquage avec la sous-formule est prise en entrée pour débuter. S'en suit deux boucles, l'une permettant d'initialiser tous les états comme étant des états qui ne vérifient pas la formule CTL ϕ et l'autre qui réalise l'opération permettant de constater si la sous-formule $EX\phi$ est vérifiée. Cette deuxième boucle va donc parcourir toutes les paires d'états (s, s') en vérifiant si l'état s' qui suit l'état s vérifie la sous-formule ϕ'. Si c'est le cas, alors l'état s vérifie bien la formule CTL $EX\phi$.

Algorithm 4 Fonction de marquage : cas 4 $\phi = EX\phi'$

```
1: procedure MarkingEX(𝒦, φ)                           ▷ Cas 4 : φ = EXφ'
2:     MarkingEX(𝒦, φ') ;         ▷ Appel récursif de la fonction de marquage sur la sous-formule φ'
3:     for each s in S do :                                  ▷ Initialisation des états
4:         s.φ := false
5:     for each (s, s') ∈→ do :   ▷ Parcours des paires d'états (s,s') où s' est l'état successeur de s
6:         if s'.φ' = true then :                ▷ Vérification que l'état successeur vérifie φ'
7:             s.φ := true
```

***E*(*φ*1 ∪ *φ*2) : Exists *φ*1 Until *φ*2** $E(\phi_1 \cup \phi_2)$ est une formule s'intéressant à deux états tout comme $EX\phi$ mais cette fois, il ne s'agit pas nécessairement de l'état suivant mais de tous les états futurs à l'état courant s. L'élément qui nous intéresse est l'état qui effectue un changement de sous-formule ϕ_1 vers ϕ_2. Pour ce faire, une liste `marked` retenant les des états qui ont été étiquettés ϕ_2 ainsi que les états $\phi 1$ dont un de ses états successeurs s' est étiquetté ϕ_2 est nécessaire.
De la même manière que les derniers algorithmes présentés, celui-ci est caractérisé par l'appel récursif de la fonction de marquage sur les sous-formule ainsi que d'une initialisation des états $s.\phi$ qui sont mis à `false` ainsi que de la liste `marked`. Le corps de l'algorithme est formé d'une première boucle parcourant tous les états s afin de ne retenir que les états étiquettés par ϕ_2. Une deuxième boucle réalise l'étiquetage de ces états à la formule CTL de base ϕ. Finalement, une dernière boucle imbriquée à la seconde est effectuée afin d'étiquetter les états s respectant la formule ϕ, c'est-à-dire pour lesquels un des états successeurs s' a été étiquetté ϕ.

Algorithm 5 Fonction de marquage : cas 5 $\phi = E(\phi_1 \cup \phi_2)$

```
1: procedure MarkingEU(K, φ)                          ▷ Cas 5 : φ = φ = E(φ1 ∪ φ2)
2:     MarkingEU(K, φ1) ;        ▷ Appel récursif de la fonction de marquage sur la sous-formule φ1
3:     MarkingEU(K, φ2) ;        ▷ Appel récursif de la fonction de marquage sur la sous-formule φ2
4:     marked := {}
5:     for each s in S do :                                  ▷ Initialisation des états
6:         s.φ := false
7:         s.seenBefore := false
8:     for each s' in S do :                  ▷ Récupération des états étiquettés par φ2
9:         if s'.φ2 = true then :                ▷ Vérification que l'état vérifie φ2
10:            marked += s'
11:    do
12:        chose s' ∈ marked
13:        marked -= s' ; s'.φ = true
14:        for each (s, s') ∈→ do   ▷ On vérifie que l'état s prédécesseur à s' vérifient bien φ1
15:            if s.seenBefore = false then                ▷ Si s n'a pas encore été traité
16:                s.seenBefore = true
17:                if s.φ1 = true then
18:                    marked += s                  ▷ ajouté à marked pour être etiquetté s.φ
19:    while marked ≠ {}
```

$A(\phi_1 \cup \phi_2)$: Always ϕ_1 Until ϕ_2 L'algorithme de $A(\phi_1 \cup \phi_2)$ ressemble vraisemblablement à celui de $A(\phi_1 \cup \phi_2)$. La seule différence est qu'il est indispensable que **tous** les états s précédant un état s' vérifiant ϕ_2 soient vérifiés par ϕ_1. Cette condition va se traduire par une variable `s.degree` qui correspond au degré (niveau) de l'état s. Un degré 0 équivaut à la racine et à l'état initial s_0. L'objectif est que tous les degrés précédant celui de s' (donc jusqu'à l'état initial, degré 0) soient vérifiés par ϕ_1. (cfr. ligne 16 dans Algorithm 6)

Algorithm 6 Fonction de marquage : cas 6 $\phi = A(\phi_1 \cup \phi_2)$

```
1: procedure MarkingAU(K, φ)                          ▷ Cas 6 : φ = A(φ1 ∪ φ2)
2:     MarkingAU(K, φ1) ;        ▷ Appel récursif de la fonction de marquage sur la sous-formule φ1
3:     MarkingAU(K, φ2) ;        ▷ Appel récursif de la fonction de marquage sur la sous-formule φ2
4:     marked := {}
5:     for each s in S do :                                  ▷ Initialisation des états
6:         s.φ := false
7:         s.degree := degree(s)
8:     for each s' in S do :                  ▷ Récupération des états étiquettés par φ2
9:         if s'.φ2 = true then :                ▷ Vérification que l'état vérifie φ2
10:            marked += s'
11:    do
12:        chose s' ∈ marked
13:        marked -= s' ; s'.φ = true
14:        for each (s, s') ∈→ do   ▷ On vérifie que les états s prédécesseurs à s' vérifient bien φ1
15:            s.degree -= 1 ;
16:            if (s.degree = 0) and (s.φ1 = true) and (s.φ = false) then   ▷ Tous les préc. traités...[9]
17:                marked += s                  ▷ ajouté à marked pour être etiquetté s.φ
18:    while marked ≠ {}
```

4.2 Résultats

Afin d'illustrer les algorithmes abordés, une exécution de chaque algorithme sur la structure de Kripke (modélisation) du problème de distributeur de boissons va être présentée.

9. Si on a bien traité tous les états précédents l'état s' et que l'état initial vérifie ϕ_1

⚠ ***Remarque :*** Pour rappel, **l'exécution des algorithmes présentés se font sur la structure de Kripke passée en entrée.** Uniquement par souci de visibilité, les actions des transitions entre les états ne sont pas annotés. De plus, l'annexe section B présente ces algorithmes sur les 3 premiers niveaux du dépliage en arbre (degrés) de la structure de Kripke afin d'avoir un autre point de vue (plus lisible) de l'exécution.

EXϕ La formule CTL que la fonction prend en entrée est ϕ qui correspond donc à $\phi = EX\phi'$. ϕ' correspond ici à la propriété lorsque le distributeur est dans le mode "sélection de boissons" (alias s). A la fin de l'exécution, et donc de la seconde boucle (représenté en vert dans la Figure 9), nous obtenons les états accepteurs représentés par un état "doublement encadré" qui vérifient la propriété ϕ, à savoir $\{s_1, a_1\}$ qui précèdent un état dont $\phi' = true$ ainsi que les transitions $X \xrightarrow{\text{action}} Y$ permettant la satisfaisabilité de ϕ sont mises en évidence où dans ce cas, X est un état précédant ϕ' et Y un état qui satisfait ϕ'.

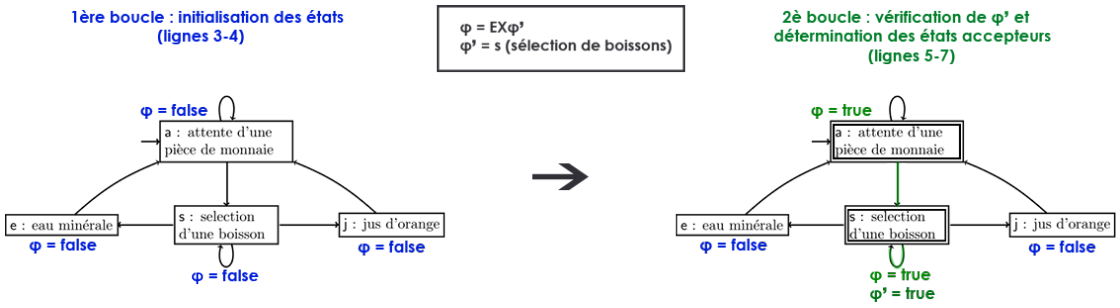

FIGURE 9: Exécution de l'algorithme 4 - $EX\phi$

E(ϕ₁ ∪ ϕ₂) La formule CTL que la fonction prend en entrée est ϕ qui correspond donc à $\phi = E(\phi_1 \cup \phi_2)$. ϕ_1 correspond ici à la propriété lorsque le distributeur est en attente de la monnaie (alias a) et ϕ_2 correspond au mode "sélection de boissons" (alias s). A la fin de la deuxième boucle (en bleu dans la Figure 13), `marked` contient les états qui vérifient ϕ_2, à savoir $\{s_i\}$ avec $0 \leq i < \inf$. S'en suit la boucle principale (en vert dans la Figure 10) qui va vérifier les états prédécesseurs à `marked` $= \{s_i\}$. Les états s_i vont être traités un par un et se voient retirés de `marked`. Les états satisfaisant ϕ_1 vont être ajoutés à `marked` afin de passer leur étiquette ϕ à `true` pour finalement être retirés et obtenir une liste *marked* vide qui marque la fin de la boucle.

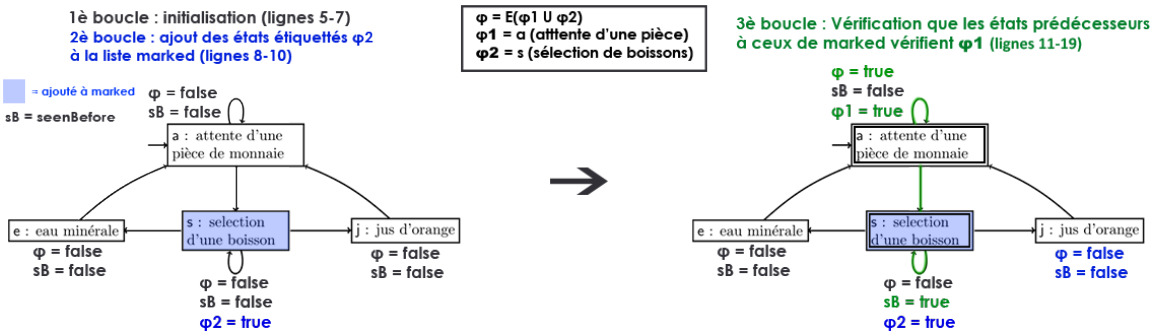

FIGURE 10: Exécution de l'algorithme 5 - $E(\phi_1 \cup \phi_2)$

$A(\phi_1 \cup \phi_2)$ La formule CTL que la fonction prend en entrée est ϕ qui correspond donc à $\phi = A(\phi_1 \cup \phi_2)$. ϕ_1 et ϕ_2 correspondent aux mêmes propriétés que précédemment pour $\phi = E(\phi_1 \cup \phi_2)$, à savoir respectivement a et s. L'exécution est la même pour les 2 premières boucles sauf que dans ce cas-ci, nous avons besoin d'ajouter une variable `deg` (degré) à chaque état comme expliqué précédemment. Les variables de degré d et k correspondent au nombre de transitions $a_i \longrightarrow a_{i+1}$ et $s_i \longrightarrow s_{i+1}$. A la fin de la seconde boucle, `marked` contient donc toujours $\{s_i\}$. La boucle suivante (en vert dans la Figure 14) tente de vérifier que tous les états prédécesseurs à ceux contenus dans `marked` vérifient ϕ_1. Les états a_i de la transition $a_i \xrightarrow{\text{insert\_coin()}} s_i$ ou de $a_i \longrightarrow a_{i+1}$ ne posent pas de soucis tandis que l'état prédecesseur s_j de s_i résultant de la transition $s_j \longrightarrow s_i$ (mis en évidence en rouge dans la Figure 11) ne satisfait pas ϕ_1 qui engendre donc la non satisfaisabilité de la propriété ϕ.

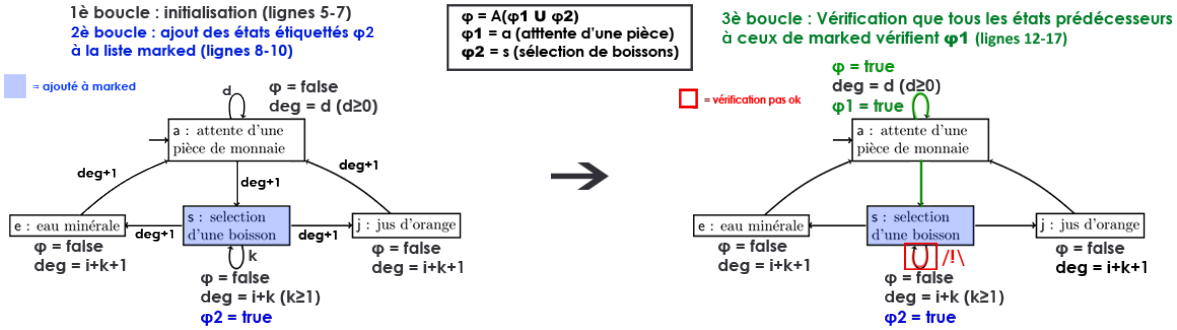

FIGURE 11: Exécution de l'algorithme 6 - $A(\phi_1 \cup \phi_2)$

4.3 Complexité

La complexité en temps des différents algorithmes présentés est caractérisée par des boucles parcourant la totalité des états du système, et serait donc en $\mathcal{O}(|TS|)$ avec $|TS|$ la taille du système. Néanmoins, comme cité précédemment, la complexité générale du model checking CTL ne se limite pas qu'à la taille du système mais dépends également de la taille de la formule CTL à vérifier. Dans notre cas, seules des formules basiques ont été traités, celles-ci prenant donc un temps de $\mathcal{O}(1)$ mais dans des cas plus complexes, ce temps est évidemment à prendre en compte, ce qui nous amène à rappeler la complexité générale du model checking CTL qui est en $\mathcal{O}(|TS| \cdot |\phi|)$.

5 Conclusion

En conclusion, les bases de l'algorithme du model checking et plus particulièrement du model checking de CTL ont été vus. Le model checking CTL s'avère être une méthode de vérification automatique efficace, ceci se remarquant notamment à l'aide de l'algorithme par labelling présentée dans la section précédente. En effet, un avantage majeur de l'algorithme par labelling du model checking CTL est qu'il tourne en temps linéaire en chacune des entrées (structure de Kripke et formule CTL). L'algorithme repose sur le fait que toute formule CTL peut s'exprimer par un nombre restreint de formules sur les états. Cela nous permet de raisonner en termes d'états (satisfaisant la formule) plutôt que d'exécutions.

Néanmoins, nous avons vu que le model checking ne présente pas uniquement que des avantages et certains inconvénients tel que le *state explosion problem* s'avèrent être impactant au niveau des performances du model checking. Malgré que certaines recherches aient déjà eu lieu, le model checking ne reste cependant pas parfait et peut donc faire objet de recherches supplémentaires quant à l'amélioration de celui-ci.

Annexe A Théorie de la complexité

A.1 Qu'est-ce que la théorie de la complexité ?

La théorie de la complexité algorithmique s'intéresse à l'estimation de l'efficacité des algorithmes et donc à l'étude formelle de la difficulté des problèmes en informatique. Deux types de complexité existent :

- la complexité en **temps** : se base essentiellement sur le nombre d'opérations élémentaires (boucles, conditions, ...) pour traiter une donnée de taille n.
- la complexité en **espace mémoire** : évalue l'espace mémoire utilisé en fonction de la taille des données.

A.2 Les machines de Turing déterministes et non déterministes

Machines de Turing déterministes Les machines de Turing déterministes font toujours un seul calcul à la fois. Ce calcul est constitué d'étapes élémentaires ; à chacune de ces étapes, pour un état donné de la mémoire de la machine, l'action élémentaire effectuée sera toujours la même.

Machines de Turing non déterministes Une machine de Turing non déterministe est une variante théorique à une machine de Turing habituelle, qui, elle, est déterministe, mais s'en différencie dans le fait qu'étant non déterministe elle peut avoir plusieurs transitions activables, pour un état donné. À chaque étape de son calcul, cette machine peut donc effectuer un choix non-déterministe : elle a le choix entre plusieurs actions, et elle en effectue une.

A.3 Complexités PTIME & PSPACE(-complet)

PTIME La classe P (ou PTIME) est la classe des problèmes de décision pour lesquels il existe un algorithme de résolution polynomial en temps (donc généralement rapide) par une machine de Turing déterministe. Autrement dit,

$$PTIME = \bigcup_{k \in \mathbb{N}} TIME(n^k)$$

où n est la taille de l'entrée et $TIME(t(n))$ est la classe des problèmes de décision qui peuvent être décidés en temps de l'ordre de grandeur de $t(n)$.

PSPACE PSPACE est la classe de complexité des problèmes de décision qui peuvent se résoudre sur une machine de Turing déterministe avec un espace polynomial. En d'autres termes,

$$PSPACE = \bigcup_{k \in \mathbb{N}} SPACE(n^k)$$

où n est la taille de l'entrée et $SPACE(n^k)$ est l'ensemble des problèmes de décision décidés par des machines de Turing déterministes utilisant un espace $t(n)$.

PSPACE-complet Un problème A est dit PSPACE-complet si les deux conditions suivantes sont remplies :

- le problème A est dans la classe PSPACE, c'est-à-dire $A \in PSPACE$,
- tout problème PSPACE se réduit polynomialement à A, c'est-à-dire $\forall B \in PSPACE$: $B \leq_p A$. Si le problème A vérifie cette condition, il est alors dit **PSPACE-dur**.

Annexe B Exécution des algorithmes par labelling du MC CTL sur le dépliage en arbre de la structure de Kripke

B.1 $EX\phi$

Etats accepteurs = $\{s_1, a_1\}$

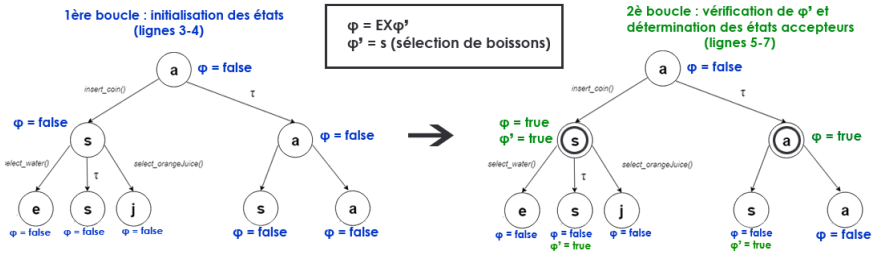

FIGURE 12: Exécution de l'algorithme 4 - $EX\phi$

B.2 $E(\phi_1 \cup \phi_2)$

Etats accepteurs = $\{a_0, s_1, a_1, s_2'\}$

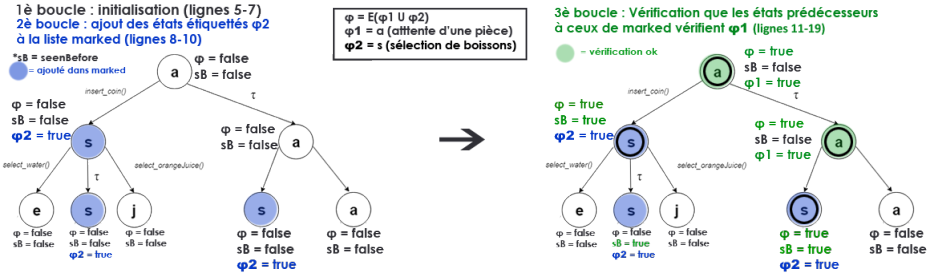

FIGURE 13: Exécution de l'algorithme 5 - $E(\phi_1 \cup \phi_2)$

B.3 $A(\phi_1 \cup \phi_2)$

Etats accepteurs = $\{\varnothing\}$

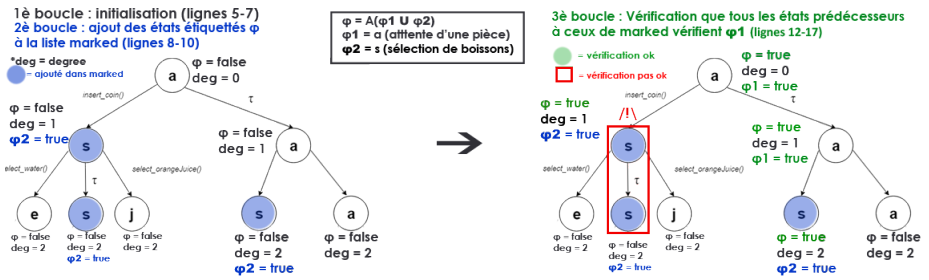

FIGURE 14: Exécution de l'algorithme 6 - $A(\phi_1 \cup \phi_2)$

Bibliographie

[1] (2008). Informatique théorique : Décidabilité et complexité. *IUT de Nice Côte d'Azur*.

[2] Artale, A. (2010). Formal methods, lecture iv : Computation tree logic (ctl). `http://www.inf.unibz.it/~artale/FM/slide4.pdf`.

[3] Baier, C. et Katoen, J.-P. (2008). *Principles of Model Checking*. The MIT Press. Cambridge, Massachusetts.

[4] Bardin, S. (2008a). Cours de model checking, leçon 2 : Logiques temporelles. `http://sebastien.bardin.free.fr/MC-cours2.pdf`.

[5] Bardin, S. (2008b). Introduction au model checking. *ENSTA*.

[6] Bourdil, P.-A. (2015). Contribution à la modélisation et la vérification formelle par model checking - symétries pour les réseaux de petri temporels. *Université de Toulouse*.

[7] Charfeddine, M. (2008). La logique temporelle.

[8] Clarke, E. M. (2008). The birth of model checking. *Carnegie Mellon University, Department of Computer Science. Pittsburgh, PA, USA*.

[9] Edmund M. Clarke, William Klieber, M. N. P. Z. (2012). Model checking and the state explosion problem. *Carnegie Mellon University & ETH Zürich*.

[10] Finis, J. (2012). Incremental model checking of recursive kripke structures. *Augsburg University*.

[11] Freyja (2017). How much could software errors be costing your company ? `https://raygun.com/blog/cost-of-software-errors/`.

[12] Kripke, S. A. (1963). Semantical considerations on modal logic. *Acta philosophica fennica*, 16(1963) :83–94.

[13] Kuhtz, L. (2010). Model checking finite paths and trees. *Universität des Saarlandes, Saarbrücken*.

[14] Merz, S. (2010). An introduction to model checking.

[15] Schwoon, S. (2002). Model-checking pushdown systems. *Lehrstuhl für Informatik VII der Technischen Universität München*.

[16] Wikipedia (2018). Vérification de modèles. `https://fr.wikipedia.org/wiki/V%C3%A9rification_de_mod%C3%A8les`.