

Programmation synchrone

Adrien Guatto

2020–2021

Ces notes de cours proposent une introduction à la programmation des systèmes réactifs par le biais des langages de programmation dits *synchrone*. Elles correspondent à un enseignement délivré durant l’année universitaire 2020–2021 à l’Université de Paris, en Master 2 Informatique ainsi qu’à l’École d’Ingénieur Denis Diderot.

Version du 30 septembre 2020. Je suis preneur de toute coquille, erreur ou remarque par courriel à l’adresse adrien@guatto.org.

Ne pas redistribuer.

Table des matières

<i>Introduction</i>	1
<i>Programmes et systèmes réactifs</i>	1
<i>Langages synchrones</i>	4
<i>Le reste du cours</i>	5
<i>La programmation synchrone à flots de données</i>	6
<i>Premiers programmes</i>	8

Introduction

L’apprentissage de la programmation se structure traditionnellement le long de deux axes :

1. la pratique de la programmation dans des langages variés, par exemple Java, C ou OCaml ;
2. l’étude de concepts et techniques algorithmiques indépendants du langage de programmation, par exemple l’algorithmique des tris, ou encore celle des graphes.

Le but du présent cours est de prolonger ces deux axes dans une direction qui devrait être nouvelle pour vous : celle des programmes *réactifs*, par opposition aux programmes *transformationnels*¹.

1. La distinction entre ces deux classes de programmes est traditionnellement attribuée à Zohar Manna et Amir Pnueli [5].

Programmes et systèmes réactifs

Un programme transformationnel lit une entrée, la traite, puis produit un résultat complet en temps fini. L’utilitaire `sort` d’UNIX, qui trie une liste de lignes par ordre alphabétique, constitue un exemple très simple de programme transformationnel. Un exemple plus sophistiqué est fourni par n’importe quel compilateur capable de traduire un fichier source en un fichier en langage cible² réutilisable³. La plupart des programmes que vous avez écrits jusqu’ici sont transformationnels.

2. Par exemple, le langage machine compris par votre processeur pour gcc, ou le code-octet de la machine virtuelle Java pour javac.

3. Le cas des compilateurs à la volée (*just-in-time*), qu’on trouve notamment dans les navigateurs web, est plus complexe et ne rentre pas facilement dans la dichotomie transformationnel *vs.* réactif.

Par opposition, un **programme réactif est en interaction continue avec un environnement extérieur** qu'il va chercher à contrôler, surveiller ou réguler. Cet environnement extérieur est généralement un environnement physique, que notre programme observe par le biais de capteurs et influence par le biais d'actuateurs — voir fig. 1. Pensez au pilote automatique d'un avion (*fly-by-wire* en anglais), au *firmware* du modem 4G de votre téléphone, ou plus modestement au contrôleur de votre four à micro-ondes.

Question 1. Pouvez-vous citer d'autres exemples de programmes transformationnels ? De programmes réactifs ? De programmes qui ne semblent pas appartenir nettement à un des deux côtés de cette classification ?

Un programme réactif ne peut généralement pas être considéré en isolation de son environnement extérieur. Par exemple, le pilote automatique d'un avion fait des hypothèses sur l'environnement aérien (altitude, force du vent, etc.) ainsi que sur l'avion lui-même (poids, portance, etc.). Pour cette raison, on parlera généralement de *système réactif* pour englober sous un même terme le logiciel et son environnement, en insistant sur leur interdépendance⁴.

En plus de cette définition générale, beaucoup de systèmes réactifs partagent un petit nombre de caractéristiques essentielles, que nous allons maintenant aborder.

Criticité. Certains programmes réactifs contrôlent des dispositifs physiques où l'erreur peut avoir des conséquences catastrophiques sur la vie humaine, ou bien un coût financier démesuré. Citons deux erreurs restées tristement célèbres.

- Le 4 juin 1996, le premier vol d'Ariane 5 aboutit à la destruction du lanceur 37 secondes après son décollage (fig. 2). Une partie du logiciel avait été reprise d'Ariane 4 sans être mise à jour pour tenir compte de l'évolution des caractéristiques physiques du lanceur⁵.
- Entre 1985 et 1987, les machines de radiothérapie Therac-25 (fig. 3) ont causé six irradiations massives, dont trois mortelles. Le tout a été causé par la combinaison d'une pratique défailante du génie logiciel (manque de test), de l'absence de protections physiques (jugées redondantes), et de la présence de bogues de programmation concurrente de type "condition de course" (*race condition*).

Ces systèmes réactifs, où l'erreur est inadmissible, sont dits *critiques*. Leur conception et réalisation doit assurer un haut niveau de sûreté, et exige donc l'emploi d'une méthodologie adaptée.

Question 2. Pouvez-vous citer un exemple de système critique dans le secteur des transports ? Dans le secteur bio-médical ? Dans le secteur militaire ?

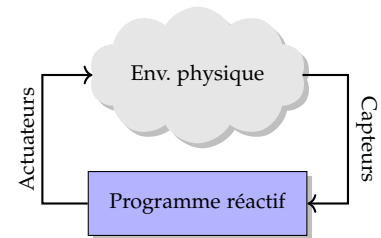


FIGURE 1: système réactif générique.

4. On verra ultérieurement que cette interdépendance se traduit de façon concrète par la pratique qui consiste à développer simultanément le programme réactif et un modèle logiciel de son environnement physique, afin de simuler leur interaction.



FIGURE 2: vol 501 d'Ariane 5.



FIGURE 3: machine Therac-25.

5. On peut approfondir sur les causes logicielles de cet échec et leurs conséquences matérielles en lisant le rapport de la commission d'enquête. Une copie est disponible [en ligne](#).

Contraintes temporelles. Les programmes réactifs sont souvent soumis à des contraintes dites de *temps-réel* — c'est presque toujours le cas s'ils appartiennent à un système critique. Un programme temps-réel doit absolument réagir en un temps borné maximal à certains changements de l'environnement. Manquer cette échéance peut entraîner un échec catastrophique de tout le système. Par exemple, le système **TCAS II**, employé dans l'aviation civile, est chargé d'avertir un pilote en cas de présence d'un appareil intrus dans une de ses trois zones d'intérêt (cf. fig. 4). Le pilote doit être averti très rapidement pour lui laisser le temps de réagir avant qu'une collision ait pu se produire.

Un des facteurs principaux qui impose des contraintes temps-réel aux programmes réactifs est leur interaction avec l'environnement physique. L'évolution de celui-ci est régie par des lois mathématiques qui évoluent en *temps continu*. À l'inverse, l'exécution du programme réactif prend la forme d'une séquence de transitions discrètes, autrement dit, en *temps discret*. Dès lors, il faut s'interroger sur la capacité du programme réactif à se faire une idée juste du système physique. Il s'agit d'un problème d'*échantillonnage* : le programme réactif doit observer le système physique à la bonne fréquence, ni trop lente, ni trop rapide. Autrement dit, la fréquence de réaction du programme fait partie intégrante de l'algorithme employé⁶.

Mathématiques dédiées. Le paragraphe précédent illustre l'importance des mathématiques appliquées dans la conception d'un programme réactif en interaction avec un environnement. En général, cet environnement évolue au cours du temps selon ses propres lois — on parle alors de *système dynamique*. La sous-discipline des mathématiques appliquées qui se consacre au contrôle des systèmes dynamiques s'appelle *l'automatique* (*control theory* en anglais). Ce contrôle peut passer par des dispositifs numériques, mais aussi purement électriques ou mécaniques⁷. Les théorèmes d'automatique fournissent par exemple des garanties sur la correction de l'échantillonnage, comme nous en avons discuté ci-dessus, mais aussi des outils d'analyse du comportement des systèmes dynamiques.

Le présent cours n'est pas un cours de mathématiques, aussi nous ne traiterons pas d'automatique à proprement parler. Toutefois, nous ferons référence à certains de ses résultats à plusieurs reprises, et nous montrerons comment certains concepts issus de l'automatique sont utiles pour la programmation de systèmes réactifs, sans détailler leurs sous-bassements théoriques⁸.

Contraintes de ressources. Enfin, les programmes réactifs se retrouvent en général exécutés par du matériel informatique dont c'est la seule fonction. On parle typiquement de *système embarqué*. De tels systèmes

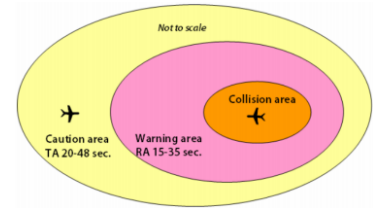
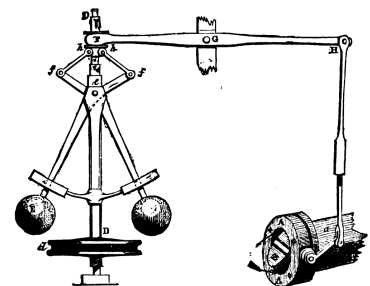


FIGURE 4: Zones d'intérêt du système d'évitement de collision TCAS II.

6. La littérature sur les systèmes temps-réel distingue souvent la *correction fonctionnelle* des contraintes temps-réel, et propose de les considérer indépendamment. Cet exemple montre que cette approche n'est pas toujours pertinente.

7. Un exemple célèbre de dispositif de contrôle mécanique est le régulateur à boules de James Watt.



Il permet de maintenir une machine à vapeur à une vitesse quasiment constante en contrôlant l'arrivée de vapeur. Celle-ci diminue si la machine est trop rapide, et augmente si elle est trop lente.

8. Si vous souhaitez découvrir ceux-ci, le livre d'Åström et Murray [6] offre une introduction à l'automatique illustrée de nombreux exemples. La deuxième édition est disponible [en ligne](#).

Si plusieurs familles de langages synchrones existent, nous nous focalisons sur celle des langages synchrones à *flots de données* (*data-flow* en anglais)⁹. Ceux-ci sont en effet largement les plus utilisés dans l'industrie, tout en restant en développement actif dans le monde universitaire. Ils facilitent la programmation réactive en offrant des concepts proches à la fois de l'automatique et de la programmation fonctionnelle. Ils sont issus de travaux pionniers de chercheurs grenoblois, qui ont voulu comprendre et généraliser la méthodologie empirique suivie par les ingénieurs pour concevoir les premiers dispositifs de contrôle numériques¹⁰. En une quarantaine d'années, ces langages ont contribué à l'évolution graduelle d'une méthodologie basée sur des schémas informels, suggestifs mais sans contenu calculatoire, à de véritables langages de programmation rigoureux, dotés d'une sémantique solide et de compilateurs sophistiqués (fig. 5).

Le reste du cours

Le but de ce cours est donc d'offrir une introduction aux systèmes réactifs ainsi qu'aux langages synchrones. Il sera structuré selon les deux axes décrits au début de ce texte, programmation et algorithmique, auxquels on adjoindra un troisième sujet, l'implémentation des langages synchrones, qui occupera la fin du semestre.

Tout d'abord, on pratiquera la programmation dans un langage synchrone à flots, le langage Heptagon. Heptagon est un langage très proche de SCADE 6, mais développé par des universitaires. Il dispose de fonctionnalités modernes : compilation séparée, automates imbriqués, gestion des tableaux. Sa proximité vis-à-vis de SCADE vous assure que vos compétences seront transférables facilement. De plus, il s'agit d'un logiciel libre.

Dans un second temps, nous discuterons des bases d'automatique appliquée qui servent de soubassements algorithmiques à la plupart des programmes réactifs. Nous n'entrerons quasiment pas dans les détails mathématiques, adoptant à la place une approche expérimentale de la conception de contrôleurs.

Enfin, nous terminerons le semestre avec une introduction à l'implémentation des langages synchrones. Il s'agit d'étudier les analyses statiques et techniques de génération de code employées par les compilateurs, les secondes reposant sur les premières. Pour bien comprendre leur fonctionnement, une initiation à la sémantique formelle des langages synchrones sera indispensable.

9. On pourra trouver un panorama historique des autres familles de langages synchrones dans l'article de Benveniste et al. [1].

10. On peut lire l'article [3] original de ces chercheurs au sujet du langage Lustre, l'ancêtre commun de tous les langages synchrones à flots de données.

Vous pouvez déjà avoir un aperçu du langage en consultant son site.

<http://heptagon.gforge.inria.fr>



La programmation synchrone à flots de données

On a vu qu'un programme réactif est en interaction continue avec un environnement physique via capteurs et actuateurs, comme illustré de façon schématique par la fig. 1. Les langages synchrones partent du principe que l'exécution d'un tel programme se produit de façon *cyclique*, c'est à dire comme une suite d'interactions entre le programme et son environnement. Un programme réactif n'agit en général pas uniquement en fonction de la valeur courante des capteurs, mais également de celles recueillies durant les interactions précédentes : il doit donc, pour ce faire, avoir accès à une *mémoire* dont le contenu persiste d'une interaction sur l'autre. De façon générale, chaque interaction se divise en trois phases distinctes, *lecture-calcul-écriture* :

- L. lecture des capteurs et de l'état courant de la mémoire ;
- C. phase de calcul des consignes des actuateurs et du nouvel état ;
- E. positionnement des actuateurs et mise à jour de la mémoire.

Enfin, les langages synchrones ne se préoccupent pas des détails bas-niveau d'accès aux capteurs et actuateurs. Il est donc utile de s'abstraire de la nature physique de l'environnement — sans toutefois oublier qu'elle induit les contraintes temporelles discutées précédemment. Une fois adopté ce point de vue simplifié, la structure d'un système réactif cyclique peut être représentée comme à la fig. 6, où capteurs et actuateurs ont été remplacés par des entrées et sorties génériques.

Le choix de structurer l'exécution comme une suite infinie d'interactions introduit naturellement une notion de *temps logique*. Ce temps logique est constituée d'une succession d'*instants logiques*, chacun d'eux correspondant à une interaction entre programme et environnement. Le caractère *logique* de cette temporalité réside dans l'omission volontaire du temps d'exécution. En d'autres termes, **un instant logique n'a pas d'épaisseur : du point de vue synchrone, l'interaction avec l'environnement est instantanée**. On appelle ce principe *l'hypothèse synchrone*. La fig. 7 illustre la relation entre temps logique et temps physique : le cycle Lecture-Calcul-Écriture (LCE) réalisé à chaque interaction peut prendre un temps d'exécution variable. De plus, rien ne garantit que les interactions soient exécutées à intervalle régulier.

On peut trouver l'hypothèse synchrone surprenante dans la mesure où, comme on l'a vu, de nombreux systèmes réactifs sont soumis à des contraintes temporelles strictes. On verra que ce choix simplifie en réalité leur mise au point en repoussant la prise en compte du temps physique aussi longtemps que possible durant le développement.

Les langages synchrones font donc le choix d'une exécution cyclique et d'une structuration du temps comme succession d'instants logiques. À un certain niveau d'abstraction, un programme synchrone

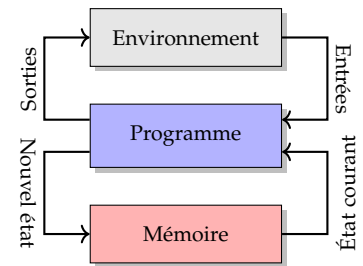


FIGURE 6: système réactif cyclique.

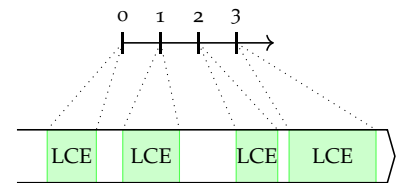


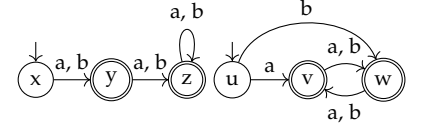
FIGURE 7: temps logique, temps physique.

peut donc être vu comme une machine à état, dont chaque transition correspondrait à un instant logique. Comme on cherche un formalisme mathématiquement simple pour décrire ces programmes, il est naturel de se tourner vers la théorie des automates et, plus précisément, des *transducteurs*, c'est à dire des automates finis qui, en plus de recevoir un mot en entrée, produit également un mot en sortie. Toutefois, les automates restent des objets complexes : par exemple, raisonner sur l'équivalence entre automates passe naturellement par la *bisimulation* (cf. fig. 8), une notion profonde mais relativement technique. Une autre difficulté est celle de la modularité : si on peut définir diverses manières d'assembler plusieurs automates en un automate plus gros — par exemple par la composition séquentielle de transducteurs — le résultat sera un automate à plat, sans structure. En bref, si le formalisme des automates est indispensable à la vérification de systèmes réactifs cycliques, il ne semble pas fournir un *langage* adapté au génie logiciel, du moins s'il n'est pas complété par d'autres principes de structuration.

On peut contraster la notion d'automate à une autre notion mathématique qui, bien qu'élémentaire, s'est révélée très compatible avec le génie logiciel : celle de fonction. Une fonction est un objet plus simple qu'un automate au sens où l'égalité entre fonctions est triviale — par définition, deux fonctions sont égales lorsqu'elles envoient les mêmes entrées vers les mêmes sorties. De plus, les langages dits “fonctionnels” tels que Haskell ou OCaml ont montré comment bâtir un langage de programmation au dessus de la notion de fonction. Il est donc naturel de chercher à concilier le monde des automates avec celui des fonctions.

Pour ce faire, on peut partir d'une observation simple. Si Σ_1 est l'alphabet du mot d'entrée et Σ_2 l'alphabet du mot de sortie, un transducteur déterministe¹¹ A implémente une fonction $f_A : \Sigma_1^* \rightarrow \Sigma_2^*$ telle que $f_A(w) = w'$ lorsque A produit le mot w' en lisant le mot w . Cette fonction a toujours plusieurs propriétés remarquables, notamment son caractère *synchrone* : elle associe toujours un mot de longueur n à un mot de longueur n . L'idée clef des langages synchrones dits à *flots de données* est de partir de la fonction synchrone pour aller vers l'automate, plutôt que l'inverse. Autrement dit, un programme va consister en une fonction synchrone, et c'est le compilateur qui va reconstruire le transducteur sous-jacent, celui-ci étant vu comme une implémentation concrète de la fonction (cf fig. 9). De plus, comme on s'intéresse aux systèmes réactifs, qui s'exécutent sans discontinuer, la fonction synchrone va consommer et produire non pas des mots finis dans Σ_1^* et Σ_2^* , mais des suites infinies de lettres, aussi appelée *flot*, et dont les ensembles sont dénotés Σ_1^ω et Σ_2^ω . On espère ainsi bénéficier du meilleur des deux mondes : la proximité des automates avec le modèle d'exécution sous-jacent, et l'expressivité des langages fonctionnels¹².

Les deux automates ci-dessous sont-ils équivalents, c'est à dire, reconnaissent-ils le même langage ? (Cet exemple simple est issu d'un article de Bonchi et Pous [2].)



On peut étudier cette question à l'aide de la notion de bisimulation. Écrivons S pour l'ensemble des états de nos automates. Une bisimulation est une relation $R \subseteq S \times S$ telle que, pour toute paire d'états $(s_1, s_2) \in R$, on ait :

1. s_1 est final ssi s_2 est final,
2. si $s_1 \xrightarrow{a} s'_1$ alors il existe s'_2 tel que $s_2 \xrightarrow{a} s'_2$ et $(s'_1, s'_2) \in R$,
3. si $s_2 \xrightarrow{a} s'_2$ alors il existe s'_1 tel que $s_1 \xrightarrow{a} s'_1$ et $(s'_1, s'_2) \in R$.

On peut montrer que deux états s_1, s_2 d'automates finis *déterministes* reconnaissent le même langage ssi il existe une bisimulation R telle que $(s_1, s_2) \in R$. Dans le cas qui nous occupe, existe-t-il une bisimulation R telle que $(x, u) \in R$? Oui ! Essayez de définir $R = \{(x, u), \dots\}$ en énumérant les couples manquants.

FIGURE 8: bisimulations entre automates.

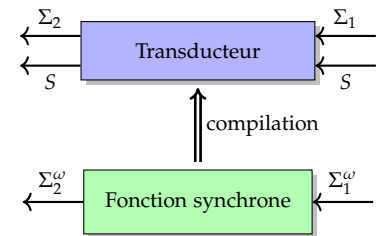


FIGURE 9: compilation des langages synchrones à flots de données.

11. Cette fonction peut être partielle si le transducteur est incomplet. Plus généralement, un transducteur non-déterministe donne lieu à une relation $R_A \subseteq \Sigma_1^* \times \Sigma_2^*$. Les relations implémentables par des transducteurs sont dites *rationnelles* (ou *régulières*), par analogie avec les langages et expressions rationnelles (ou régulières).
12. L'idée qui consiste à décrire des systèmes réactifs par des fonctions de flots est issue du travail pionnier de Kahn [4].

Premiers programmes

Nous allons maintenant explorer les langages synchrones à flots de données de façon concrète. Notre véhicule pour ce faire sera le langage Heptagon, qui est très proche du langage industriel SCADE 6 mais dont le compilateur est un logiciel libre.

Un programme Heptagon est un ensemble de fonctions synchrones sur les flots de données. On appelle ces fonctions des *nœuds*. Chaque nœud dispose d'une *interface*, liste finie de sorties et d'entrées déclarées avec leurs types, ainsi que d'un *corps*, qui est une liste d'équations définissant la valeur des sorties en fonction de celles des entrées. On peut par exemple définir un nœud correspondant à la fonction identité sur les entiers comme ci-dessous.

```
node identite(x : int) returns (y : int)
let
  y = x;
tel
```

Il est important de remarquer que le type **int**, en Heptagon, ne désigne pas un unique entier, mais un *flot* d'entiers. Il en va de même pour les types **float**, **bool**, etc. Le nœud ci-dessous représente donc la fonction mathématique $id : \mathbb{Z}_{32}^\omega \rightarrow \mathbb{Z}_{32}^\omega$, où \mathbb{Z}_{32} désigne l'ensemble des entiers relatifs représentable en complément à deux sur 32 bits¹³.

On peut compiler un programme Heptagon en demandant au compilateur heptc de produire une sortie en langage C. Les fichiers sources ainsi générés contiennent une implémentation du transducteur sous une forme ressemblant¹⁴ au code ci-dessous.

```
/* Définition de l'état du transducteur. */
struct identite_state { };
/* Définition de la fonction d'initialisation de l'état. */
void identite_reset(struct identite_state *state) { }
/* Définition de la fonction de transition. */
void identite_step(struct identite_state *state,
                  int x, int *y) { *y = x; }
```

On verra lors des cours et séances de travaux pratiques suivants une façon commode d'exécuter la fonction de transition.

Un nœud Heptagon peut également disposer de variables *locales*, qui ne sont ni des entrées ni des sorties. Elles doivent être déclarées avec le mot clef **var** et définies dans le corps du nœud.

```
node identite_bis(x : int) returns (y : int)
var z : int;
let
  z = x;
```

On peut représenter le comportement d'un nœud sur une entrée choisie à l'aide d'un *chronogramme*, c'est à dire d'un tableau dont chaque colonne correspond à un instant logique distinct. Par exemple, le chrono

x	-2	0	1	-3	2	2	...
y	-2	0	1	-3	2	2	...

Ce chronogramme illustre le caractère synchrone de la fonction identité, vue comme agissant sur des flots : les n premières valeurs de y dépendent uniquement des n premières valeurs de x . On verra que ce sera aussi le cas de fonctions bien plus complexes.

13. En réalité, Heptagon ne fixe pas la taille des entiers utilisés, mais aligne son type **int** au type **int** du langage C. Sa taille dépend donc de votre machine, et plus précisément du compilateur C utilisé pour compiler le code généré par Heptagon (cf. plus bas).
14. En pratique, le compilateur applique certaines transformations qui rendent le code moins lisible mais plus efficace et plus court.


```

    y = z;
tel

```

Un point très important, commun à tous les langages synchrones à flots de données, est que l'ordre des définitions n'importe pas. Ainsi, on peut réécrire notre fonction identité de façon strictement équivalente mais en définissant y avant z.

```

node identite_ter(x : int) returns (y : int)
var z : int;
let
    y = z;
    z = x;
tel

```

Cet exemple montre que le point-virgule qui sépare les équations n'a rien à voir avec une construction de séquentialisation, comme en C, Java ou OCaml. Au contraire, il s'agit simplement de marquer la fin d'une équation dans le bloc de définitions *mutuellement récursives* compris entre **let** et **tel**.

La possibilité d'écrire des équations mutuellement récursives est nécessaire pour pouvoir écrire des fonctions de flots générales, comme on le verra ultérieurement. Elle ouvre néanmoins la porte à la possibilité d'erreurs. Considérons le code ci-dessous, en apparence une simple modification du précédent.

```

node identite_bad(x : int) returns (y : int)
var z : int;
let
    y = z;
    z = y;
tel

```

Ce programme est très suspect : on a défini y en fonction de z, et vice-versa ! Si l'on essaie de le compiler avec heptc, on obtient un message d'erreur.

```

$ heptc -target c ex-04-bad.ept
Causality error: the following constraint is not causal.
^z < y || ^y < z

```

On appelle les erreurs causées par ce genre de définitions circulaires, ou *cercles vicieux*, des erreurs de *causalité*¹⁵. L'étude de la notion de causalité est au coeur des langages synchrones, et il faut en comprendre le fonctionnement général pour programmer productivement dans un langage comme Heptagon.

15. On trouve parfois employé le terme plus sobre de *productivité*.

Références

- [1] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The Synchronous Languages 12 Years Later. *Proceedings of the IEEE*, 2003.
- [2] F. Bonchi and D. Pous. Checking NFA equivalence with bisimulations up to congruence. In *Principles of Programming Languages (POPL'13)*. Association for Computing Machinery, 2013.
- [3] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE : A declarative language for programming synchronous systems. In *Principles of Programming Languages (POPL'87)*. Association for Computing Machinery, 1987.
- [4] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing Congress (IFIP'74)*. IFIP, 1974.
- [5] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer New York, 1992.
- [6] K. J. Åström and R. M. Murray. *Feedback Systems : An Introduction for Scientists and Engineers*. Princeton University Press, Jan 2008.