Mini Mémoire
Le Model-Checking de CTL

BUI QUANG PHUONG Linh – 000427796
Promoteur : Prof. GEERAERTS Gilles

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.

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

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 [6] 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 effectué 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 *"Point de vue algorithmique"* 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.
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 (si 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 "Point de vue algorithmique"
2. cfr. subsection 3.1 "Introduction à la logique temporelle"
3. section 1.3 - 1er désavantage

L'approche de model checking comprenant ses différents intérêts et différents phases est illustrée dans la Figure 1.

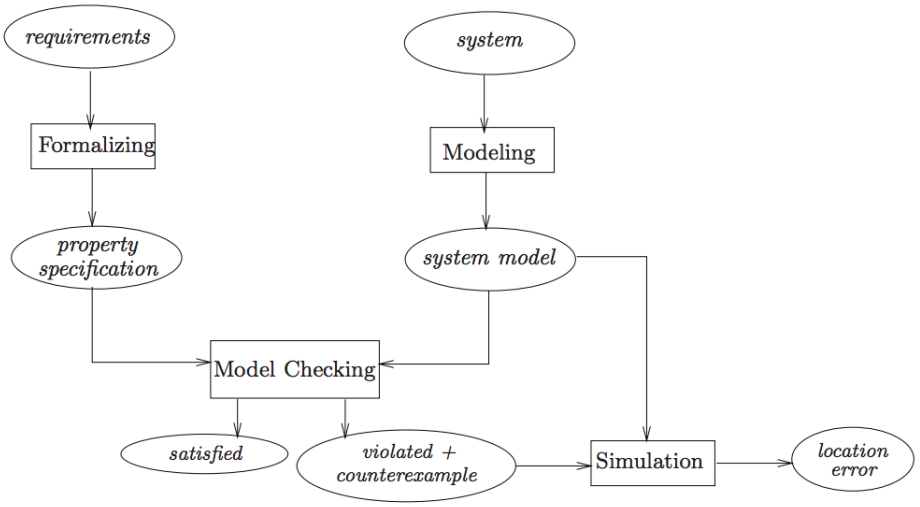


FIGURE 1: Vue schématique de l'approche du model-checking [4]

1.3 Avantages & désavantages

Dans cette section, plusieurs avantages et inconvénients seront présentés sur base des recherches effectuées en 2008 par *Clarke* [3] ainsi que du livre de référence "Principles of Model Checking" [1].

Avantages du model-checking Un 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 repérés est donc moindre par rapport aux tests simples.
- Le model-checking agit sur tout le 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 système.
- 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 :

4. Source : Christel Baier and Joost-Pieter Katoen, Principles of Model Checking [1], p.8 figure 1.4

— Le model-checking grandit exponentiellement au nombre de processus actifs (appelés états, correspondant à la taille du système). 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* [4]. Par conséquent, le model-checking ne pourrait donc pas croître. Cependant, plusieurs recherches [5] 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. Les résultats obtenus pour le modèle correspondent en grande partie au système réel, néanmoins, il est nécessaire d'utiliser des techniques supplémentaires comme des tests pour trouver les éléments qui pourraient différer entre le modèle et le système réel, que ce soit au niveau du hardware tel que des défauts de fabrication ou bien au niveau software tel que des erreurs de codage.
— Conséquence du dernier point : le résultat n'est donc pas garanti à 100% et pourrait donc contenir des erreurs.

2 Point de vue algorithmique

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 noeud 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 :

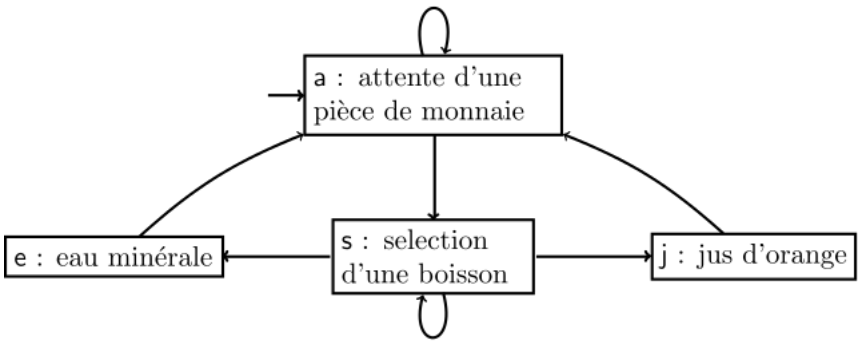


FIGURE 2: Modèle du problème d'un distributeur de boissons [6]

où chaque état est définie 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 suit la **structure de Kripke** et est donc appelé *modèle de Kripke*.

5. La section 5 "State Explosion Problem" de l'article de Clarke [4] reprend les principales recherches de solutions à ce problème.
6. Source : Wikipedia, "Vérifications de modèles"

2.2 Structure de Kripke

Définition *Kripke* définit sa structure, donnant nom à la structure de Kripke [7] :

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

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,
- $\boldsymbol{\delta \subseteq S \times A \times S}$ est une relation de transitions entre états,
- $\boldsymbol{\lambda : S \rightarrow 2^{AP}}$ est une fonction de labelisation (étiquetage) des états qui fait correspondre chaque état avec la proposition qui y est liée.

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.

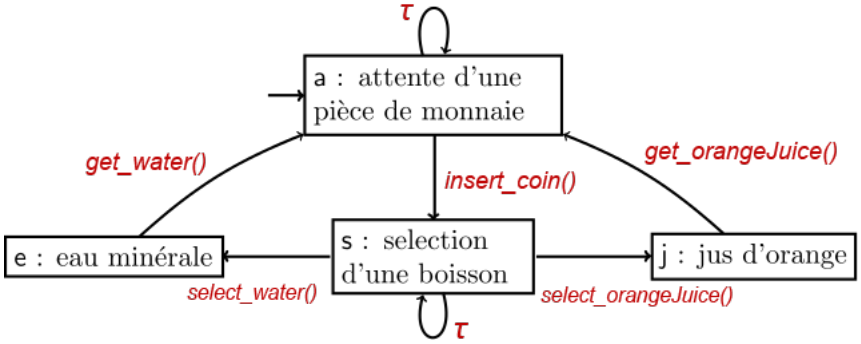


FIGURE 3: Modèle de Kripke complet du problème d'un distributeur de boissons

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 fait donc partie de l'ensemble d'action A. Le symbole τ représente une action qui n'est pas particulièrement intéressant à étiquetter 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 définies, 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. $\boldsymbol{\delta(a, insert\_coin, s)}$: 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. $\boldsymbol{\delta(s, select\_orangeJuice, j)}$: représente la transition de l'état *sélection d'une boisson* à l'état *jus d'orange* grâce à l'action *select_orangeJuice*.

3. $\boldsymbol{\delta(j, get\_orangeJuice, a)}$: 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.
La fonction de labelisation serait donc $\lambda(s) = \{s\} \forall s \in S$.

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 noeuds correspondent aux états tel que la racine est l'état initial du modèle de Kripke. Au niveau i, les fils d'un noeud sont les états successeurs au niveau i+1, c'est-à-dire les états subissant une transition à partir du noeud 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.

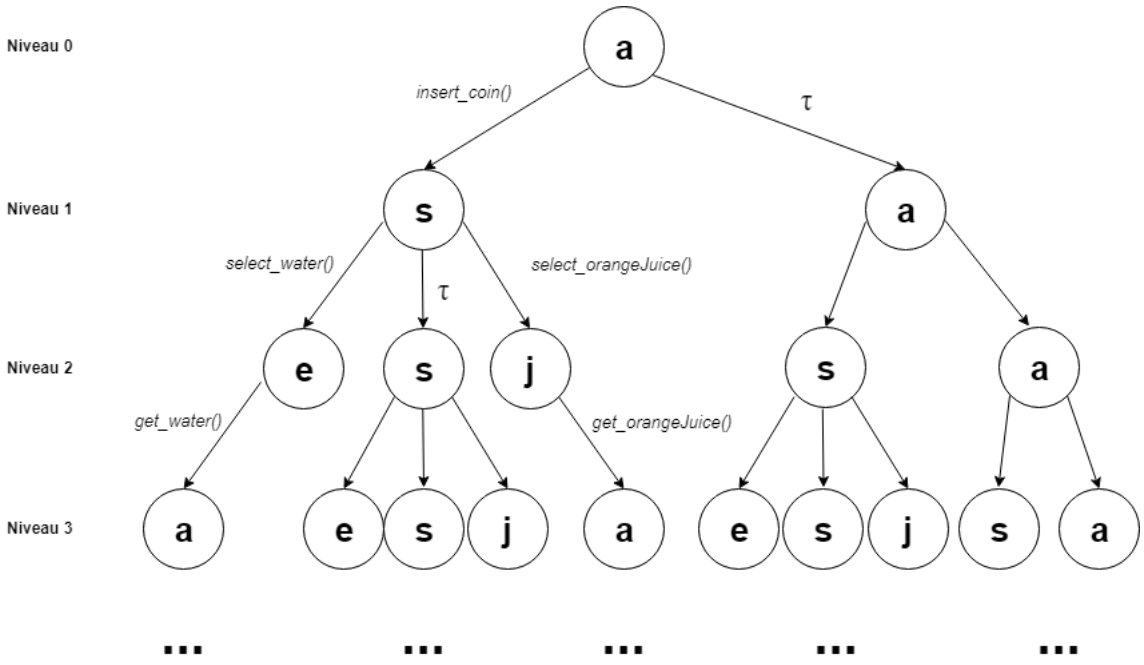


FIGURE 4: Arbre du problème d'un distributeur de boissons

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. Le niveau 0 représente la racine de l'arbre et donc l'état initial du système.

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 [7] ("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 étend la logique classique par des modalités temporelles présentés dans la Figure 5 qui référence vers différents moments dans le temps (passé ou futur).

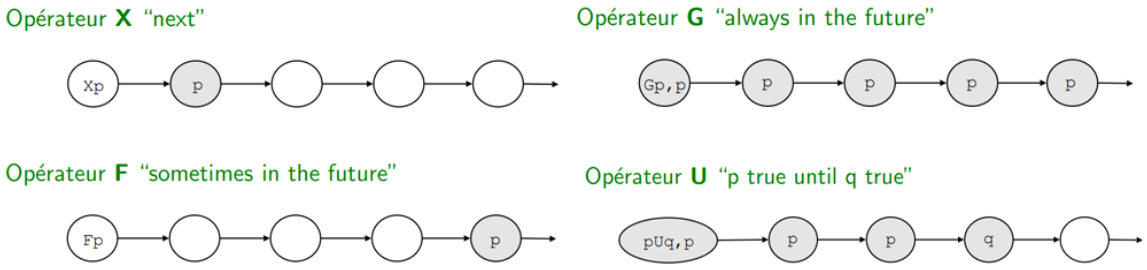


FIGURE 5: Modalités temporelles de LTL [8]

Computation Tree Logic - CTL CTL est une logique temporelle basée sur la logique propositionnelle avec une notion discrète du temps dont le future n'est pas déterminé. CTL offre notamment une logique temporelle en arbre permettant la formulation d'un ensemble conséquent de propriétés. 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.

7. 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.

8. Source : Sébastien Bardin http ://sebastien.bardin.free.fr/MC-cours2.pdf - Slides 7/44

3.3 Syntaxe de CTL

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

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 valide sur tous les chemins possibles (= inévitable).
- **E** pour "**E**xists" : la propriété est valide sur minimum un chemin (= possible).

Chacune de ses parties possède 4 types de connecteurs :
- **X** pour "ne**X**t" : la propriété doit être valide pour le prochain état du chemin.
- **F** pour "**F**inally" : la propriété est valide pour au moins un état du chemin.
- **G** pour "**G**lobally" : la propriété est valide pour tous les états du chemin.
- $\cup$ pour "**U**ntil" : la propriété doit être valide jusqu'à ce que la deuxième est "déclenchée" et devient donc valide.

La logique temporelle présentée à la Table 2 est illustrée dans la Figure 6.

9. Source : Alessandro Artale (2010) http ://www.inf.unibz.it/ artale/FM/slide4.pdf - Slide 14/37

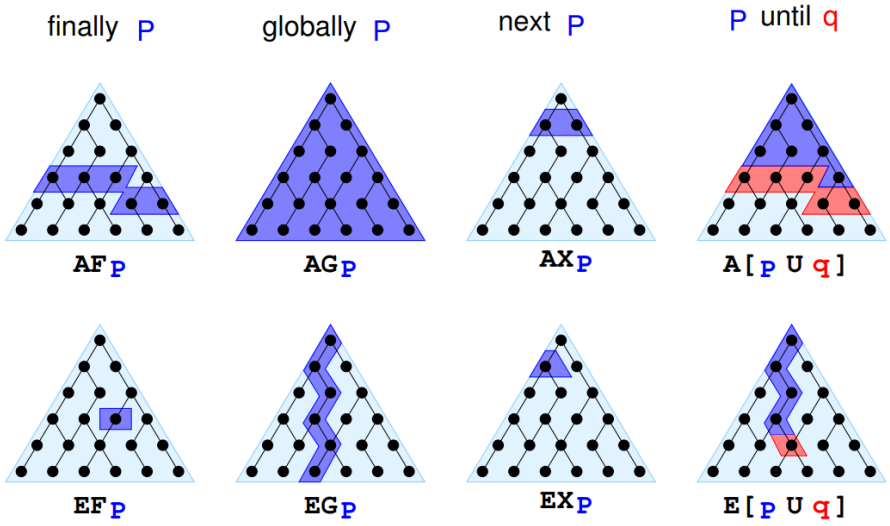


FIGURE 6: Illustration de la logique temporelle de CTL [9]

3.4 Sémantique de CTL

Les formules CTL sont interprétées par rapport aux états et aux chemins d'un système de transisition (qui suit la structure de Kripke [10]). 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 π :

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 :

$$
\begin{array}{lcl}
(\mathcal{K}, s) \vDash \neg\phi & \Leftrightarrow & \neg s \vDash \phi \\
(\mathcal{K}, s) \vDash \phi_1 \wedge \phi_2 & \Leftrightarrow & (s \vDash \phi_1) \wedge (s \vDash \phi_2) \\
(\mathcal{K}, s) \vDash \phi_1 \vee \phi_2 & \Leftrightarrow & (s \vDash \phi_1) \vee (s \vDash \phi_2) \\
(\mathcal{K}, s) \vDash \exists\phi & \Leftrightarrow & \pi \vDash \phi \text{ pour minimum un chemin } \pi \in \text{Chemins(s)} \\
(\mathcal{K}, s) \vDash \forall\phi & \Leftrightarrow & \pi \vDash \phi \text{ pour tous les chemins } \pi \in \text{Chemins(s)}
\end{array}
$$

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

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

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 : $s \vDash \phi_1 \Leftrightarrow s \vDash \phi_2$.

10. cfr. subsection 2.2

Exemples d'équivalence sémantique

— Les lois de De Morgan formulés en CTL :

$$\neg AF\phi \equiv EG\neg\phi$$
$$\neg EF\phi \equiv AG\neg\phi$$
$$\neg AX\phi \equiv EX\neg\phi$$

— Les lois d'expansions :

$$AG\phi \equiv \phi \wedge AXAG\phi$$
$$EG\phi \equiv \phi \wedge EXEG\phi$$
$$AF\phi \equiv \phi \vee AXAF\phi$$
$$EF\phi \equiv \phi \vee EXEF\phi$$
$$A(\phi_1 \cup \phi_2) \equiv \phi_2 \vee (\phi_1 \wedge AXA(\phi_1 \cup \phi_2)$$
$$E(\phi_1 \cup \phi_2) \equiv \phi_2 \vee (\phi_1 \wedge EXE(\phi_1 \cup \phi_2)$$

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

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.

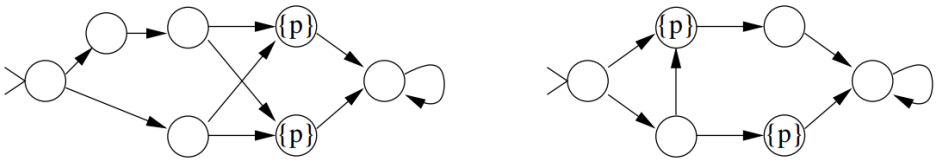
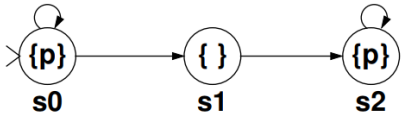


FIGURE 7: Exemple 2 - Expressivité CTL/LTL



$\mathcal{K} \models \mathbf{F}\,\mathbf{G}\,p$ but $\mathcal{K} \not\models \mathbf{AF}\,\mathbf{AG}\,p$

FIGURE 8: Exemple 3 - Expressivité CTL/LTL

Complexité théorique

- La complexité d'un problème de model checking utilisant LTL est PSPACE-complet. Plus particulièrement, sa complexité 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 d'un problème de model checking utilisant CTL est PTIME. Plus particulièrement, sa complexité 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 Pseudo-code de l'algorithme

Pseudo-code (ou code personnel) simple d'un algorithme de Model Checking CTL.

4.2 Analyses et résultats

Analyse détaillée du code de l'algorithme et présentation résultats obtenus

4.3 Complexité

Complexité du code donné ainsi que d'un model checking de CTL en général

Bibliographie

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

[2] 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*.

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

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

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

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

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

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

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

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