Définitions et notations de base

Ce chapitre détaille les définitions de base qui sont utilisées dans ce manuscrit. Le lecteur est convié à y retourner à tout instant pour clarifier notre intention autour de ces définitions. Il dispose également d'un index alphabétique en fin de manuscrit qui dispose de pointeurs vers toutes les définitions explorées entre ces pages.

1.1 Ensembles et séquences

Pour S un ensemble, le cardinal de S, noté $|S|$, est le nombre d'éléments que contient S. L'ensemble des parties de S, noté $\wp(S)$, est l'ensemble $\wp(S) = \{S' | \ S' \subseteq S\}$.

Exemple 1. *Soit* $S = \{a, b, c\}$ *un ensemble. On observe* $|S| = 3$ *et* $\wp(S) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{b, c\}, \{a, c\}, \{a, b, c\}\}$.

Une séquence est un ensemble ordonné d'éléments, que l'on note $P = (p_1, p_2, \ldots)$. Pour Λ un ensemble, l'ensemble des séquences finies dont les éléments sont dans Λ est noté Λ^*. Une séquence peut être finie ou infinie. Dans le cas d'une séquence finie, la taille de la séquence est notée $|P|$. Pour tout nombre $k > 0$, le k-ième élément de P est également noté P_k.

Soient P et P' deux séquences d'ensembles. On définit l'union de P et P', notée $P \cup P'$, comme la séquence composée de l'union des ensembles dans l'ordre des séquences originelles. Pour tout $k > 0$, la séquence est définie par

$$(P \cup P')_k = \begin{cases} P_k \cup P'_k & \text{si } P_k \text{ et } P'_k \text{ sont définis} \\ P_k & \text{si seulement } P_k \text{ est défini} \\ P'_k & \text{si seulement } P'_k \text{ est défini} \end{cases} .$$

Exemple 2. *Pour* $P = (\{1\}, \{2\}, \{3\})$ *et* $Q = (\{a\}, \{b\}, \{c\}, \{d\})$, $P \cup Q = (\{1, a\}, \{2, b\}, \{3, c\}, \{d\})$.

La séquence P^k, pour k un nombre entier, est la séquence construite comme la répétition de la séquence P un nombre k de fois.

Exemple 3. *Si* $P = (1, 2, 3)$, *alors* $P^3 = (1, 2, 3, 1, 2, 3, 1, 2, 3)$.

1.2 Fonctions

Une fonction f est dite *partielle* de A vers B s'il existe un sous-ensemble $A' \subseteq A$ tel que f est définie sur $A' \to B$. On note alors $f : A \nrightarrow B$. S'il n'existe pas de tel sous-ensemble $A' \subsetneq A$, f est dite *totale*.

Exemple 4. *Soit $f : \mathbb{R} \to \mathbb{R}$ qui, à tout nombre réel a, associe la valeur $f(a) = \frac{a}{2}$. La fonction f est correctement définie pour toute valeur de $\mathbb{R}$, elle est donc totale. Soit $g : \mathbb{R} \nrightarrow \mathbb{R}$ la fonction pour tout nombre réel a associe la valeur $g(a) = \frac{1}{a}$. Par définition, g est une fonction totale sur $\mathbb{R} \setminus \{0\} \to \mathbb{R}$, ce qui fait de g une fonction partielle.*

Pour une fonction $f : A \to B$, l'ensemble A est le *domaine* de f, noté $\mathrm{dom}(f)$. L'ensemble B est le *co-domaine* de f, noté $\mathrm{codom}(f)$.

Pour une fonction $f : A \to B$, on nomme image de f l'ensemble défini par $\mathrm{img}(f) = \{f(a) \mid a \in A\}$. L'image d'une fonction diffère de son co-domaine dès que la fonction n'est pas surjective, c'est-à-dire qu'il existe au moins une valeur $b \in B$ telle qu'il n'existe aucun $a \in A$ tel que $f(a) = b$.

Exemple 5. *Soit $f : \mathbb{N} \to \mathbb{N}$ la fonction qui à tout entier n associe $f(n) = 2n$. L'ensemble des valeurs retournées par la fonction f, $img(f)$, est l'ensemble des entiers naturels pairs.*

Soient f et g deux fonctions définies sur $f : A \to B$ et $g : B \to C$. La *composition* de g et f, notée $g \circ f$, est la fonction définie telle que $g \circ f : A \to C$ qui vérifie $g \circ f(x) = g(f(x))$.

Exemple 6. *Soit $f, g : \mathbb{R} \to \mathbb{R}$ deux fonctions définies telles que $f(x) = \frac{x}{2}$ et $g(x) = x + 1$. Par définition, $(f \circ g)(x) = \frac{x+1}{2}$, et $(g \circ f)(x) = \frac{x}{2} + 1$.*

Soit f une fonction définie sur $f : A \to B$ et A' un sous-ensemble de A. La *réduction* de f à A' est la fonction notée $f|_{A'}$ définie sur $f|_{A'} : A' \to B$ telle que $f|_{A'}(x) = f(x)$.

Pour tout ensemble A, on définit la *fonction identité*, dénotée id, comme la fonction définie sur $A \to A$ telle que $\mathsf{id}(a) = a$ pour tout $a \in A$.

Pour f une fonction définie sur $f : A \to A$ et k un nombre entier, on définit l'exponentiation de f par k, notée f^k, comme la fonction définie par les équations récursives

$$f^0 = \mathsf{id}$$
$$f^{k+1} = f \circ f^k.$$

Exemple 7. *Soit $f : \mathbb{N} \to \mathbb{N}$ la fonction qui vérifie $f(n) = n+1$. Par définition, la fonction f^k est la fonction définie sur les mêmes ensembles qui vérifie $f^k(n) = n + k$.*

1.3 Vecteurs

Un *vecteur* est une collection de valeurs indicées par un certain ensemble. Pour A l'ensemble d'indices et B l'ensemble de valeurs, l'ensemble des vecteurs qui associent

une valeur dans B aux indices dans A est noté B^A. Pour $x \in B^A$ un certain vecteur et $a \in A$, la valeur associée à l'indice a dans x est notée x_a.

Dès lors qu'il existe un ordre implicite sur l'ensemble des indices d'un vecteur, nous introduisons la notation alternative des valeurs de ce vecteur comme un mot composés des valeurs du vecteur dans l'ordre croissant de leurs indices.

Exemple 8. *Soit $A = \{a, b, c, d\}$ et $B = \{0, 1, 2\}$. Soit v un vecteur défini sur les indices A et avec valeurs dans B tel que $v_a = 0$, $v_b = 2$, $v_c = 0$ et $v_d = 1$. La valeur de ce vecteur peut être également notée* 0201.

Pour x un vecteur dans B^A et y un vecteur dans $B^{A'}$, tels que $A \cap A' = \emptyset$, on définit la concaténation de x et y, notée $x \cdot y$, comme le vecteur dans $B^{A \cup A'}$ tel que

$$(x \cdot y)_a = \begin{cases} x_a & \text{si } a \in A \\ y_a & \text{si } a \in A' \end{cases}.$$

Exemple 9. *Soient $A = \{a, b\}, A' = \{c, d\}, B = \{0, 1, 2\}$ trois ensembles, avec $v =$ 02 un vecteur défini dans B^A et $v' =$ 01 un vecteur défini dans $B^{A'}$. La concaténation de v et v' est égale à $v \cdot v' =$ 0201.*

Alternativement, nous définissons la fonction concat qui, pour tout ensemble de vecteurs, retourne la concaténation de cet ensemble :

$$\text{concat}(\{v_1, v_2, \ldots, v_n\}) = v_1 \cdot v_2 \cdot \ldots \cdot v_n.$$

Sans perte de généralité, on peut, à tout moment, considérer un vecteur $x \in B^A$ comme une fonction $f : A \to B$ qui associe une valeur à chacun de ses indices. En ces termes, nous étendons certaines notations spécifiques aux fonctions sur les vecteurs.

Pour x un vecteur dans B^A et $f : C \to A$ une fonction, on définit la *composition* de x et f, notée $x \circ f$, le vecteur définit dans B^C qui vérifie $(x \circ f)_c = x_{f(c)}$, pour tout c dans C.

Exemple 10. *Soit $v =$ 0201 un vecteur défini sur B^A, pour $B = \{0, 1, 2\}$ et $A = \{a, b, c, d\}$. Soit $f : A \to A$ la fonction définie par $f(a) = d$, $f(b) = c$, $f(c) = b$ et $f(d) = a$. On observe $v \circ f =$ 1020.*

Pour x un vecteur dans B^A et A' un sous-ensemble de A, on définit la projection de x dans $B^{A'}$, notée $x|_{A'}$, comme le vecteur qui vérifie $(x|_{A'})_a = x_a$ pour tout a dans A'.

Exemple 11. *Soit $v =$ 0201 un vecteur défini sur B^A, pour $B = \{0, 1, 2\}$ et $A = \{a, b, c, d\}$. Soit $A' = \{b, c\}$. On observe $v|_{A'} =$ 20. On note que toute projection peut être alternativement définie comme une composition. Dans notre exemple, on prend $f : \{b, c\} \to A$ la fonction qui vérifie $f(b) = b$ et $f(c) = c$. Par définition, $v \circ f = v|_{A'} =$ 20.*

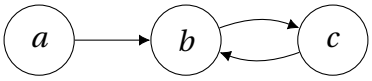

FIGURE 1.1 – Illustration du graphe G défini dans l'exemple 12.

1.4 Graphes

Un *graphe orienté* $G = (S, A)$ est défini par un ensemble de sommets S et un ensemble d'arcs $A \subseteq S \times S$. Les ensembles S et A sont respectivement notés $V(G)$ et $E(G)$. Dans le reste de ce manuscrit, le terme "graphe" sera régulièrement utilisé pour décrire un graphe orienté.

Exemple 12. *Soient $S = \{a, b, c\}$ un ensemble de sommets et $A = \{(a, b), (b, c), (c, b)\}$ un ensemble d'arcs. Le graphe orienté G qui vérifie $V(G) = S$ et $E(G) = A$ est représenté en figure 1.1.*

L'*arité entrante* d'un sommet s dans un graphe $G = (S, A)$ est définie par le nombre d'arêtes uniques (s', s) dans A. L'*arité sortante* d'un sommet s dans un graphe $G = (S, A)$ est définie par le nombre d'arêtes uniques (s, s') dans A. L'*arité* d'un sommet est la somme de son arité entrante et de son arité sortante.

Pour $G = (S, A)$ et $G' = (S', A')$ deux graphes, le graphe G' inclut le graphe G', noté $G \subseteq G'$, si et seulement si $S \subseteq S'$ et $A \subseteq A'$.

On appelle *chemin* toute séquence de nœuds $(s_1, s_2, \ldots)$ telle que $(s_i, s_{i+1}) \in A$ pour tout i. Un *cycle* est un chemin $(s_1, s_2, \ldots s_k)$ qui vérifie $s_1 = s_k$.

Un graphe G est *acyclique* s'il ne dispose d'aucun cycle. Dans un graphe acyclique, les sommets d'arité sortante 0 sont appelés les puits, et les sommets d'arité entrante 0 sont appelés les sources.

Pour $G = (S, A)$ un graphe, G est un *arbre* si et seulement si pour toute paire de sommets (s, s'), il n'existe au plus qu'un seul chemin possible pour aller de s à s' dans G. Les puits d'un arbre sont appelés les feuilles, et les sources sont appelées les racines.

Un *graphe étiqueté* est défini par un ensemble de sommets S, un ensemble d'arêtes A et un ensemble d'étiquettes E, tels que $A \subseteq S \times E \times S$. Une arête (s, e, s') signifie que l'étiquette e décore l'arête allant de s vers s'.

Un graphe $G = (S, A)$ est *fortement connexe* si et seulement si, pour toute paire s, s' de sommets distincts, il existe un chemin de s vers s' et un chemin de s' vers s. De façon similaire, une *composante fortement connexe* de G est un ensemble maximal $S' \subseteq S$ tel que pour toute paire s, s' de sommets distincts dans S', il existe un chemin de s vers s' et de s' vers s parmi les sommets S'.

On nomme *composante fortement connexe terminale* d'un graphe toute composante fortement connexe dont il est impossible de sortir. Formellement, pour $G = (S, A)$ et S' un sous-ensemble de S, S' décrit une composante fortement connexe terminale si et seulement si S' est une composante fortement connexe et si pour tout $s, s' \in S$, si $s \in S'$ et $(s, s') \in A$, alors $s' \in S'$.

1.5 Fonctions booléennes

L'ensemble des nombres *booléens* est noté $\mathbb{B}$ et défini par $\mathbb{B} = \{0,1\}$. Une *fonction booléenne* est une fonction définie de $\mathbb{B}^A \to \mathbb{B}$, pour A un certain ensemble. Pour $f : \mathbb{B}^A \to \mathbb{B}$, l'arité de f est le cardinal de A.

Une *porte booléenne* est une fonction parmi $\neg$, $\vee$ et $\wedge$. La porte $\neg$, aussi appelée "non logique", est d'arité un et est définie par $\neg 0 = 1$ et $\neg 1 = 0$. Les portes $\vee$ et $\wedge$ sont d'arité deux : la porte $\vee$, aussi appelée "ou logique", vaut 1 si et seulement si au moins une de ses deux entrées est évaluée à 1. la porte $\wedge$, aussi appelée "et logique", vaut 1 si et seulement si ses deux entrées sont évaluées à 1.

Les portes booléennes sont combinées pour former des graphes qui permettent le calcul de fonctions booléennes. Ces graphes sont acycliques, et ne possèdent qu'un seul puit. Les sources d'un tel graphe représentent les variables de la fonction booléenne, et chaque autre sommet est associé à une porte booléenne qui calcule sa valeur selon la valeur des nœuds incidents. Il est important que l'arité entrante de chaque sommet qui dispose d'une porte corresponde à l'arité de la porte : arité entrante 1 pour une porte non, et arité 2 pour une porte ou, ou et. Pour exécuter la fonction booléenne d'un tel graphe, on note la valeur de chaque variable, et on remonte le long des arcs du graphe jusqu'au puit du réseau, en exécutant les portes rencontrées sur le chemin. La valeur trouvée au puit du réseau correspond à l'évaluation de la fonction encodée par le réseau. Dans le cas général, ces objets sont appelés *circuits booléens*.

Exemple 13. *Soit $S = \{a, b, c, p1, p2\}$ un ensemble de nœuds, et les arêtes $A = \{(a, p1), (b, p1), (c, p2), (p1, p2)\}$. On associe à $p1$ la porte logique ou, et à $p2$ la porte logique et. Le circuit booléen C associé encode une fonction booléenne f d'arité 3 dont la table de vérité est*

a	b	c	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Le graphe du circuit défini dans l'exemple 13 est représenté en figure 1.2.

Lorsqu'un tel graphe possède la propriété supplémentaire d'être un arbre, on l'appelle une *formule booléenne*. La structure d'une formule booléenne permet de la représenter sous la forme d'une formule mathématique, en représentant la structure de l'arbre grâce à des parenthèses.

Exemple 14. *On considère le circuit booléen C décrit dans l'exemple 13. Le graphe du circuit C possède la propriété d'être un arbre, et est donc également une formule booléenne. Son écriture sous forme de formule est $(a \vee b) \wedge c$.*

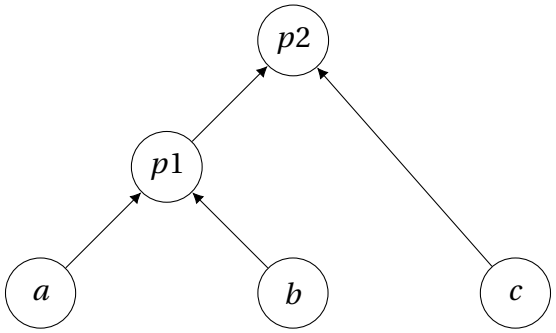

FIGURE 1.2 – Graphe du circuit booléen défini dans l'exemple 13.

1.6 Machines de Turing

Les machines de Turing sont un modèle de calcul que l'on peut représenter comme un robot qui avance et recule sur un ruban infini d'information. Le robot dispose d'une tête de lecture et écriture, d'un état et d'une table de transition. Cette table contient une entrée pour chaque état et chaque valeur de lecture possible ; pour chacune de ces paires, cette table retourne une valeur à écrire, un déplacement optionnel et un nouvel état pour la machine de Turing. Mettre à jour la machine repose sur la lecture de cette table et l'application de son résultat. Ce calcul peut ne jamais s'arrêter ; cependant on définit souvent des états spéciaux, dit finaux, qui une fois établis vérouillent la machine dans une position et préviennent toute autre modification. C'est une convention pour signifier la fin du calcul d'une machine de Turing.

On fait la distinction entre les machines de Turing déterministes et les machines de Turing non déterministes sur le critère qu'une machine de Turing déterminisite dispose toujours d'un et un seul choix en toute circonstance. Une machine de Turing non déterminisite peut au contraire disposer d'une table de transition qui définit au moins deux possibles résultats pour les mêmes circonstances.

1.6.1 Cas déterministe

Formellement, une machine de Turing déterministe se définit comme un quintuplet $(S, \Lambda, s_0, \delta, F)$, où S est un ensemble fini d'états, Λ est l'alphabet des symboles du ruban, s_0 est un état de S dit initial, $\delta : S \times \Lambda \to S \times \Lambda \times \{-1, 0, +1\}$ est la fonction qui encode la table de transition, et F décrit le sous-ensemble des états de la machine dits finaux. Le ruban de la machine de Turing est représenté par un vecteur bi-infini $R \in \Lambda^{\mathbb{Z}}$.

Pour mettre à jour une telle machine $T = (S, \Lambda, s_0, \delta, F)$ depuis un état s sur le ruban $R \in \Lambda^{\mathbb{Z}}$, et telle que la position de la tête soit actuellement un certain $z \in \mathbb{Z}$, nous calculons $(s', y, p) = \delta(s, R_z)$. Il résulte de cette mise à jour que la machine T est maintenant à l'état s' et sa tête à la position $z + p$, et le nouveau ruban R' est défini en tout point égal à R excepté $R'_z = y$. Dans le cas particulier où l'état actuel de la machine est dans F, la machine n'est pas mise à jour.

Exemple 15. *Soit* $T = (\{a,b\}, \mathbb{B}, a, \delta, \emptyset)$ *la machine de Turing telle que*

$$\begin{aligned}\delta(a,0) &= (b,1,+1)\\ \delta(a,1) &= (a,0,-1)\\ \delta(b,0) &= (b,1,-1)\\ \delta(b,1) &= (a,0,+1).\end{aligned}$$

On observe l'exécution périodique de T suivante :

état a	$\ldots 000\check{0}000\ldots$
état b	$\ldots 0001\check{0}00\ldots$
état b	$\ldots 000\check{1}100\ldots$
état a	$\ldots 0000\check{1}00\ldots$
état a	$\ldots 000\check{0}000\ldots$

Chaque ligne correspond à une étape de la mise à jour de la machine T. Dans cette représentation, l'état de la machine de Turing est indiqué sur la gauche, et la configuration d'un segment du ruban bi-infini est indiquée sur la droite. Les notations $\check{0}$ *et* $\check{1}$ *précisent la position de la tête de lecture.*

1.6.2 Cas non déterminisite

Une machine de Turing non déterministe est un quintuplet $(S, \Lambda, s_0, \Delta, F)$, où S, Λ, s_0 et F sont définis de la même façon que pour une machine déterministe. Ici, Δ est défini comme un sous-ensemble de $S \times \Lambda \times S \times \Lambda \times \{-1, 0, +1\}$, tel que pour toute paire (s, x), il existe au moins un triplet (s', y, p) tel que $(s, x, s', y, p) \in \Delta$.

Ainsi, pour tout quintuplet $(s, x, s', y, p) \in \Delta$, la machine a la possibilité de passer de l'état s à l'état s' lorsqu'elle lit la valeur x, en entraînant l'écriture de l'état y et le déplacement p.

Mettre à jour une machine de Turing non déterministe à partir d'une valeur de lecture x et d'un état s repose sur la sélection d'un quintuplet $(s, x, s', y, p) \in \Delta$. Lorsque plusieurs tels quintuplets existent, le calcul est non déterministe ; plutôt que de disposer d'une exécution possible, la machine possède plusieurs possibilités. On considèrera en général chaque possibilité comme dessinant un arbre de configurations dont la racine est la configuration initiale de la machine, telle que chaque nœud de l'arbre dispose d'autant de fils qu'il y a de possibles mises à jour de la configuration qui correspond au nœud. Ainsi, les feuilles de l'arbre correspondent aux différentes façons dont la machine peut terminer son calcul à partir de cette configuration initiale.

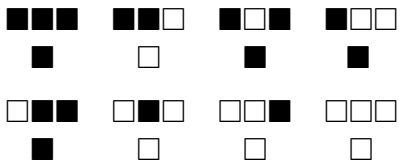

FIGURE 1.3 – Table de vérité de la fonction booléenne qui encode l'automate cellulaire élémentaire 184. Ce nombre est encodé par le nombre binaire obtenu par la concaténation de la réponse de la fonction à chaque configuration binaire de longueur 3, dans l'ordre décroissant. Dans notre exemple, $184_{10} = 10111000_2$.

1.7 Automates cellulaires élémentaires

Les automates cellulaires disposent d'un ensemble régulier de cellules, comme une grille ou une ligne, et chacune de ces cellules contient une valeur. On met à jour un automate cellulaire en appliquant à chaque cellule une fonction locale qui prend en entrée la valeur de la cellule et de ses voisins. Les automates cellulaires élémentaires sont un ensemble de 256 automates cellulaires parmi les plus simples qu'il est possible de définir : ils opèrent sur des configurations en forme de ligne et sur un alphabet binaire, et considèrent une définition de voisinage très simple : les voisins d'une cellule sont elle-même, sa voisine de gauche, et sa voisine de droite.

Ainsi, la fonction locale d'une cellule d'un automate cellulaire élémentaire est une fonction qui prend en entrée trois valeurs booléennes et retourne une valeur booléenne. Il n'existe que $2^{2^3} = 256$ de ces fonctions, ce qui invite la convention de nommer un automate cellulaire élémentaire par le numéro qui encode la fonction locale qui le définit (WOLFRAM 1984).

Exemple 16. *On considère l'automate cellulaire élémentaire* 184, *dont la table de vérité de la fonction locale est représentée en figure 1.3. Soit x une configuration représentée par un vecteur* 01100010111010. *On suppose que cette configuration forme une boucle de façon à ce que le voisin de gauche de la cellule plus à gauche soit la cellule la plus à droite, et vice versa. La mise à jour de la règle* 184 *sur cette configuration donne la configuration* 01010001110101.

Malgré la grande simplicité de définition des automates cellulaires élémentaires, l'automate cellulaire élémentaire 110, aussi connu sous le nom règle 110, est connu pour être capable de calcul universel. Ce calcul prend place grâce à l'interaction de planeurs dans l'évolution de la configuration qui permettent un transfert d'information. La table de vérité de cet automate est illustrée en figure 1.4.

Le calcul d'un automate cellulaire élémentaire repose sur la mise à jour successive d'une configuration initiale. La séquence des configurations ainsi obtenues s'appelle un diagrame espace-temps. L'illustration d'une exécution de la règle 110 est faite en figure 1.5.

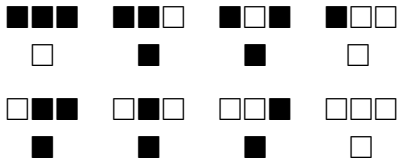

FIGURE 1.4 – Table de vérité de la fonction booléenne qui encode l'automate cellulaire élémentaire 110.

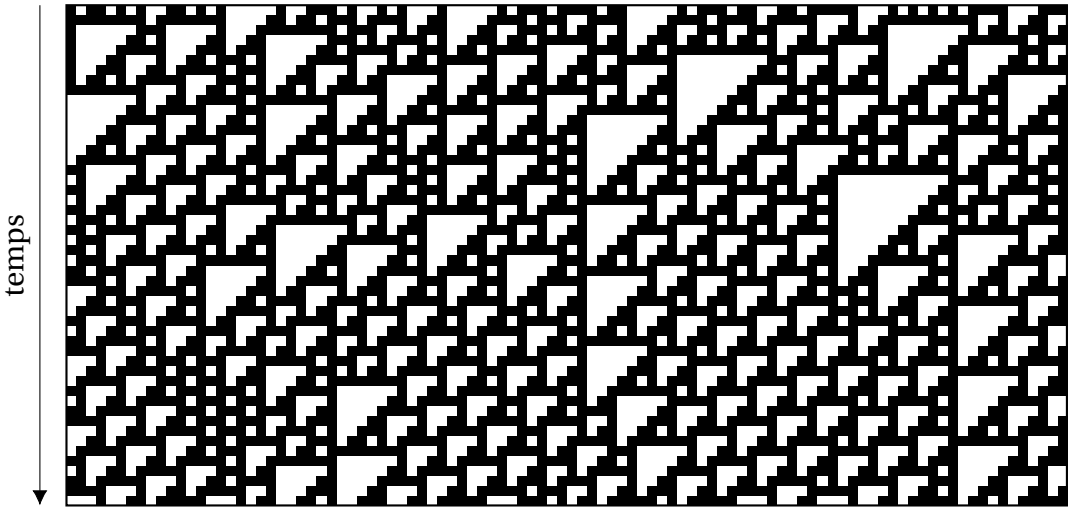

FIGURE 1.5 – Diagramme espace-temps d'une exécution de la règle 110. L'exécution est représentée sous la forme d'une grille, dans laquelle chaque cellule noire correspond à la valeur 1 et chaque cellule blanche correspond à la valeur 0. Chaque ligne correspond à une configuration. La configuration initiale de l'automate correspond à la première ligne de la grille. La mise à jour de cette configuration est représentée juste en dessous : la lecture d'une telle exécution se fait de haut en bas.

Une configuration d'un automate cellulaire peut être finie ou infinie ; dans le contexte de ce manuscrit, l'information utile de chaque configuration sera limitée. Par conséquent, on considèrera les configurations des automates cellulaires soit comme des configurations finies et circulaires, c'est-à-dire dont la cellule la plus à gauche est définie comme la voisine de droite de la cellule la plus à droite, et vice versa ; soit comme infinie et redondante, c'est-à-dire comme la répétition infinie d'un segment donné. Dans les deux cas, la dynamique observée est moralement la même, et nous illustrerons les deux cas de façon équivalente.

1.8 Complexité

La théorie de la complexité procure un ensemble d'outils qui permet la mesure de la difficulté d'une question ou de l'efficacité d'un algorithme. Elle est utilisée dans pratiquement tous les domaines de l'informatique théorique. Nous définissons ici les bases qui permettent la lecture sereine de ce manuscrit. Cette théorie repose sur la définition et la catégorisation de problèmes.

1.8.1 Problèmes de décision

Un problème de décision définit un ensemble infini d'instances, chacune d'entre elles étant une instance positive ou négative. Une instance peut être vue comme une question ; le signe de l'instance représente la réponse à cette question. Un problème de décision se définit de la façon suivante :

▶ Problème d'évaluation d'un circuit booléen
Entrée: Un circuit C, encodé comme un graphe.
Question: L'évaluation de la racine du réseau C donne-t-elle 1 ?

La définition d'un problème de décision repose ainsi sur trois éléments : un nom, une famille d'instances (dans cet exemple, nous définissons une instance pour chaque circuit C), et une question pour chacune de ces instances. Résoudre ce problème repose sur la capacité à résoudre cette question pour toute instance.

1.8.2 Classes de complexité

Pour classifier l'infinité des problèmes de décision possibles, la théorie de la complexité propose un ensemble de classes, qui déterminent le type de machine et le temps nécessaire à résoudre l'ensemble des questions posées par le problème. Parmi les plus célèbres, la classe P contient tous les problèmes qui peuvent être résolus avec une machine de Turing déterministe en temps polynomial dans le pire des cas. Cela signifie qu'il existe un polynôme g et une machine de Turing déterministe T tels que lorsque T est initiée sur le nombre k qui correspond à une instance du problème, cette machine répond correctement par "oui" ou "non" par le biais d'états finaux spécialisés après au plus $g(|k|)$ étapes de temps, où $|k|$ est le nombre de bits nécessaires pour représenter k.

Parmi d'autres classes célèbres, nous pouvons citer NP, définie comme la classe de problèmes dont la solution peut être vérifiée en temps polynomial avec une machine de Turing déterministe dans le pire des cas, c'est-à-dire que pour toute instance positive du problème il existe un p-prédicat de taille polynomiale qui permet de vérifier que l'instance est positive. Définie comme la classe duale de NP, coNP est la classe qui contient les problèmes dont les instances peuvent être vérifiées négatives en temps polynomial par une machine de Turing déterministe dans le pire des cas, c'est-à-dire que, pour toute instance négative, il existe un p-prédicat de taille polynomiale

qui permet de vérifier que l'instance est négative. Dans le cas de la classe NP, le p-prédicat sert généralement à donner la solution ; dans le cas de la classe coNP, ce p-prédicat représente généralement une sorte de contre-exemple à l'instance.

La classe FPT (pour *Fixed Parameter Tracktable*, DOWNEY, FELLOWS et STEGE 1999) contient les problèmes de la forme (x, k), pour k un paramètre, qui peuvent être résolus par une machine Turing déterministe en au plus $f(k)g(|x|)$ étapes de temps, où g est un polynôme et f une fonction arbitraire. Nous utilisons cette classe pour illustrer quels aspects d'une instance d'un problème donné contribuent le plus fortement à sa complexité.

On peut également citer la classe EXP qui contient les problèmes qui peuvent être résolus avec une machine de Turing qui dispose d'un temps exponentiel en la taille de l'entrée dans le pire des cas.

1.8.3 Oracles

Une autre façon de définir des classes de complexité est par l'utilisation d'oracles. La notation coNP^{NP} décrit la classe de problèmes qui sont dans coNP lorsque l'on dispose d'un oracle NP. Un oracle est une machine qui permet la résolution immédiate d'un problème de décision. Un problème est dans NP^{NP}, par exemple, si ce problème peut être résolu par une machine de Turing non déterministe qui dispose de la capacité d'appeller un nombre polynomial d'oracles NP. Sous l'hypothèse que la classe P est différente de la classe NP, cette notation d'oracles, utilisée sur les classes P, NP et coNP, permet la définition d'une famille infinie de classes de complexité, appelée la hiérarchie polynomiale.

1.8.4 Réduction

La notion de réduction permet de montrer qu'un problème peut se réduire en tout point à un autre. En particulier, la réduction polynomiale se définit de la façon suivante : une réduction polynomiale depuis un problème A vers un problème B est une fonction f définie depuis les instances de A vers les instances de B calculable par une machine de Turing en temps polynomial, de façon à ce que x est une instance positive (resp. négative) de A, si et seulement si $f(x)$ est une instance positive (resp. négative) de B. On note cette réduction $A \leq_p B$. Prouver une telle réduction polynomiale permet de montrer que le problème A est au moins aussi facile que le problème B, puisque l'on peut maintenant résoudre A en résolvant le problème B avec le prix ajouté d'une fonction calculable en temps polynomial. Cette réduction montre également que le problème B est au moins aussi difficile que le problème A ; en effet, résoudre toutes les instances de B demande en particulier de savoir résoudre chaque instance de A. La réduction polynomiale est le principal outil qui permet de prouver l'appartenance ou la non appartenance d'un problème à la classe NP et à toutes les classes plus larges que NP dans la hiérarchie polynomiale. Si tout les problèmes d'une classe donnée peuvent être réduits à un problème particulier, alors ce problème est un problème

difficile de cette classe. Si un problème est à la fois un problème difficile d'une classe et un problème de cette classe, alors ce problème est dit complet pour cette classe. Les problèmes complets sont les représentants des problèmes les plus difficiles que l'on peut rencontrer dans cette classe.

Pour les classes inférieures à NP dans la hiérarchie polynomiale, on utilise des notions plus fortes de réduction, comme la réduction en espace logarithmique. Le problème d'évaluation d'un circuit booléen est un exemple typique de problème P-complet : c'est un problème qui est résolvable en temps polynomial par une machine de Turing déterministe et qui permet la résolution de tout problème dans P par le biais de la réduction en espace logarithmique. Une réduction en espace logarithmique se définit comme une réduction qui est calculable sur une machine de Turing à trois rubans : un ruban d'entrée (accessible en lecture seulement) qui contient l'entrée, un ruban de travail (accessible en lecture et en écriture) qui est de taille logarithmique en la taille de l'entrée et permet de faire des calculs, et un ruban de sortie (accessible en écriture seulement) qui permet de rédiger la sortie.

1.8.5 Problème fonctionnel

Un problème fonctionnel est un problème qui, plutôt que d'associer une solution binaire à ses instances, associe une solution complexe. Un tel problème peut être vu comme une fonction de l'ensemble de ses instances à l'ensemble de ses sorties, à l'exception qu'une telle instance peut être négative et ne pas avoir de solution définie. Un tel problème se définit comme suit :

▶ Problème de factorisation d'un entier
Entrée: Un entier naturel n.
Sortie: La décomposition de n en facteurs premiers.

Les problèmes fonctionnels disposent de leur propres classes, qui sont la traduction naturelle des classes de décision. Les classes FP et FNP sont les classes dont les problèmes fonctionnels sont résolus par une machine de Turing respectivement déterministe et non déterministe en temps polynomial. La classe FcoNP regroupe les problèmes fonctionnels dont les instances sont prouvées ne pas avoir de solution en temps polynomial par une machine de Turing non déterministe.

La notion de réduction polynomiale est difficile à traduire dans le contexte des problèmes fonctionnels. Cela vient du fait que certains problèmes que nous voudrions définir comme FP-difficiles ne disposent d'aucune instance négative. Cette difficulté technique empêche la définition d'un ensemble de problèmes complets pour une classe de problèmes fonctionnels. Dans ce manuscrit, nous prouvons la difficulté d'un problème fonctionnel par réduction vers un problème de décision difficile.

Pour $f : A \to B$ une fonction, le problème fonctionnel qui correspond à f est le problème qui admet ses instances dans A, et qui pour toute instance $a \in A$ attend la réponse $f(a)$. Si ce problème est dans FP, alors la fonction f est dite calculable en temps polynomial. Dans le cas où ce problème peut être résolu par une machine de

Turing limitée à un espace de travail logarithmique en la taille de l'entrée, f est dite calculable en espace logarithmique.

1.8.6 Problème fonctionnel avec promesse

Les problèmes de complexité avec promesse permettent une sélection des instances du problème considéré, dont le coût calculatoire ne repose pas sur la résolution de l'instance. Une exploration plus approfondie de cette notion peut être trouvée dans (GOLDREICH 2006). Notre notation de ces problèmes est la suivante :

▶ Coloriage d'un graphe planaire

Entrée: Un graphe G non orienté, et k un entier naturel.

Promesse: Le graphe G est planaire.

Sortie: Une fonction f qui, à tout sommet de G, associe un entier naturel inférieur à k telle que pour deux sommets s, s', $(s, s') \in A \iff f(s) \neq f(s')$.

Pour résoudre l'ensemble des instances de ce problème, nous n'avons besoin que de considérer les instances pour lesquelles la promesse est vérifiée. Ainsi, tout algorithme résolvant ce problème a la possibilité d'adopter un comportement arbitraire sur les instances qui ne respectent pas la promesse. Il n'a donc jamais besoin de vérifier que l'entrée concernée encode un graphe planaire valide, puisque cela est garanti par la promesse.

2 Simulation : contexte et généralisations

La simulation est un concept singulier dans le calcul naturel. Bien que son intuition, la capacité de la reproduction du calcul, semble universelle, les définitions dont nous disposons sont toujours propres aux modèles qui les concernent. Ce schisme vient certainement du fait que nous n'avons jamais de définition unifiée de ce qu'est un calcul ; nous sommes réduits dans la pratique à ne définir que des exemples locaux de son apparition. En est-il de même avec la simulation, avec qui le calcul semble hautement apparenté ?

Nous ne sommes pas les premiers à nous poser la question (Boas 2014). Il est désirable d'obtenir une telle définition générale si elle existe, simplement car l'unification d'une partie des méthodes d'un large domaine de recherche est à la clé. Devant une question aussi ambitieuse, nous adoptons la démarche de décrire les intuitions qui semblent être clés dans les définitions de simulation dont nous disposons dans certains modèles de calcul comme les automates cellulaires, puis de poursuivre la généralisation de ces intuitions vers, nous l'espérons, une définition applicable dans un cadre plus général.

2.1 Une définition informelle

La notion de simulation est fortement liée à la notion de l'universalité, dont il existe des variantes. La plus classique d'entre elles, l'universalité Turing, prend racine dans la thèse de Church-Turing, des deux mathématiciens qui, dans les années 1930, déclarent indépendamment avoir défini des modèles de calcul qui ont la faculté de reproduire tout calcul qu'un humain peut opérer si on lui procure une quantité infinie de feuilles, de crayon et de temps. Bien que cette thèse n'ait jamais été prouvée (car trop informelle), elle est communément admise et n'a jamais été contredite depuis 90 ans. Les machines de Turing sont parmi les exemples les plus connus de modèles dit Turing universels.

Une fois admise l'universalité Turing d'un modèle, il est possible d'étendre cette universalité à des modèles parfois de natures complètement différentes. Pour ce faire, on utilise la notion de simulation. La simulation est une relation entre modèles de calcul qui s'applique entre un modèle simulateur et un modèle simulé. En pratique, tout calcul opérable par la machine simulée peut être reconstruit comme un calcul opérable par la machine simulatrice, de façon à ce que le calcul de la machine simu-

latrice permette de prédire tout aspect du calcul simulé. Une telle relation se devra d'être réflexive et transitive : une machine doit être capable de reproduire son propre calcul, ce qui est trivial ; si une machine en simule une seconde qui en simule une troisième, alors elle simule la troisième. Ces deux propriétés font de la simulation un pré-ordre.

Une autre variante de l'universalité est nommée universalité intrinsèque, et ne fait sens que lorsque l'on examine la capacité d'un modèle à simuler tout autre modèle inclus dans la même famille. Cela est généralement dit d'un modèle spécifiquement capable d'une grande souplesse calculatoire. C'est le cas de certaines machines de Turing universelles, qui sont conçues pour accepter et reproduire le code décrivant n'importe quelle autre machine : une telle machine est intrinsèquement universelle dans la famille des machines de Turing. L'universalité intrinsèque ne découle cependant pas toujours de l'universalité Turing. Pour être intrinsèquement universel, un modèle devra souvent opérer son calcul en exploitant les propriétés intrinsèques de la famille de modèles utilisés. Les conditions spécifiques de ces propriétés changent avec la famille considérée, mais en général impliquent une simulation plus efficace.

Afin de se construire une intuition du fonctionnement de la simulation, examinons des définitions tirées de la littérature.

2.1.1 Simulation d'automates cellulaires

Les automates cellulaires sont une famille de modèles centrale dans le calcul naturel. Une présentation sommaire d'une fraction emblématique de ces modèles, les automates cellulaires élémentaires, est faite en section 1.7.

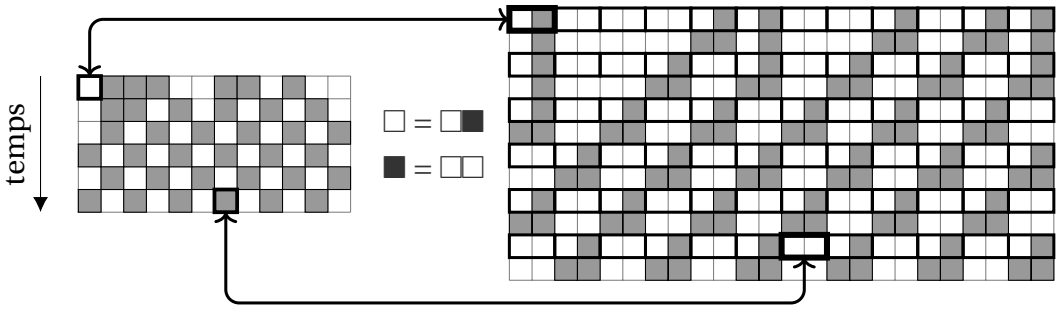

FIGURE 2.1 – Simulation d'une exécution de la règle 184 (gauche) par la règle 134 (droite). L'exécution de la règle 184 est encodée dans l'exécution de la règle 134. En ne considérant qu'une configuration sur deux, chaque paire de cellule (encadrées en noir sur la droite) encode une cellule de l'automate 184 de la façon suivante : l'état □□ encode la valeur ■ et l'état □■ encode la valeur □.

Lorsque l'on opère une simulation d'un automate cellulaire par un autre automate cellulaire, la méthode employée repose généralement sur l'encodage des valeurs des

cellules de l'automate simulé par des regroupements de cellules dans l'automate simulateur. La figure 2.1 détaille l'exemple de la simulation de la règle 184 par la règle 134 : ici, chaque automate de la règle 184 est remplacé par 2 cellules dans l'automate 134. On note également qu'une mise à jour supplémentaire est nécessaire entre chaque étape de la simulation. En d'autres termes, cette simulation opère par la mise à l'échelle dans l'espace et le temps de la règle 134. En choisissant une configuration initiale donnée par le bon encodage, la règle 134 se trouve reproduire "tous les calculs" de la règle 184.

Beaucoup d'autres simulations ont été développées dans la littérature des automates cellulaires, ce qui a permis l'identification de nombreux modèles intrinsèquement universels ou Turing universels (OLLINGER 2012 ; BECKER, MALDONADO, OLLINGER et al. 2018 ; COOK 2004 ; MARTIEL 2015). Un grand travail a été placé dans la généralisation des notions de simulation et d'universalité chez les automates cellulaires (OLLINGER 2002 ; THEYSSIER 2005), avec par exemple la généralisation de l'ensemble des méthodes reposant sur le groupage des cellules dans le simulateur par la notion de Bulking (DELORME, MAZOYER, OLLINGER et al. 2011a ; DELORME, MAZOYER, OLLINGER et al. 2011b), qui emploie des outils algébriques pour généraliser le plus fondamentalement possible la notion de mise à l'échelle de l'espace et du temps. Cette notion très générale de regroupement est ensuite utilisée pour définir plusieurs notions de simulation entre automates cellulaires.

La notion de Bulking permet une définition très générale de la simulation intrinsèque entre automates cellulaires ; c'est-à-dire toute simulation qui exploite les propriétés de localité et d'uniformité propres à ces modèles : leur nature uniforme et régulière invite à décrire une simulation efficace d'un automate par un autre comme une remise à l'échelle.

Cette définition est considérée satisfaisante pour englober les notions de simulation intrinsèque qui concernent les automates cellulaires. Qu'en est-il dans un cadre plus général ? Il existe par exemple des généralisations des automates cellulaires qui brisent la régularité de leur réseau ou l'uniformité de leur mise à jour dans l'espace ou le temps, comme les réseaux d'automates. Dans ces modèles, qui ne sont pas uniformes dans l'espace et le temps, le Bulking n'est pas applicable. De fait, l'intuition à la base de la notion de la simulation intrinsèque change radicalement dès que l'on change la famille de modèles que l'on considère.

2.1.2 Simulation de réseaux d'automates

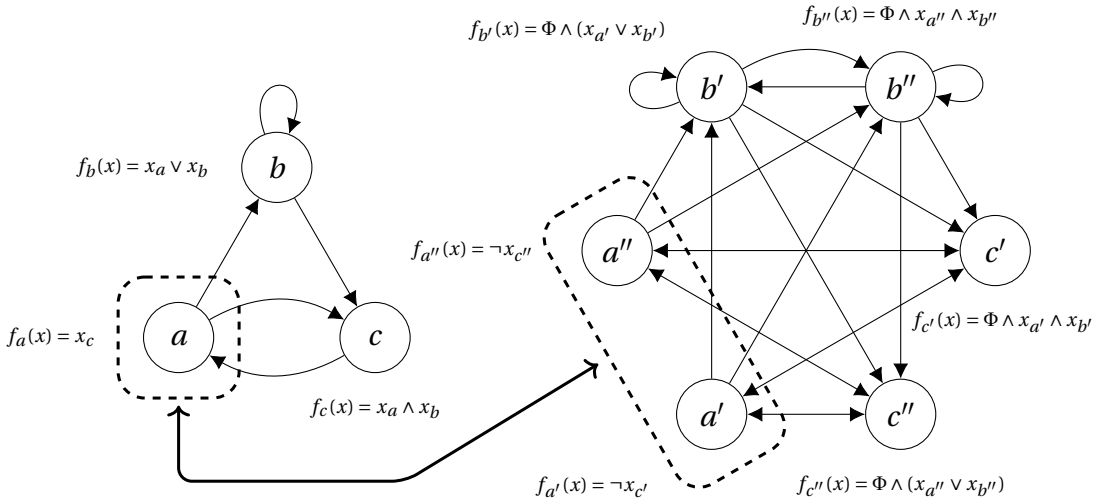

FIGURE 2.2 – Illustration de la simulation d'un réseau d'automates booléens de taille 3 (gauche) par un réseau d'automates booléens de taille 6 localement monotone (droite), dans lequel $\Phi = (x_{a'} \vee x_{a''}) \wedge (x_{b'} \vee x_{b''})$. Cette simulation est une application du corollaire 3 donné en section 5.3.3. L'état de chaque automate à gauche est encodé par les états de deux automates à droite ; par exemple, l'état de a est encodé par les états de a' et a''.

Les réseaux d'automates sont une variation des automates cellulaires qui ne sont pas définis sur un espace régulier. Si un automate cellulaire est composé de cellules, un réseau d'automates est composé d'éléments appelés automates. Les automates d'un réseau d'automates ne sont pas nécessairement mis à jour de façon homogène, ni même de façon synchrone. Un développement approfondi de ces modèles sera proposé au chapitre 4.

Puisque ces modèles ne garantissent pas la régularité de leur réseau ni celle de leur mise à jour, la réalisation d'une remise à l'échelle ne sera pas suffisante. À vrai dire, leur définition fait de chaque réseau d'automates un objet au graphe d'interaction particulier. Cependant, la plupart des simulations entre réseaux d'automates suit une logique similaire aux simulations entre automates cellulaires : chaque automate du réseau simulé sera représenté par un ou plusieurs automates dans le réseau simulateur. L'état de ce groupe d'automates encode alors l'état de l'automate simulé. Une illustration d'une telle simulation est représentée en figure 2.7, dans laquelle un réseau de taille 3 est simulé par un réseau de taille 6, tel que le comportement de chaque automate du réseau simulé est reproduit par une paire d'automates dans le simulateur. Cette méthodologie semble très similaire à celle développée dans l'exemple de la figure 2.1 : dans les deux cas, chaque automate est simulé localement par une petite structure. Une généralisation limitée de cette notion a été proposée par (PERROT, PERROTIN et

SENÉ 2018) et est également exprimée en section 5.3.3.

La simulation de tout réseau d'automates booléens par un réseau d'automates booléens localement monotone (PERROT, PERROTIN et SENÉ 2018) est un résultat qui permet l'établissement de l'universalité intrinsèque de ces derniers. La simulation de réseaux exécutés en parallèle par des réseaux exécutés par blocs-séquentiels (BRIDOUX, GUILLON, PERROT et al. 2017) en est un autre exemple. Une étude détaillée de la simulation dans les réseaux d'automates est faite par les travaux (BRIDOUX, CASTILLO-RAMIREZ et GADOULEAU 2020 ; BRIDOUX 2019 ; BRIDOUX, GADOULEAU et THEYSSIER 2020). Les notions liées aux réseaux d'automates sont développées en détail dans le chapitre 4.

2.2 Simulation par forme

La simulation intrinsèque semble en pratique souvent reposer sur une remise à l'échelle des modèles considérés : un élément simple est alors simulé par un ensemble d'éléments qui, mis ensemble, forment une structure au comportement recherché. Guidés par cette observation, nous développons une façon de percevoir les modèles de calcul qui permet la définition d'une notion de simulation, nommée simulation par forme. Son approche est très simple : les modèles de calcul sont perçus comme des collections de diagrammes espace-temps. Chacun de ces diagrammes assigne un état à chaque élément de l'espace et du temps. La simulation par forme repose sur l'encodage d'un tel diagramme par l'assemblage d'un diagramme simulateur obtenu par le remplacement de chaque élément atomique du diagramme simulé par une "forme".

2.2.1 Modèles de forme et sous-espaces fonctionnels

Comme décrit plus haut, les modèles de calcul sur lesquels la simulation par forme se définit sont considérés comme des collections d'exécutions. Dans cette section, l'ensemble X sera en général associé avec l'ensemble des positions dans l'espace et le temps d'un modèle de calcul. Dans beaucoup d'exemples, X est le produit cartésien d'un ensemble spatial P et d'un ensemble temporel T.

Exemple 17. *Considérons un automate cellulaire élémentaire. L'ensemble d'espace-temps X qui correspond à ce modèle est le produit cartésien $X = P \times T$, où $P = \mathbb{Z}$ correspond à l'espace d'une configuration infinie, et $T = \mathbb{N}$ correspond au temps, qui est infini avec une configuration initiale* 0. *Chaque paire $(p, t) \in X$ correspond au moment t de la cellule à la position p.*

L'ensemble S sera quant à lui associé à l'ensemble des états que chaque position dans l'espace-temps peut admettre.

Exemple 18. *Considérons un modèle d'automate cellulaire élémentaire. Ce modèle est booléen, et son espace d'états est $S = \mathbb{B}$.*