Rapport de stage de M2 | Avril - Août 2018

Observateurs pour le test à la *QuickCheck* de systèmes hybrides

Ismail Lahkim Bennani, encadré par Marc Pouzet
Équipe Parkas, École Normale Supérieure

Le contexte général

De plus en plus d'industries ont recours aux systèmes embarqués dans leurs produits (e.g, logiciel de commandes de vol, aiguillages des trains). Ces systèmes intègrent du logiciel de contrôle et de communication temps réel interagissant avec des outils de mesures physiques. Ces systèmes, dits cyber-physiques, sont réputés complexes et difficile à prédire ce qui fait de leur conception un véritable défi.

Une approche répandue pour développer ces systèmes est la conception basée sur les modèles (*model based design*). Les composants du système sont décrit à haut niveau avant d'être simulés numériquement ou compilés vers du code bas niveau.

Un outil utilisé dans l'industrie est SCADE [CPP17], dont le langage est basé sur Lustre [HCRP91]. Ces langages sont fondés sur un modèle de *temps synchrone*. On y programme un système en décrivant par une spécification mathématique son comportement : on y écrit des suites et des fonctions de suites. Ces suites sont indicées par du temps logique, c'est-à-dire par des moments qui se succèdent instantanément.

Les *systèmes hybrides* sont des systèmes qui combinent des signaux à temps continu et des évolutions discrètes de ces signaux. Ces langages permettent de décrire les signaux continus par des équations différentielles ordinaires (ODEs). Ces équations sont ensuite calculées par un solveur numérique durant l'exécution du programme. Un outil utilisé dans l'industrie pour décrire ces systèmes est Simulink [1].

Se pose alors la question du test et du débogage de ce type de programmes.

Le problème étudié

Les systèmes cyber-physiques sont utilisés pour des applications critiques, il est primordial de s'assurer de leur bon fonctionnement. Tester un programme, c'est vérifier qu'il respecte une spécification donnée.

Cette question n'a rien de nouveau. Dans la littérature du test de systèmes hybrides, on peut distinguer deux types d'approches. D'une part, le test à base de modèles (*model-based testing*) : il se veut relativement précis au prix d'une grande complexité (résolution de problèmes NP-complets, utilisation de solveurs SAT ...), il utilise une représentation abstraite du système sous test pour vérifier sa spécification. D'autre part, le test à base de propriétés (*property based testing*) [DM10] [FP09] [BBBK16], à la *QuickCheck* [CH00] : l'idée est d'essayer de casser la spécification du programme en l'exécutant un grand nombre de fois sur des entrées différentes.

Des outils de test à base de propriété existent pour traiter les systèmes synchrones. On peut citer Lutin [RRJ08] (pour Lustre) pour traiter les systèmes à temps discret. D'autres tels que Taliro [ALFS11] et Breach [Don10] (pour Simulink) existent pour traiter des systèmes à temps continu.

1. https://fr.mathworks.com/products/simulink.html

Ces outils utilisent des logiques temporelles comme la Metric Temporal Logic (MTL) [Koy90] ou la Metric Interval Temporal Logic (MITL) [AFH96] pour décrire la spécification des systèmes à tester. Ces logiques ne sont pas causales, c'est-à-dire que la véracité de la formule à un instant donné peut dépendre des valeurs futures.

Cela a conduit ces outils à faire de la vérification hors ligne ; ils commencent par simuler le système pendant un temps donné puis récupèrent une trace, c'est-à-dire une version échantillonnée des signaux simulés, qu'ils utilisent pour déterminer la valuation d'une formule. Le problème de cette approche est que la vérification hors-ligne oblige à aller au bout de la simulation entre chaque test.

Notre travail porte sur le test de systèmes hybrides, par une approche à base de propriétés, appliquée au langage hybride Zélus [BP13]. Je me suis demandé s'il est possible de faire du test de systèmes hybrides en ligne : c'est-à-dire de trouver un moyen de décrire des signaux continus de sorte à pouvoir les générer automatiquement, et de trouver un moyen de spécifier ces signaux de sorte à pouvoir évaluer la véracité de cette spécification en ligne.

La contribution proposée

En ce qui concerne la spécification des signaux, j'ai d'abord commencé par écrire une version en ligne des algorithmes de l'état de l'art. Cependant, cela m'a obligé à interrompre régulièrement le solveur numérique durant la simulation.

Je me suis alors demandé s'il était possible de résoudre ce problème sans arrêter le solveur. J'ai d'abord essayé d'adapter les méthodes précédentes, c'est à dire de traduire les étapes de calcul discrets en calculs d'ODEs, sans succès. Puis j'ai essayé de traduire les formules de certaines logiques temporelles en automates hybrides [MNP06], mais cette solution n'était pas satisfaisante. Finalement, la solution que je propose est une librairie d'observateurs hybrides inspirée du fragment passé de la logique MITL. En ce qui concerne la génération d'entrées, j'ai écrit une librairie basée sur certaines idées que l'on retrouve dans Lutin [RRJ08] adaptées aux signaux continus.

Les deux librairies sont écrites en Zélus.

Les arguments en faveur de sa validité

La solution proposée est effectivement en ligne, ce qui implique d'une part qu'elle termine le plus tôt possible et d'autre part qu'elle peut être utilisée pour faire de la surveillance en temps réel. Les librairies que je propose sont suffisamment expressives pour traiter les exemples utilisés dans les différents outils Breach et S-TaLiro.

Par ailleurs, cette solution est modulaire : les deux librairies sont composées de nœuds de base composables pour fabriquer des signaux d'une part et spécifier des signaux d'autre part.

Le bilan et les perspectives

La compositionnalité de cette approche fait qu'elle peut facilement s'adapter à des cas nouveaux ; il est facile d'étendre les librairies pour des cas qui ne seraient pas supportés actuellement.

Toutefois, plusieurs questions restent en suspend. D'une part, l'un des objectifs du test est de produire de petits contre-exemples, il faut définir cette notion dans le cas de signaux continus. D'autre part, dans mon travail j'ai considéré les systèmes à tester comme des boites noires, pourtant on a accès à leur code. Cette information est déjà utilisée pour générer des cas de tests [MNBB16], dans quelle mesure peut-on utiliser la spécification du système pour les améliorer ?

Table des matières

I. SYSTÈMES SYNCHRONES HYBRIDES

Les systèmes embarqués sont souvent limités en ressources (mémoire, processeur, ...). Par ailleurs, ils sont en constante interaction avec leur environnement, souvent à travers des capteurs physiques (température pour une bouilloire, champs électromagnétiques pour un récepteur radio, ...).

Une approche répandue pour développer ces logiciels est la programmation synchrone. Je vais présenter succinctement cette façon de programmer à travers le langage Zélus [BP13] [2].

a. Systèmes à temps discret

Les programmes synchrones sont des programmes basés sur le modèle *temps synchrone*. Ils permettent d'écrire des programmes *réactifs*, c'est-à-dire des programmes réagissant aux variations de leur environnement. Les valeurs qu'ils manipulent sont des *flots* de données. Ces flots peuvent être vus comme des suites indicées par du temps logique, c'est-à-dire un ensemble de moments qui se succèdent instantanément. Les opérateurs arithmétiques, les opérateurs booléens et les fonctions sont donc définis sur des flots (figure 1).

Exemple Demi-aditionneur et additionneur implémentés en Zélus.

```
let fun half_add(a: bool, b: bool) = (s: bool, co: bool) where
      s = a xor b
  and c = a and b
```

```
let fun full_add(a: bool, b: bool, c:bool) = (s: bool, co: bool) where
    rec s1, c1 = half_add(a, b)
    and s, c2 = half_add(s1, c)
    and co = c1 or c2
```

inputs	a	false	true	true	false	true	...
	b	true	false	true	false	true	...
	c	false	true	false	true	true	...
half_add(a, b)	s	true	true	false	false	false	...
	co	false	false	true	false	true	...
full_add(a, b, c)	s	true	false	false	true	true	...
	co	false	true	true	false	true	...

Résultats de simulation

Un programme est un ensemble d'équations. Lors de l'exécution de ce programme, ces équations sont exécutées une fois à chaque entrée reçue.

Pour pouvoir les exécuter, ces équations doivent être mises dans un bon ordre. C'est un des rôles du compilateur : l'ordonnancement (*scheduling*) des équations. Une condition suffisante pour qu'un ordonnancement existe est qu'il n'existe aucune boucle de dépendance dans le calcul. Dans ce cas on peut faire les calculs dans un ordre topologique.

Exemple Le système `x = 2. *. y and y = 2. *. z and z = 0.5 *. x` est rejeté par le compilateur zeluc : bien qu'il admette une solution unique $x = y = z = 0$, il y a une boucle de dépendance dans le calcul : x dépend de y qui dépend de z qui dépend de x.

2. cf. http://zelus.di.ens.fr/man/ pour une présentation plus détaillée

	c	false	true	...	false
	x	x_0	x_1	...	x_n
	y	y_0	y_1	...	y_n
Opérateurs	x ⋄ y	$x_0 \diamond y_0$	$x_1 \diamond y_1$	...	$x_n \diamond y_n$
Conditionnelle	if c then x else y	y_0	x_1	...	y_n
Retard	pre x	nil	x_0	...	x_{n-1}
Initialisation	x → y	x_0	y_1	...	y_n

Figure 1 – *Quelques expressions Zélus, $\diamond \in \{ +, -, *, /, >, \geq, <, \leq, ==, <>, \&,$ or, xor, xand $\}$, nil est une valeur arbitraire du bon type.*

Remarque
Les boucles de dépendance peuvent être cassées par un retard pre.

Il y a plusieurs sortes de fonctions, parmi elles on a les nœuds discrets (définis par la construction let node). Ce sont des fonctions à mémoire et à temps discrets (*stateful functions*) manipulant des flots. Elles peuvent entre autre utiliser les valeurs passées de leurs variables locales grâce à la primitive pre. On a aussi des fonctions combinatoires (définies par la construction let fun). Ce sont des fonctions sans mémoire (*stateless functions*) leurs sorties dépendent uniquement de la valeur courante de leurs entrées.

Exemple Bouilloire (exemple tiré du cours d'introduction sur Lustre donné par Nicolas Halbwachs au Collège de France[3])

Une bouilloire contient de l'eau à une température temp. La résistance chauffante de la bouilloire a une température c lorsqu'elle est allumée et 0 sinon. L'état de la résistance est donné par un booléen u. Lorsque la résistance est allumée, l'eau chauffe et on a que $\dot{temp} = \alpha(c - temp)$ où $\dot{temp}$ est la dérivée de $temp$. Lorsque la résistance est éteinte, l'eau refroidit et on a que $\dot{temp} = \beta(temp\_ext - temp)$. La température initiale de la bouilloire est temp0. On veut pouvoir simuler l'évolution de la température en fonction des autres paramètres.

```
let node heater_d(h, c, alpha, beta, temp_ext, temp0, u) = temp where
    rec temp = temp0 -> pre(temp +. dtemp *. h)
    and dtemp = if u then alpha *. (c -. temp)
                else beta *. (temp_ext -. temp)
```

Le nœud heater_d calcule l'évolution de la température à l'aide d'un schéma d'Euler de pas h.
Le contrôleur de la bouilloire s'occupe d'allumer ou d'éteindre la résistance. Étant donné un seuil de température haut *high* et un seuil bas *low*, on a que si $temp > high$ alors $u = false$ et si $temp < low$ alors $u = true$. Une implémentation possible du contrôleur est la suivante.

```
let node relay_d(low, high, v) = u where
    rec u = if v < low then true
            else if v > high then false
            else false -> pre u
```

Le nœud heater_d est compilé par le compilateur zeluc en un code Caml. Il est représenté par :

— une fonction heater_d_alloc qui sert à créer la mémoire du programme représentée par le type enregistrement

```
type ('b, 'a) _heater_d =
  { mutable i_32 : 'b; mutable m_28 : 'a }
```

3. https://www.college-de-france.fr/site/gerard-berry/seminar-2010-01-13-11h00.htm

```
let node main_d(reference) = (u, temp) where
  rec u = relay_d(reference -. 1.0, reference +. 1.0, temp)
  and temp = heater_d(0.1, 50.0, 0.1, 0.1, 0.0, 0.0, u)
```

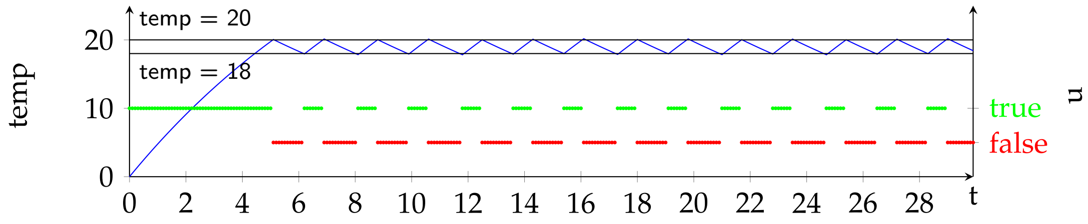


Figure 2 – *Composition des nœuds* relay_d *et* heater_d *et résultats de simulation du programme avec* reference = 19.0 *avec un échantillonnage de 0.1 secondes. À partir de* $t = 6$*, les valeurs extrémales de la températures sont* 17.84 *et* 20.32.

Cette mémoire est composée d'un booléen i_{32} qui vaut true à l'instant initial et d'un entier m_{28} qui contient la valeur précédente de l'expression temp +. dtemp *. h.

— une fonction heater_d_reset qui sert à initialiser la mémoire du programme (ou à la réinitialiser au besoin).

— une fonction heater_d_step qui fait un pas de calcul du programme. Elle assigne la dernière valeur de l'expression temp +. dtemp *. h à temp puis elle calcule la nouvelle valeur de dtemp Enfin, elle met à jour la mémoire pour l'étape suivante.

Si l'on compose les deux nœuds de l'exemple précédent et que l'on définit les constantes de l'environnement, on obtient un modèle de bouilloire qui essaye de maintenir son eau à une température proche d'une température de référence (figure 2).

Remarque
Les valeurs de la température sur la simulation dépassent les bornes fixées dans le code. C'est dû à l'échantillonnage.

b. Systèmes à temps continu

Zélus permet aussi d'écrire des fonctions à temps continu grâce à la construction let hybrid, on appelle ces fonctions des nœuds hybrides. Dans un nœud hybride, on peut définir les variables par des équations différentielles ordinaires (*ODEs*). La construction der x = y init x_0 représente l'équation $x(t) = x_0 + \int_0^t y$. C'est un signal à temps continu.

La primitive up définit un événement dit de traversée de zéro (*zero-crossing*), c'est l'ensemble des instants où son argument passe d'une valeur strictement négative à une valeur strictement positive.

Un événement est un ensemble d'instants de $\mathbb{R}$. La primitive present permet de surveiller ces événements, c'est-à-dire de créer un bloc de code qui sera exécuté ponctuellement, à chaque occurrence de l'événement surveillé.

Exemple Bouilloire, temps continu (figure 3)

```
let hybrid heater_c(c, alpha, beta, temp_ext, temp0, u) = temp where
    rec der temp = dtemp init temp0
    and dtemp = if u then alpha *. (c -. temp)
                else beta *. (temp_ext -. temp)
let hybrid relay_c(low, high, v) = u where
  rec u = present
        | up(low -. v) -> true
        | up(v -. high) -> false
        init (v < high)
```

```
let hybrid main_c(reference) = (u, temp) where
  rec u = relay_c(reference -. 1.0, reference +. 1.0, temp)
  and temp = heater_c(50.0, 0.1, 0.1, 0.0, 0.0, u)
```

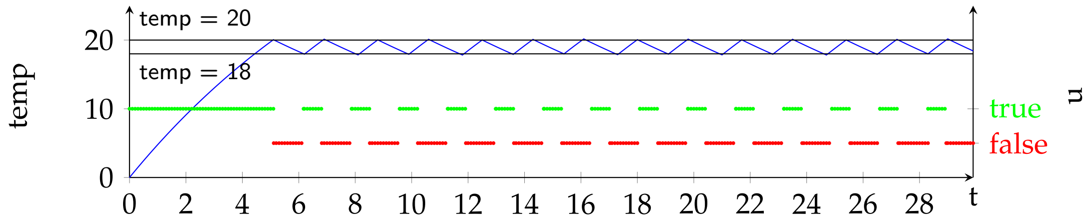


Figure 3 – *Composition des nœuds* relay_c *et* heater_c *et résultats de simulation du programme avec* reference = 19.0 *et un échantillonnage de 0.1 secondes. A partir de $t = 6$ les valeurs extrémales de la température sont* 18.00 *et* 20.00.

Sur la simulation de la bouilloire à temps continu, la température ne dépasse pas les bornes définies dans le programme. Dans la version discrète, le relais décidait s'il fallait allumer ou éteindre la bouilloire à chaque étape de la simulation, c'est-à-dire toutes les 0.1 secondes. Dans la version continue, il prend cette décision à chaque fois que v = low ou que v = high.

Remarque
En pratique, le solveur approxime cet instant, mais dans cet exemple l'approximation était suffisamment correcte pour que la valeur de *temp* soit égale à la borne avec une erreur inférieure à 10^{-6}.

Néanmoins, sur cet exemple en particulier, nous n'avons pas besoin d'intégration pour trouver la valeur de la température. On peut résoudre analytiquement le système et utiliser un nœud discret pour calculer les solutions de $temp(t) = 18$ et $temp(t) = 20$, ce qui nous permettrait de sauter d'événement en événement.

L'utilisation de nœuds hybrides est particulièrement utile lorsqu'on ne sait pas résoudre analytiquement le système d'équation. Dans ce cas on ne peut pas prédire quand auront lieu les événements, il faut les calculer durant la simulation et c'est ce que fait le solveur.

c. Un exemple hybride : Système de transmission automatique

J'utiliserai dans ce rapport l'exemple d'un système de transmission automatique. Ce modèle a été écrit en Simulink[4] par MathWorks[5] pour en faire un des exemples fournis par l'outil. Je l'ai implémenté en Zélus.

Ce modèle prend en entrée l'accélération (en %) et le couple de freinage (en $ft.lb^{-1}$) du véhicule et calcule sa vitesse (en $miles.hr^{-1}$), son rapport de vitesse actuel et la vitesse de son moteur (en $tr.min^{-1}$).

Des exemples des simulations du système sont disponibles en annexe B.

Ces résultats ont été obtenus grâce à Zélus et sont proches des résultats obtenus avec le modèle implémenté dans Simulink. Des courbes comparant les résultats Zélus aux résultats Simulink sont également disponibles en annexe B.

Voir annexe A pour une présentation du modèle physique à simuler et des implémentations Simulink et Zélus.

II. État de l'art

Quelle que soit la méthode utilisée pour tester un programme, il faut (figure 4) :

4. https://fr.mathworks.com/products/simulink.html
5. https://fr.mathworks.com/

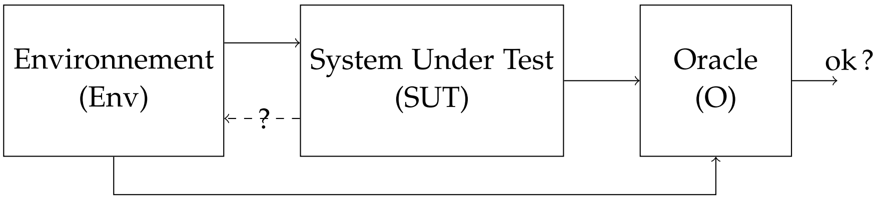


Figure 4 – *Schéma général du test*

— une description des entrées possibles (*environnement*)
— une description des couples (entrées, sorties) valides (*oracle*)

Ces descriptions sont ensuite utilisées par l'outil de test pour chercher de mauvais comportements. Plusieurs approches existent pour résoudre ce problème. L'approche que j'ai choisie d'utiliser est l'approche dite à base de propriétés (Property-Based).

Cette approche demande de pouvoir exécuter les programmes (l'environnement, le système et l'oracle). D'autres méthodes n'ayant pas ce prérequis existent : on peut par exemple décrire le système par un automate hybride (possiblement non déterministe) dont l'alphabet est l'ensemble des valeurs autorisées par l'environnement. On peut alors traduire la description de l'oracle en un automate reconnaissant exactement le langage des mauvaises traces. Il faut ensuite montrer que le langage reconnu par la composition de ces deux automates est vide.

Cette méthode utilise une approche dite à base de modèles (*model based*), elle est par exemple utilisée en combinaison avec un oracle décrit par une formule de la logique temporelle MITL [MNP06].

Remarque
Cette logique est aussi utilisé dans des outils utilisant l'approche à base de propriétés.

a. Test à base de propriétés (Property-Based Testing)

L'approche que j'ai étudié et mis en œuvre dans mon stage est celle du test à base de propriété. L'idée fondamentale est qu'on cherche à tester une spécification (une propriété) en exécutant le programme sur un grand nombre d'entrées différentes.

L'outil emblématique utilisant cette méthode est QuickCheck [CH00]. C'est une librairie Haskell qui permet d'écrire des procédures de tests aléatoires et de les répéter un nombre arbitraire de fois (typiquement des centaines) pour trouver des couples (entrées, sorties) que l'oracle refuse.

Les principaux objectifs de ses auteurs étaient de faire un outil simple et modulaire. L'idée était d'automatiser les tâches répétitives de l'écriture de tests. Aujourd'hui, cet outil a été ré-implémenté dans de nombreux langages (38 implémentations sont citées sur la page Wikipédia de QuickCheck [6]).

Les propriétés à vérifier sont décrites par des fonctions à valeurs booléennes. Les entrées sont créées par des générateurs : ce sont des fonctions qui, étant donnés un entier représentant une taille renvoient un objet du bon type. Ces générateurs peuvent ensuite être utilisées pour créer de nouveaux générateurs.

Par exemple, un générateur possible pour le type `Int` est la fonction qui prend une taille n en argument et renvoie un nombre aléatoire dans $[-n, n]$. Un générateur possible pour le produit cartésien (`Int`, `Int`) est le générateur précédent appelé deux fois.

L'argument entier des générateurs sert à limiter la taille de la valeur produite : dans le cas du générateur de listes présent par défaut dans l'outil, il sert à fixer la taille de la liste à générer.

Pour avoir plus de chance de tomber sur un petit contre-exemple, QuickCheck va générer des entrées de taille croissante durant les tests.

b. Logique MITL : comment spécifier un signal

Les logiques temporelles sont des représentations formelles utilisées dans la littérature pour spécifier les programmes synchrones et hybrides. On peut citer parmi elles la logique linéaire temporelle (LTL)

6. https://en.wikipedia.org/wiki/QuickCheck

[Pnu77] pour décrire des propriétés sur des signaux discrets, son extension, la logique linéaire métrique (MTL) [Koy90] pour les signaux continus ou encore une restriction de cette dernière, la Metric Interval Temporal Logic (MITL) [AFH96] (figure 5).

En plus des opérateurs classiques $\wedge$ et $\neg$, LTL définit des opérateurs temporels : le X (Ensuite, Next) et le U (Jusqu'à, Until). L'opérateur X indique la propriété que le signal doit vérifier à l'instant suivant tandis que le U encode l'idée de séquence. La propriété $\varphi_1 \; U \; \varphi_2$ est vérifiée par un signal s à l'instant t si ce signal vérifie φ_1 jusqu'à un certain instant $t' \geq t$, et qu'il vérifie φ_2 à l'instant t'.

Dans le cas de MTL, l'opérateur X n'aurait pas de sens ; il n'y a pas de prochain instant en temps continu. Pour exprimer la même idée, on indice l'opérateur U avec un intervalle qui contraint le temps auquel survient le changement dans la séquence. Par ailleurs la sémantique du U impose que φ_2 soit vérifiée par le signal au moins une fois dans l'intervalle $t +_{\mathbb{R}} I$. Ainsi, pour dire "dans 1 à 2 secondes φ est vérifiée" on écrira $\top \; U_{[1;2]} \; \varphi$ (ou $\Diamond_{[1;2]}\varphi$).

Ces formules sont utilisées pour donner une spécification formelle du système qui puisse être calculée numériquement.

Par ailleurs, il est utile de définir une sémantique quantitative sur les formules : plutôt que de répondre à la question "φ est-elle vérifiée par s à l'instant t ?", on répondra à la question "à quel point φ est-elle vérifiée par s à l'instant t".

Cette sémantique quantitative est utilisée pour transformer un problème de falsification (trouver une entrée qui contredit l'oracle) en un problème d'optimisation (trouver une entrée qui a une image négative par l'oracle).

L'oracle renverra donc une valeur réelle plutôt que booléenne : le signe du résultat représentera la valeur de vérité booléenne (≥ 0 signifie vrai) et sa valeur absolue représentera le degré de certitude de cette valeur de vérité. On appelle cette valeur la *robustesse* de la formule [FP09] sur le signal s à l'instant t (figure 6).

Exemple quelques propriétés et leur représentation en MITL (*gear*, *speed* et *rpm* sont des fonctions du temps). On définit deux nouveaux opérateurs :

$$\begin{aligned} Ncessairement\ (Eventually) &: \Diamond_{[a;b]}\varphi = \top \; U_{[a;b]}\varphi \\ Toujours\ (Always) &: \Box_{[a;b]}\varphi = \neg \; \Diamond_{[a;b]}(\neg \; \varphi) \end{aligned}$$

- La vitesse n'est jamais inférieure à 30 mph lorsqu'on est en 3e vitesse
  $\hookrightarrow$ MITL : $\Box\Big(\neg \; (gear(t) = 3 \wedge speed(t) < 30)\Big)$
- Lorsqu'on passe en 2e vitesse, on y reste pendant au moins 1 seconde
  $\hookrightarrow$ MITL : $\Box\Big(\Big((\neg \; gear(t) = 2) \wedge \Diamond_{[0.01,0.02]}gear(t) = 2\Big) \Rightarrow \Box_{[0,1]}gear(t) = 2\Big)$
- Pendant les 25 premières secondes, la vitesse est supérieure à 30 mph, pendant les 25 secondes suivantes, la vitesse est inférieure à 150 mph
  $\hookrightarrow$ MITL : $\Big((\Box_{[0,25]}speed(t) < 150) \wedge (\Box_{[25,50]}speed(t) > 30)\Big)$
- Soit la vitesse est toujours inférieure à 100 mph, soit elle est toujours supérieure à 30 mph et est supérieure à 100 mph dans les 25 premières secondes
  $\hookrightarrow$ MITL : $\Big((\Box \; speed(t) < 100) \vee (\Diamond_{[0,25]}(speed(t) > 100) \wedge (\Box \; speed(t) > 30))\Big)$
- L'assertion "la vitesse est supérieure à 100 mph dans la première seconde et le moteur tourne toujours à moins de 4000 rpm" est fausse
  $\hookrightarrow$ MITL : $\neg\Big((\Diamond_{[0,1]}speed(t) > 100) \wedge (\Box \; rpm(t) < 4000)\Big)$

c. Test à base de propriétés dans Simulink

J'ai étudié deux boites à outils Simulink : Breach [Don10] et S-TaLiro [ALFS11]. Elles permettent d'évaluer la robustesse (et a fortiori la valeur de vérité) d'une formule MITL sur un signal échantillonné. Elles implémentent aussi des outils de falsification de propriétés MITL, c'est-à-dire des outils pour trouver une entrée de l'environnement telle que son image par le système sous test contredise la

$$\text{LTL} \quad \varphi \ := \ \top \mid v \sim f \mid \neg\, \varphi \mid \varphi \wedge \varphi \mid X\ \varphi \mid \varphi\ U\ \varphi$$

avec v une variable, $\sim$ un opérateur de comparaison et f un flottant

$$\begin{aligned}
[\![\top]\!]_{LTL}(s,i) &= true \\
[\![\neg\ \varphi]\!]_{LTL}(s,i) &= \neg\ [\![\varphi]\!]_{LTL}(s,i) \\
[\![\varphi_1 \wedge \varphi_2]\!]_{LTL}(s,i) &= [\![\varphi_1]\!]_{LTL}(s,i) \wedge [\![\varphi_2]\!]_{LTL}(s,i) \\
[\![X\ \varphi]\!]_{LTL}(s,i) &= [\![\varphi]\!]_{LTL}(s,i+1) \\
[\![\varphi_1\ U\ \varphi_2]\!]_{LTL}(s,i) &= \exists\ i_0 >= i, \\
&\quad (\forall\ i_1 \in [i,i_0], [\![\varphi_1]\!]_{LTL}(s,i_1)) \wedge \\
&\quad [\![\varphi_2]\!]_{LTL}(s,i_0)
\end{aligned}$$

$$\text{M(I)TL} \quad \varphi \ := \ \top \mid v \sim f \mid \neg\, \varphi \mid \varphi \wedge \varphi \mid \varphi\ U_I\ \varphi$$

avec v une variable, $\sim$ un opérateur de comparaison, f un flottant et

— $I \subset \mathbb{R}$ pour MTL
— $I = [a;b] \mid a,b \in \mathbb{R}^+ \cup \{+\infty\}$ et $a < b$ pour MITL

$$\begin{aligned}
[\![\top]\!]_{M(I)TL}(s,t) &= true \\
[\![\neg\ \varphi]\!]_{M(I)TL}(s,t) &= \neg\ [\![\varphi]\!]_{M(I)TL}(s,t) \\
[\![\varphi_1 \wedge \varphi_2]\!]_{M(I)TL}(s,t) &= [\![\varphi_1]\!]_{M(I)TL}(s,t) \wedge [\![\varphi_2]\!]_{M(I)TL}(s,t) \\
[\![\varphi_1\ U_I\ \varphi_2]\!]_{M(I)TL}(s,t) &= \exists\ t' \in\ t +_{\mathbb{R}} I, \\
&\quad (\forall\ t'' \in [t,t'], [\![\varphi_1]\!]_{M(I)TL}(s,t'')) \wedge \\
&\quad [\![\varphi_2]\!]_{M(I)TL}(s,t')
\end{aligned}$$

avec $t +_{\mathbb{R}} I = \{t' \mid \exists t_I \in I, t' = t + t_I\}$

Figure 5 – *Syntaxe et sémantique de formules LTL, MTL et MITL. $[\![\varphi]\!]_{LTL}(s,i) \in Bool$ est la valuation de φ sur la trace s à l'instant $i \in \mathbb{N}$. $[\![\varphi]\!]_{M(I)TL}(s,t) \in Bool$ est la valuation de φ sur la trace s à l'instant $t \in \mathbb{R}$*

$$\begin{aligned}
\rho(\top)(s,t) &= \ +\infty \\
\rho(\neg\ \varphi)(s,t) &= \ -\ \rho(\varphi)(s,t) \\
\rho(\varphi_1 \wedge \varphi_2)(s,t) &= \ \min(\rho(\varphi_1)(s,t),\ \rho(\varphi_2)(s,t)) \\
\rho(\varphi_1\ U_I\ \varphi_2)(s,t) &= \min_{t' \in (t +_R I)} \left(\max\left(\rho(\varphi_2)(s,t'), \max_{t<t''<t'} \rho(\varphi_1)(s,t'')\right)\right)
\end{aligned}$$

Figure 6 – *Sémantique quantitative d'une formule MITL, $\rho(\varphi)(s,t) \in \mathbb{R} \cup \{+\infty\}$ est la robustesse de φ sur la trace s à l'instant $t \in \mathbb{R}$*

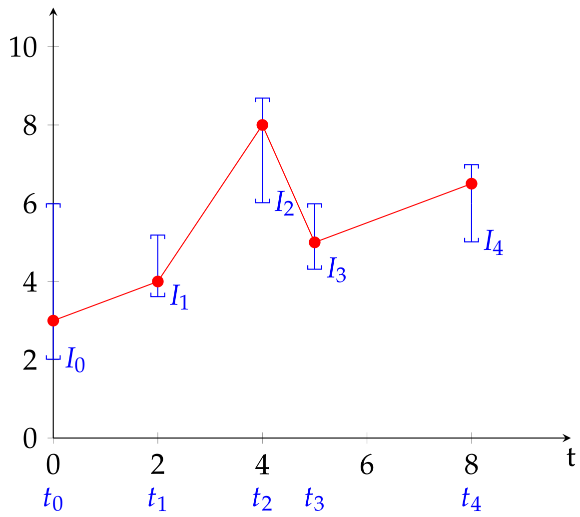


Figure 7 – *Exemple de génération d'entrée (interpolation de degré 1) dans Breach et S-TaLiro. En bleu les données de l'utilisateur, en rouge un signal respectant les contraintes*

propriété.

Dans ces deux boites à outils, on décrit les entrées possibles en donnant des intervalles datés. Pour générer une entrée on tirera un point dans chaque intervalle et lui assignera la date correspondante. Les outils vont ensuite transformer ces points en polynômes par interpolation[7] (figure 7). Ces polynômes sont les entrées du système Simulink à tester. Ce dernier renvoie un tableau de valeurs représentant les signaux de sortie échantillonnés. Les outils calculent alors la robustesse de la spécification (formule MITL) sur cette trace.

Si le but est de falsifier la spécification, cette robustesse servira de valeur à minimiser dans un problème d'optimisation : trouver une entrée qui viole la spécification φ sur le système SUT revient à générer une entrée $\vec{u}$ telle que $\rho(\varphi)(SUT(\vec{u}), 0) < 0$. Les deux boites à outils implémentent plusieurs algorithmes pour résoudre ce problème d'optimisation, les deux mis en avant sont Monte-Carlo [NSF+10] pour S-TaLiro et Nelder-Mead [NM65] pour Breach.

Les deux boites à outil utilisent un algorithme hors-ligne pour le calcul de robustesse, c'est-à-dire qu'on simule d'abord le système, qu'on enregistre les valeurs des signaux de sortie à intervalle régulier, puis qu'on donne ces valeurs à un algorithme qui calcule la robustesse de la formule.

> **Remarque**
> Breach utilise en fait un algorithme "semi en-ligne". Plutôt que de simuler le système en une seule fois, l'algorithme va faire des pas de simulations d'une certaine taille Δt définie par l'utilisateur puis envoyer les signaux de sorties échantillonnés calculés sur cette durée à un algorithme de calcul hors-ligne de robustesse. Cet algorithme peut soit terminer s'il a assez de données soit demander plus de valeurs, ce qui relance la simulation pendant un temps Δt, etc.

III. TRAVAIL RÉALISÉ

a. Spécification du système : l'oracle

a.1 Robustesse de formules MITL : Algorithme en ligne

La première question que je me suis posé est de savoir si l'on peut évaluer la véracité de formules MITL en ligne. Cela permettrait non seulement de faire du test en ligne, ce qui a pour avantage de terminer le plus tôt possible mais aussi de la surveillance en temps réel du système.

7. Les deux outils implémentent des interpolations de degré 0 et 1

C'est possible si on échantillonne les signaux [FP09]. J'ai implémenté un algorithme qui calcule la robustesse d'une formule d'une variante de MITL : $MITL_{[a,b]}$ [MN04] (aussi appelée clMTL dans [FP09]). La différence est que l'intervalle I de l'opérateur U_I est forcément un intervalle $[a, b]$ avec $0 \leq a < b$ et $a, b \in \mathbb{R}$. On empêche les intervalles de la forme $[a, +\infty]$ pour être sûr que l'algorithme termine sur toutes les formules.

Propriété Soient deux formules MITL φ_1 et φ_2, soit une trace $s = (t_i, v_i)_{i \in \mathbb{N}}$, soit $\forall i \in \mathbb{N}, \delta t_i = t_{i+1} - t_i$, on a

$$\begin{aligned}\llbracket \varphi_1 \; U_I \; \varphi_2 \rrbracket_{MITL}(s, t_i) = & \llbracket \varphi_1 \rrbracket_{MITL}(s, t_i) \wedge \\ & \Big(\llbracket \varphi_2 \rrbracket_{MITL}(s, t_i) \wedge 0 \in I \vee \llbracket \varphi_1 \; U_{I -_{\mathbb{R}} \delta t_i} \varphi_2 \rrbracket_{MITL}(s, t_{i+1}) \Big)\end{aligned}$$

Cette propriété sépare l'évaluation d'un until en une partie qui ne dépend que de l'instant courant et une partie qui dépend du futur. Le calcul de la robustesse d'un until peut-être fait de la même façon.

Propriété Soit $K_{\in}^{\infty}(0, I) = +\infty$ si $0 \in I$ et $-\infty$ sinon.

$$\begin{aligned}\rho(\varphi_1 \; U_I \; \varphi_2)(s, t_i) = \max \Big(\quad & \rho(\varphi_1)(s, t_i), \\ & \min \big(\max(\rho(\varphi_2)(s, t_i), K_{\in}^{\infty}(0, I)), \\ & \quad \max(\rho(\varphi_1)(s, t_i), \rho(\varphi_1 \; U_{I -_{\mathbb{R}} \delta t_i} \varphi_2)(s, t_{i+1})) \big) \Big)\end{aligned}$$

Démonstration voir annexe C

L'idée de la preuve est la même. On peut alors évaluer la partie qui dépend de l'instant courant et attendre les prochains points pour évaluer la suite.

Propriété

$$I \cap \mathbb{R}_{\geq 0} = \emptyset \Rightarrow \quad \begin{array}{l} \llbracket \varphi_1 \; U_I \; \varphi_2 \rrbracket_{MITL}(s, t) = false \\ \rho(\varphi_1 \; U_I \; \varphi_2)(s, t) = -\infty \end{array}$$

On peut donc arrêter la réécriture et rendre notre résultat final au moment où $I \cap \mathbb{R}_{\geq 0} = \emptyset$.

Il faut pouvoir encoder dans une formule les calculs déjà effectués. Pour cela j'utilise le type de données inductif Caml suivant :

```
let comp = Gt | Lt
and expr =
| NEconst  of float
| NEvar    of string
| NEcomp   of expr * comp * expr
and formula =
| Nexpr    of expr
| Nnot     of formula
| Nand     of formula * formula
| Nuntil   of formula * float * float * formula
```

Le type expr représente les formules de la forme $v \sim f$ et la formule $\top$ qui est NEconst $+\infty$. On utilise NEconst pour stocker les valeurs déjà calculées lors des étapes précédentes.

fonction derive(φ, $\vec{x}$, δt, last)
    si φ = f alors f
  sinon si φ = v ~ f alors
      si ~ = > alors $\vec{x}$(v) − f
    sinon si ~ = < alors f − $\vec{x}$(v)
  sinon si φ = ¬ φ' alors
    si φ' = f ou φ' = v ~ f alors −derive(φ', $\vec{x}$, δt, last)
    sinon si φ' = ¬ φ'' alors derive(φ'', $\vec{x}$, δt, last)
    sinon ¬ derive(φ', $\vec{x}$, δt, last)
  sinon si φ = φ_1 ∧ φ_2 alors
    derive(φ_1, $\vec{x}$, δt, last) ∧ derive(φ_2, $\vec{x}$, δt, last))
  sinon si φ = φ_1 $\mathrm{U}_{[a,b]}$ φ_2 alors
    si last alors
      derive(φ_1, $\vec{x}$, δt, last) ∧ derive(φ_2, $\vec{x}$, δt, last) ∧ $\mathrm{K}_{\in}^{\infty}(0, [a, b])$
    sinon
      derive(φ_1, $\vec{x}$, δt, last) ∧ (derive(φ_2, $\vec{x}$, δt, last) ∧ $\mathrm{K}_{\in}^{\infty}(0, [a, b])$ ∨ φ_1 $\mathrm{U}_{[a-\delta t, b-\delta t]}$ φ_2)

Figure 8 – *Algorithme d'évaluation en ligne de la robustesse d'une formule MITL. Exemple d'exécution en annexe D.*

L'algorithme (figure 8) prend en argument une formule φ, les valeurs courantes $\vec{x}$ des signaux, le temps δt qui va s'écouler avant le prochain appel de la fonction et un booléen last qui dit si c'est la dernière étape de simulation. Il renvoie à chaque étape une formule qui doit être vérifiée à l'étape suivante. Il y a en plus de ça une étape de simplification de la formule qui n'est pas explicitée. Elle sert à éviter qu'elle ne grossisse trop (à chaque étape un until produit trois termes et un autre until). Cette simplification se fait en suivant la sémantique quantitative des formules, c'est-à-dire en posant que $f_1 \wedge f_2 = \min(f_1, f_2)$ et $f_1 \vee f_2 = \max(f_1, f_2)$ pour f_1 et f_2 des réels.

Une exécution de cette fonction a une complexité linéaire en fonction de la taille de la formule. Avec une simplification triviale de la formule (évaluation des termes de la forme $f \vee f$ et $f \wedge f$ avec f un flottant), elle grossit linéairement à chaque étape : chaque until génère au plus deux nouveaux termes et un until. Lorsque deux until sont imbriqués (à gauche ou à droite), ils génèrent 3 termes et deux until par étape, lorsque trois until sont imbriqués (par exemple $(\varphi_1\ U_{I_1}\ \varphi_2)\ U_{I_2}\ (\varphi_3\ U_{I_3}\ \varphi_4))$ ou encore $(\varphi_1\ U_{I_1}\ (\varphi_2\ U_{I_2}\ (\varphi_3\ U_{I_3}\ \varphi_4))))$, ils génèrent 4 termes et trois until par étape, etc.. Pour simplifier, soit m_U le nombre d'until de la formule, cette chaîne génère $O(m_U)$ nouveaux termes à chaque étape. La complexité en temps et en espace de l'algorithme au long de la simulation sont donc de $O(nm_U)$ avec n le nombre d'appels de la fonction et m_U le nombre d'until dans la formule.

La fonction atteint un point fixe lorsque last est à vrai ou lorsque la formule après simplification n'est plus qu'un nombre. On a alors notre résultat final : la robustesse de la formule sur cette simulation.

Le résultat que l'on obtient avec cette méthode dépend directement de l'échantillonnage que l'on fait : on saute de valeur en valeur en ignorant ce qui se produit entre les deux. Avec une approche en ligne, cela veut dire qu'il faut souvent arrêter le solveur numérique pour ne pas rater de valeurs. Le problème est que cela ralentit le solveur, ça l'oblige à refaire ses estimations plus souvent.

Remarque

C'est pour cette raison que Breach utilise une approche semi en ligne, plus la durée Δt de chaque morceau de simulation est grande, plus la simulation va vite, moins le résultat est précis. Il faut trouver un bon compromis (c'est à l'utilisateur de le faire dans Breach).

a.2 Traduction d'une formule MITL en automate hybride

Pour pallier ce problème, il faut pouvoir exécuter notre oracle en parallèle avec notre système. Il faut que l'oracle soit un *observateur synchrone*. c'est-à-dire un nœud qui surveille l'état du programme sans participer au calcul.

J'ai donc essayé de traduire les formules MITL en programmes hybrides écrits en Zélus. La première difficulté est que cette logique n'est pas causale : la valeur de vérité d'une formule à l'instant t peut dépendre des valeurs qu'auront les signaux à des instants ultérieurs. Cela nous force à modifier la question que l'on se pose : à l'instant t de la simulation, on cherche à savoir si la trace que l'on a calculé jusqu'à maintenant contredit la propriété ou non. Si l'on se restreint à des formules $\text{MITL}_{[a,b]}$, on peut déterminer un instant à partir duquel les valeurs des signaux n'influent plus sur la véracité de la formule. On peut donc calculer un temps maximal de simulation à atteindre pour être sûr que la simulation en cours ne viole pas la propriété.

Par ailleurs, dans le calcul de robustesse il faut pouvoir accumuler des maximums (ou des minimums), c'est-à-dire écrire des équations de la forme $min_x = \min_t x(t)$. Une façon de calculer la valeur de min_x est d'utiliser le signe de la dérivée de x (notée derx) : si derx < 0 et x ≤ min_x, alors der min_x = derx sinon der min_x = 0.

```
der min_x = if rising then derx else 0. init x
and init rising = (derx ≥ 0.)
and present
| up(derx) on (min_x ≤ x)
| up(x −. min_x) on (derx ≥ 0.) → do rising = true done
| up(−. derx) → do rising = false done
```

Cependant, Zélus ne permet pas d'accéder à la dérivée d'une variable[8]. Cette méthode nous obligerait donc à demander à l'utilisateur de fournir ses signaux et leurs dérivées, ce n'est pas satisfaisant.

J'ai choisi de mettre ce problème de côté pour l'instant; plutôt que de calculer une robustesse, calculons la valeur de vérité de la formule. Dans ce cas là l'équation que l'on cherche à calculer est $or_x = (\exists t, x(t) = true)$. Calculer or_x revient à surveiller les discontinuités d'un signal booléen.

```
let hybrid acc_or(x) = or_x where
rec init or_x = x
and present (disc(x)) →
do or_x = (last or_x) or x done
```

disc(x) est un événement déclenché à chaque discontinuité du signal x.

Des travaux ont déjà été fait pour traduire des formules MITL en automates de büchi non déterministes [MNP06]. Le nombre d'états et d'horloges de l'automate construit est linéaire en fonction du nombre d'until dans la formule.

Cette traduction est utile pour les problèmes de vérifications de modèles. Elle n'est pas viable dans le cas du test à base de propriété car pour exécuter l'automate, il faut le déterminiser, ce qui conduit à une explosion combinatoire du nombre d'états et d'horloges.

J'ai donc cherché à simplifier la logique que j'utilise. Aucun des exemples de Breach et S-TaLiro, n'est de la forme $(\varphi_1\ U_{I_1}\ \varphi_2)\ U_{I_2}\ \varphi_3$. Par ailleurs je ne sais pas décrire intuitivement le signal spécifié par une formule de ce type.

Dans la suite on se limitera à la logique définie par :

$$
\begin{aligned}
e &:= \top \mid v \sim f \mid \neg\, \varphi \mid \varphi \wedge \varphi \\
\varphi &:= e \mid e\ U_I\ \varphi
\end{aligned}
$$

Si on se limite à ces formules, on ne peut décrire que des séquences simples : j'ai e_1 puis e_2 puis e_3, etc. (avec en plus des contraintes temporelles sur les "puis").

8. Simulink non plus

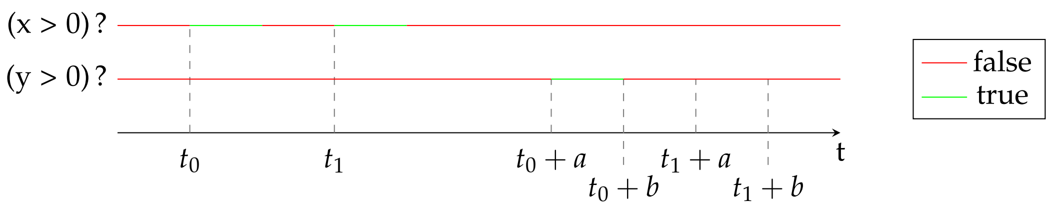


Figure 9 – *Contre-exemple pour l'approche décrite en a.2*

J'ai essayé d'écrire les oracles correspondant à cette logique en écrivant des nœuds hybrides composables reconnaissant les constructions de base ; c'est-à-dire un nœud reconnaissant la conjonction de deux formules, un autre le U_I, etc.

Exemple En utilisant des fonctions d'ordre supérieur, la formule $(x > 0 \wedge y > 0)\ U_I\ z > 0$ est reconnue par un nœud

```
let hybride oracle(inputs) = temp_until (temp_and (temp_gt x 0) (temp_gt y 0)) (temp_gt z 0) (inputs)
```

qui est une fonction qui prend en argument les valeurs des variables x, y et z et renvoie un booléen

Cependant après plusieurs tentatives je n'ai pas pu écrire de nœud hybride simple ayant la même sémantique que $\varphi_1\ U_I\ \varphi_2$. Mon idée de départ était d'écrire un automate qui vérifie que le nœud reconnaissant φ_1 renvoie *vrai* jusqu'à un certain instant $t \in I$ où le nœud reconnaissant φ_2 renvoie *vrai*.

Cet automate n'a pas la bonne sémantique : soit la formule $\Box\big(x > 0 \Rightarrow (\Diamond_{[a,b]} y > 0)\big)$ (en français "à chaque instant t où x est positif, il existe un instant $t' \in [t + a, t + b]$ tel que y est positif") et des signaux x et y tels que décrits par la figure 9. Cette formule n'est pas vérifiée par ces signaux. En utilisant l'idée précédente, l'automate va attendre jusqu'à ce que $x > 0$ (à l'instant t_0), puis il va attendre d'avoir $y > 0$ entre $t_0 + a$ et $t_0 + b$.

Il va accepter cette trace alors qu'il devrait la refuser.

En fait, le problème est que lorsque l'automate ne peut reconnaître qu'un motif à la fois. Faire en sorte qu'il en reconnaisse plusieurs en parallèle revient à écrire un automate non déterministe, ce qui nous ramène au même problème que précédemment.

a.3 Librairie d'observateurs synchrones

Code en annexe F

Le fait que la véracité d'une formule MITL dépendent du futur fait de cette logique un choix discutable pour le test en ligne. Ces problèmes existaient déjà pour les systèmes à temps discret et la logique LTL. Une solution qui a été apportée est de raisonner sur le passé des signaux plutôt que sur leur futur [HLR94].

J'ai choisi d'écrire un ensemble d'observateurs synchrones pour spécifier mes programmes. Cette approche est moins expressive que l'utilisation de formules MITL mais elle a l'avantage d'être utilisable en ligne et d'être rapide (voir Résultats comparatifs). On va cependant les confronter aux exemples écrits avec les outils Breach et S-TaLiro.

Dans cette librairie, un signal est représenté par un couple : on a d'une part un booléen qui correspond à la valeur de vérité du signal et d'autre part un événement déclenché à chaque changement de cette valeur.

Remarque
Cette représentation est redondante, on peut déduire la valeur à partir des changements (et de la valeur initiale) et les changements à partir des discontinuités de la valeur mais c'est plus pratique d'avoir les deux en permanence

Les formules de base sont des expressions de la forme $v \sim f$.

Exemple nœud traitant la formule $x > 0$.

```
let hybrid val_gt0(t, x) = r, e where
    rec init r = (x > 0.)
    and present
        up(x) → do r = true and emit e = (t, true) done
        | up(−. x) → do r = false and emit e = (t, false) done
        | (disc(x)) → do r = (x > 0.) and emit e = (t, x > 0.) done
```

Ici, en plus de surveiller les événements up(x) et up(−.x), on surveille l'événement disc(x) (discontinuités de x) car up ne traite les changements de signe que durant les phases continues.

Ces expressions peuvent être combinées à l'aide d'opérateurs logiques : les opérateurs classiques c_not, c_and et c_or et des opérateurs temporisés.

Exemple Conjonction de deux formules

```
let hybrid c_and((rA, eA), (rB, eB)) = r, e where
    rec r = rA && rB
    and present
    | eA(t, true) on (rB) | eB(t, true) on (rA) → do emit e = (t, true) done
    | eA(t, false) | eB(t, false) → do emit e = (t, false) done
```

Ce nœud émet un signal e à vrai lorsque l'expression rA && rB devient vraie, et il émet un signal à faux lorsque l'expression devient fausse.

Les opérateurs temporisés sont inspirés de la version passée de la logique MITL. Dans cette version de MITL, plutôt que de décrire le futur d'un signal, on essaye de décrire son passé : l'opérateur "jusqu'à" (until) devient "depuis" (since, S). Intuitivement, la sémantique de $\varphi_1\ S\ \varphi_2$ est : φ_1 est vraie depuis que φ_2 a été vérifiée.

Dans cette librairie, on implémente une version plus faible de l'opérateur since :

- φ_1 since_first φ_2 : φ_1 est vérifiée depuis la dernière fois que φ_2 a été vérifiée (passage de faux à vrai)
- φ_1 since_last φ_2 : φ_1 est vérifiée depuis la première fois que φ_2 a été vérifiée (passage de faux à vrai)

On implémente aussi des variantes de l'opérateur since : "toujours" (always, $\Box$) et "nécessairement" (eventually, $\Diamond$) :

- $\Box\ \varphi$: φ a toujours été vérifiée depuis le début de la simulation
- $\Diamond\ \varphi$: φ a été vérifiée au moins une fois depuis le début de la simulation

Exemples en annexe F.

b. Génération d'entrées

Code en annexe E

J'ai choisi une approche modulaire basée sur les idées de Lutin [RRJ08]. C'est un outil écrit pour Lustre pour faire du test automatique. On y décrit des contraintes linéaires entre les variables (entrées et sorties) du programme à tester puis un solveur de contraintes essaye de trouver des valeurs qui cassent ces contraintes. J'ai repris dans mon travail la façon qu'a Lutin de décrire ses contraintes : il définit des contraintes élémentaires et les combine avec les opérateurs fby (séquence), loop (boucle) et | (choix aléatoire).

J'ai aussi introduit une rétroaction du système sur la génération d'entrée : le système à tester peut influencer son environnement. Cela peut être utile : reprenons l'exemple de la bouilloire et plus particulièrement du nœud relay. Ce nœud prend en entrée la température actuelle de l'eau et les limites de températures et choisit d'éteindre la bouilloire ou de l'allumer. On doit entre autre générer la température actuelle de l'eau si on veut tester ce nœud, et l'évolution de cette température dépend du résultat du nœud relay.

Dans cette librairie, on appelle trace une fonction qui étant donnée un ensemble de paramètres et une valeur initiale renvoie un signal continu. L'ensemble de paramètre sert entre autre à fournir les sorties du système aux fonctions de générations tandis que la valeur initiale est la valeur initiale du résultat.

Remarque
Ce n'est pas tout à fait vrai, certaines traces n'ont pas besoin de valeur initiale, elles ignorent l'argument. C'est par exemple le cas de la trace constante. Cette valeur initiale servira surtout à obtenir des fonctions continues lorsqu'on construira des séquences de traces.

Générer des entrées revient donc à générer des traces.

Exemple Fonction qui génère une trace constante : const(1., 2.) est la trace constante dont la valeur est dans $[1, 2]$. pick_float(f1, f2) est une fonction combinatoire qui tire un nombre aléatoire entre f1 et f2.

```
let hybrid const(i1, i2)(params, initial) = res where
  init res = pick_float(i1, i2)
```

Pour combiner ces traces, on a besoin d'événements. Ce sont des fonctions qui prennent les même entrées qu'une trace et émettent un signal (que l'on peut récupérer avec la primitive present en Zélus). Les événements les plus simples sont les horizons : ils sont déclenchés après un certain temps.

Exemple Fonction qui crée un horizon. horizon(2) est un événement qui sera déclenché après 2 unités de temps.

```
let hybrid horizon(h)(params, initial) = e where
  rec der t = 1. init 0.
  and present up(t -. h) -> do emit e = () done
```

On définit ensuite des combinateurs de traces ; c'est-à-dire des nœuds qui construisent des traces à partir d'autres traces (exemples annexe E). La librairie implémente les fonctions suivantes :

- La séquence t_fby : elle prend deux traces t_1 et t_2 et un événement e en argument et génère une nouvelle trace t. t se comporte comme t_1 jusqu'à l'occurrence de e puis se comporte comme t_2. La valeur initiale donnée à t_1 est la valeur initiale donnée à t puis la valeur initiale donnée à t_2 est la valeur de t_1 au moment où e est déclenché.
- La boucle t_loop : elle prend une trace t_0 et un événement e en argument et renvoie une trace t. La trace créée est une séquence infinie de la trace d'entrée réinitialisée à chaque occurrence de l'événement. La valeur initiale donnée à t_0 est initialement la valeur initiale donnée à t puis à chaque occurrence de e la valeur de t à cet instant.
- Le choix t_switch : elle prend une condition et deux traces en argument et renvoie une trace. La condition est une fonction qui prend la valeur courante de la sortie en argument et renvoie un booléen. Ce booléen sert à choisir l'une des deux traces en entrée comme valeur de sortie.

IV. Résultats comparatifs

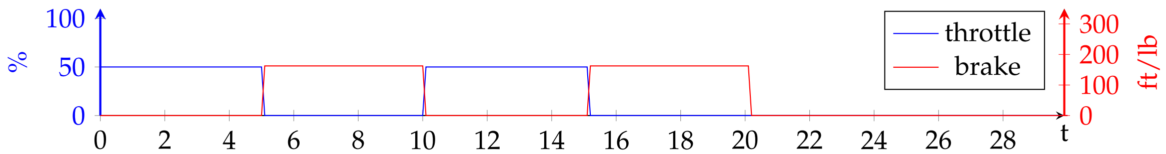
Pour comparer les librairies avec les outils déjà existants, on va utiliser l'exemple de transmission automatique (Un exemple hybride : Système de transmission automatique). Pour rappel, ce système prend une accélération et un couple de freinage en entrée et calcule la vitesse, le rapport de transmission et la vitesse de rotation moteur du véhicule.

a. Génération d'entrées

Breach et S-TaLiro proposent des outils de génération d'entrées similaires. On va donc utiliser Breach pour faire les comparaisons.

Le type d'entrées définis par Breach sur cet exemple est composé de 4 phases qui durent chacune entre 0.1 et 10 secondes suivies d'une phase qui dure indéfiniment. Durant la première et la troisième phase, l'accélération a une valeur constante entre 0 et 100% tandis que le couple de freinage vaut 0 et durant la deuxième et la quatrième phase, le couple de freinage a une valeur entre 0 et 350 $ft.lb^{-1}$ tandis que l'accélération vaut 0. Les deux entrées sont nulles durant la cinquième phase.

Exemple Exemple de signaux produits par Breach



Tout d'abord, les deux entrées ne sont pas indépendantes : l'une est nulle lorsque l'autre ne l'est pas durant les 4 premières phases. Il faut donc générer les deux entrées ensemble :

```
let hybrid const2((i11, i12), (i21, i22))(params, initial) = res1, res2 where
  res1 = const(i11, i12)(params, initial)
  and res2 = const(i21, i22)(params, initial)
```

Ensuite, nous avons besoin de tirer un horizon aléatoirement dans un intervalle :

```
let hybrid rand_horizon(h1, h2)(_) = e where
  rec der t = 1. init 0.
  and h = const(h1, h2)(0., 0.)
  and present up(t -. h) -> do emit e = () done
```

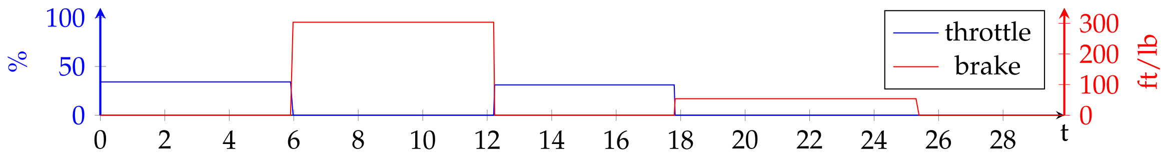
Remarque

Ici on ne peut pas utiliser le noeud horizon défini plus tôt car en Zélus, les fonctions d'ordre supérieur attendent des arguments statiques en entrée. Ici les arguments statiques sont les bornes de l'intervalle dans lequel on tire notre horizon.

Enfin, on combine ces deux noeuds à l'aide du t_fby pour créer nos deux entrées.

```
let hybrid generate() =
  t_fby(const2((0., 100.), (0., 0.)),
    rand_horizon(0., 10.),
    t_fby(const2((0., 0.), (0., 350.)),
      rand_horizon(0., 10.),
      t_fby(const2((0., 100.), (0., 0.)),
        rand_horizon(0., 10.),
        t_fby(const2((0., 0.), (0., 350.)),
          rand_horizon(0., 10.),
          const2((0., 0.), (0., 0.))))))(0., (0., 0.))
```

Exemple Exemple de signaux produits par generate



b. Spécification de systèmes

Ci-dessous certaines propriétés utilisées par Breach pour spécifier le système de transmission automatique et leur traduction en observateurs synchrones à l'aide de la librairie.

— La vitesse n'est jamais inférieure à 30 mph lorsqu'on est en 3e vitesse

$\hookrightarrow$ MITL : $\Box\Big(\neg\ (gear(t) = 3 \land speed(t) < 30)\Big)$

$\hookrightarrow$ Zélus :

```
let hybrid never_gear3_and_speed_low(t, throttle, brake_torque, rpm, gear, speed) =
let speed_low = val_lt(v_low)(t, speed) in
let gear3 = c_and(val_in(2.5, 3.5)(t, gear)) in
let gear3_and_speed_low = c_and(gear3, speed_low) in
always (c_not gear3_and_speed_low)
```

— Lorsqu'on passe en 2e vitesse, on y reste pendant au moins 1 seconde

$\hookrightarrow$ MITL : $\Box\Big(\Big((\neg\ gear(t) = 2) \land \Diamond_{[0.01,0.02]} gear(t) = 2\Big) \Rightarrow \Box_{[0,1]} gear(t) = 2\Big)$

Les observateurs since_first et since_last ne peuvent pas reconnaître ce genre de propriétés. J'ai donc écrit un nouvel observateur since_last_for(dT)(t, φ_1, φ_2) : φ_1 été vérifiée la dernière fois que φ_2 a été vérifiée (passage de faux à vrai) et l'est resté pendant au moins dT secondes.

```
let hybrid since_last_for(dT)(t, (rA, eA), (rB, eB)) = r, e where
rec init h = 0. and init r = true
and present
| eB(t0, true) on (rA) -> do r = true and next h = t0 +. dT done
| eA(t0, false) on (t < h) -> do r = false and emit e = (t0, false) and next h = 0. done
| up(t -. h) -> do r = true and emit e = (h, true) done
```

$\hookrightarrow$ Zélus :

```
let hybrid alw_stay2_for_t1(t, throttle, brake_torque, rpm, gear, speed) =
let gear2 = val_in(1.5, 2.5)(t, gear) in
since_last_for(1.)(t, gear2, gear2)
```

Voir annexe G pour la traduction de toutes les propriétés de l'exemple page 8 en observateurs Zélus.

c. Test de systèmes hybrides

Les librairies ne sont pas capables de calculer une robustesse, le seul moyen qu'on a de falsifier une propriété est d'exécuter le système (Environnement, Système, Oracle) jusqu'à trouver un contre-exemple.

On cherche à falsifier le système de transmission automatique avec la génération d'entrées décrite en a et la propriété never_gear3_and_speed_low décrite en b.

On exécute ce problème de falsification 100 fois avec l'implémentation Zélus et les implémentations Breach et S-TaLiro en s'arrêtant dès que l'on trouve un contre-exemple (table 1). Les résultats sont très différents : le test probabiliste est plus lent sur cet exemple. Cela peut s'expliquer par le fait que l'on teste une égalité dans notre propriété : $\Box\Big(\neg\ (gear(t) = 3 \land speed(t) < 30)\Big)$. La robustesse d'une égalité est toujours de 0 donc les algorithmes d'optimisation qu'utilisent Breach et S-TaLiro ne sont pas guidés lors de leur recherche.

		never_gear3_and_speed_low			vmaxmin		
		Zélus	Breach	S-TaLiro	Zélus	Breach	S-TaLiro
Nombre de falsifications		100	100	100	100	100	100
Nombre d'exécutions	min	1	230	1	1	130	1
	moy.	8.78.	257.5.	235.8.	3.02	140	48.5
	max	38	270	1000	14	150	420
Temps moyen d'exécution (secondes)		0.950	95.5	53.9	0.422	46.9	24.3

Table 1 – *Falsification des propriétés* never_gear3_and_speed_low *et* vmaxmin *:nombre de falsifications du système effectuées, nombre d'exécutions par falsification et temps moyen d'exécution par falsification.*

On recommence avec une autre propriété vmaxmin (table 1) : Pendant les 25 premières secondes, la vitesse est supérieure à 30 mph, pendant les 25 secondes suivantes, la vitesse est inférieure à 150 mph (MITL : $\Big((\square_{[0,25]} speed(t) < 150) \wedge (\square_{[25,50]} speed(t) > 30)\Big)$, code de l'observateur en annexe G).

Notons que le nombre d'exécutions effectuées par S-TaLiro lors d'une falsification est très variable : 1 et 2 au minimum, 943 et 1000 au maximum [9] sur la première propriété : le candidat initial a un impact important sur la durée de la falsification. Ce choix semble moins important dans le cas de Breach ; le nombre d'exécutions par falsification est plus resserré.

Conclusion

Ces résultats sont à prendre avec des pincettes. Il faudrait faire plus d'expériences sur des systèmes plus complexes pour avoir des résultats plus significatifs.

L'utilisation d'observateurs synchrones hybrides pour spécifier les systèmes semble toutefois être une approche viable pour faire du test en ligne : elle est suffisamment expressive pour être utilisée à la place de MITL sur un exemple concret.

Il faudrait cependant définir plus précisément ce que l'on cherche à décrire (peut-être un sous-ensemble de MITL), prouver que notre ensemble d'observateurs est suffisant.

Par ailleurs, je n'ai pas eu le temps de traiter une question primordiale : comment générer de petits contre-exemples. Dans l'exemple que j'ai utilisé ici les signaux que l'on a généré avaient une forme simple, mais dans un cas plus général où les signaux sont moins contraints, il faut pouvoir définir ce qu'est un petit contre-exemple.

Une autre question intéressante est de savoir si l'on peut écrire des observateurs qui calculent une robustesse plutôt qu'une valeur de vérité comme le font les algorithmes hors ligne de Breach et S-TaLiro. On pourrait alors chercher un contre exemple qui rendrait le signal très faux très longtemps : dans le cas de la bouilloire ce n'est pas grave si la température de l'eau dépasse légèrement la limite, par contre on aimerait trouver un scénario où la température diverge (si le modèle permet un tel scénario).

Enfin, on a traité nos systèmes comme des boites noires. Ça peut être intéressant de savoir dans quelle mesure on peut exploiter le code de la fonction et la spécification que l'on cherche à casser pour générer des entrées intéressantes.

9. S-TaLiro interrompt automatiquement la falsification après 1000 essais

Références

[AFH96] Rajeev Alur, Tomás Feder, and Thomas A. Henzinger. The benefits of relaxing punctuality. *J. ACM*, 43(1) :116–146, January 1996.

[ALFS11] Yashwanth Annpureddy, Che Liu, Georgios Fainekos, and Sriram Sankaranarayanan. S-taliro : A tool for temporal logic falsification for hybrid systems. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 254–257, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[BBBK16] Benoit Barbot, Nicolas Basset, Marc Beunardeau, and Marta Kwiatkowska. Uniform sampling for timed automata with application to language inclusion measurement. In *Proc. 13th International Conference on Quantitative Evaluation of SysTems (QEST'16)*, volume 9826 of *LNCS*, pages 175–190. Springer, 2016.

[BP13] Timothy Bourke and Marc Pouzet. Zélus : A Synchronous Language with ODEs. In Calin Belta and Franjo Ivančić, editors, *HSCC - 16th International Conference on Hybrid systems : computation and control*, Proceedings of the 16th International Conference on Hybrid systems : computation and control, pages 113–118, Philadelphia, United States, April 2013. Calin Belta and Franjo Ivančić, ACM.

[CH00] Koen Claessen and John Hughes. Quickcheck : A lightweight tool for random testing of haskell programs. 46, 01 2000.

[CPP17] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. Scade 6 : A Formal Language for Embedded Critical Software Development. In *TASE 2017 - 11th International Symposium on Theoretical Aspects of Software Engineering*, pages 1–10, Nice, France, September 2017.

[DM10] Alexandre Donzé and Oded Maler. Robust satisfaction of temporal logic over real-valued signals. In *Proceedings of the 8th International Conference on Formal Modeling and Analysis of Timed Systems*, FORMATS'10, pages 92–106, Berlin, Heidelberg, 2010. Springer-Verlag.

[Don10] Alexandre Donzé. Breach, a toolbox for verification and parameter synthesis of hybrid systems. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, pages 167–170, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[FP09] Georgios E. Fainekos and George J. Pappas. Robustness of temporal logic specifications for continuous-time signals. *Theoretical Computer Science*, 410(42) :4262 – 4291, 2009.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9) :1305–1320, Sep 1991.

[HLR94] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. Synchronous observers and the verification of reactive systems. In *Proceedings of the Third International Conference on Methodology and Software Technology : Algebraic Methodology and Software Technology*, AMAST '93, pages 83–96, Berlin, Heidelberg, 1994. Springer-Verlag.

[Koy90] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4) :255–299, Nov 1990.

[MN04] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In Yassine Lakhnech and Sergio Yovine, editors, *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 152–166, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[MNBB16] Reza Matinnejad, Shiva Nejati, Lionel C. Briand, and Thomas Bruckmann. Automated test suite generation for time-continuous simulink models. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 595–606, New York, NY, USA, 2016. ACM.

[MNP06] Oded Maler, Dejan Nickovic, and Amir Pnueli. From mitl to timed automata. In Eugene Asarin and Patricia Bouyer, editors, *Formal Modeling and Analysis of Timed Systems*, pages 274–289, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[NM65] J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4) :308–313, 1965.

[NSF+10] Truong Nghiem, Sriram Sankaranarayanan, Georgios Fainekos, Franjo Ivancić, Aarti Gupta, and George J. Pappas. Monte-carlo techniques for falsification of temporal properties of non-linear hybrid systems. In *Proceedings of the 13th ACM International Conference on Hybrid Systems : Computation and Control*, HSCC '10, pages 211–220, New York, NY, USA, 2010. ACM.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[RRJ08] Pascal Raymond, Yvan Roux, and Erwan Jahier. Lutin : a language for specifying and executing reactive scenarios. *EURASIP Journal on Embedded Systems*, 2008, 2008. http ://jes.eurasipjournals.com/content/2008/1/753821.

Annexes

A. MODÈLE DE TRANSMISSION AUTOMATIQUE

Modèle physique

Ce système a deux entrées : l'accélération d'entrée *Throttle* (en %) et le couple de freinage *Brake_torque* (en $ft.lb^{-1}$). On peut le décomposer en plusieurs modules, chaque module est défini par un système d'équations. Le système final est l'union de tous ces modules.

— Module moteur

$$\begin{cases} I_{ei}\dot{N}_e &= T_e - T_i \\ N_{e0} &= 1000 \end{cases}$$

avec

$T_e = f_1(Throttle, N_e)$	couple (en $ft.lb^{-1}$) de l'arbre moteur
f_1	fonction définie par une table de correspondance (figure A.1)
N_e	vitesse du moteur (en $tr.min^{-1}$), contrainte à être dans l'intervalle $[600, 6000]$
T_i	le couple (en $ft.lb^{-1}$) de la turbine
$I_{ei} = 0.02199$	moment d'inertie (en $lbf.ft.s^2$) de l'arbre moteur et de la turbine

— Module de transmission : composé de 2 modules

— Module de conversion de couple

$$\begin{cases} T_i &= \frac{N_e^2}{K^2} \\ T_t &= R_{TQ}T_i \\ K &= f_2(\frac{N_i}{N_e}) \\ R_{TQ} &= f_3(\frac{N_i}{N_e}) \end{cases}$$

avec

f_2, f_3	fonctions définies par des tables de correspondance (figure A.2)		
K	facteur K	R_{TQ}	rapport des couples
T_i	couple (en $ft.lb^{-1}$) de l'arbre moteur	T_t	couple (en $ft.lb^{-1}$) de la turbine
N_i	vitesse de rotation (en $tr.min^{-1}$) de l'arbre moteur		

— Module de rapport de transmission

$$\begin{cases} R_{TR} &= f_4(gear) \\ T_{out} &= R_{TR}T_t \\ N_i &= R_{TR}N_{out} \end{cases}$$

avec

$gear$	le rapport de vitesse		
f_4	fonction définie par une table de correspondance (figure A.3)		
N_{out}	vitesse de rotation (en $tr.min^{-1}$) de l'arbre de sortie		
R_{TR}	rapport de transmission	T_{out}	couple (en $ft.lb^{-1}$) de l'arbre de sortie

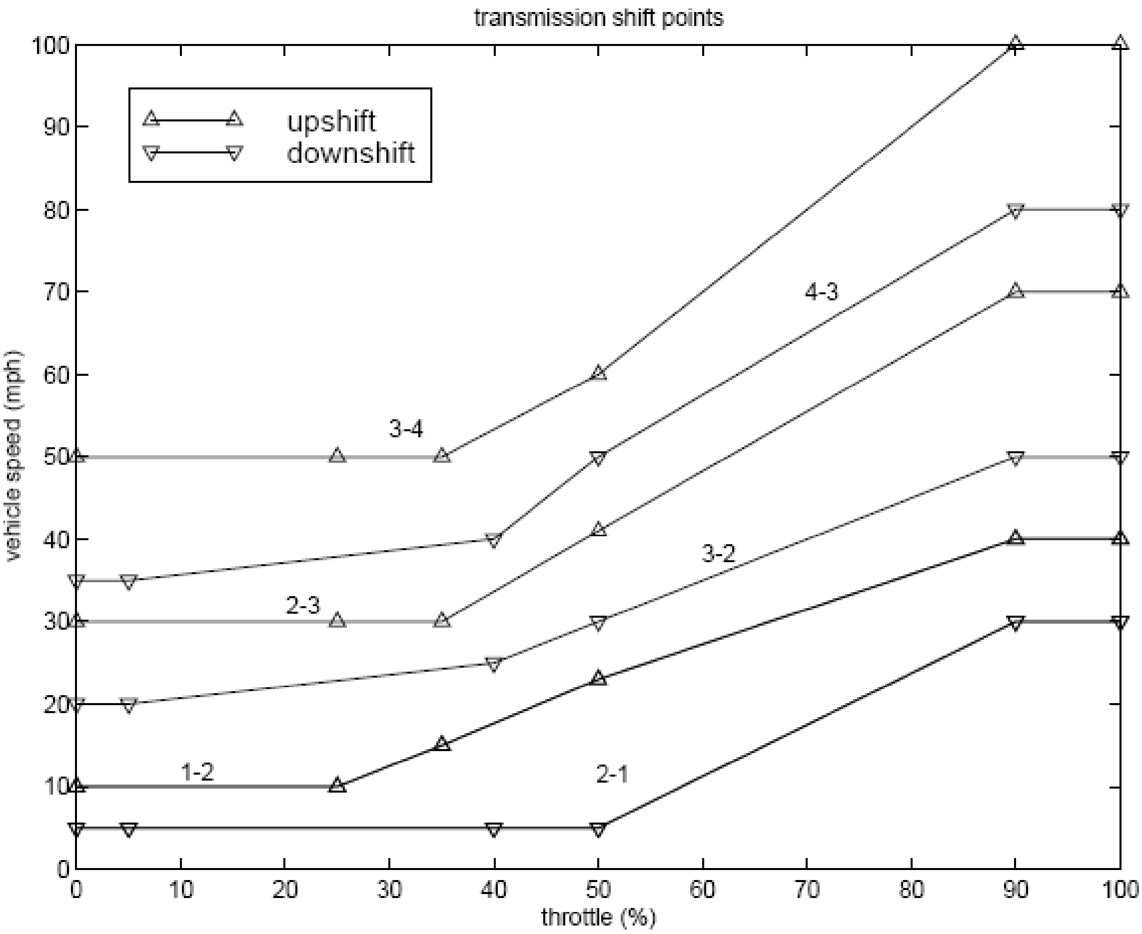
— Module véhicule

$$\begin{cases} I_v \dot{N}_w &= R_{fd}(T_{out} - T_{load}) \\ N_{w0} &= 0 \\ T_{load} &= sgn(mph)(R_{load0} + R_{load2} mph^2 + Brake\_Torque) \\ mph &= 2\pi R_w N_w \end{cases}$$

avec

N_w	vitesse de rotation (en $tr.min^{-1}$) des roues
mph	vitesse linéaire (en $ft.min^{-1}$) de la voiture
T_{load}	couple (en $ft.lb^{-1}$) de charge
$R_w = 1.000$	rayon des roues (en ft)
$I_v = 12.09$	moment d'inertie (en $lbf.ft.s^2$) du véhicule
$R_{fd} = 3.230$	rapport de transmission final
$R_{load0} = 40.00$	coefficient de friction
$R_{load2} = 0.02000$	coefficient de traînée aérodynamique

— Module de contrôle du rapport de vitesse : Le rapport de vitesse *gear* est modifié en fonction de l'accélération *Throttle* et de la vitesse du véhicule *mph*. Lorsque le couple (*Throttle*, *mph*) franchit un seuil haut (upshift), on passe la vitesse supérieure, lorsqu'il franchit un seuil bas (downshift), on passe la vitesse inférieure. Les seuils sont tracés ci-dessous.



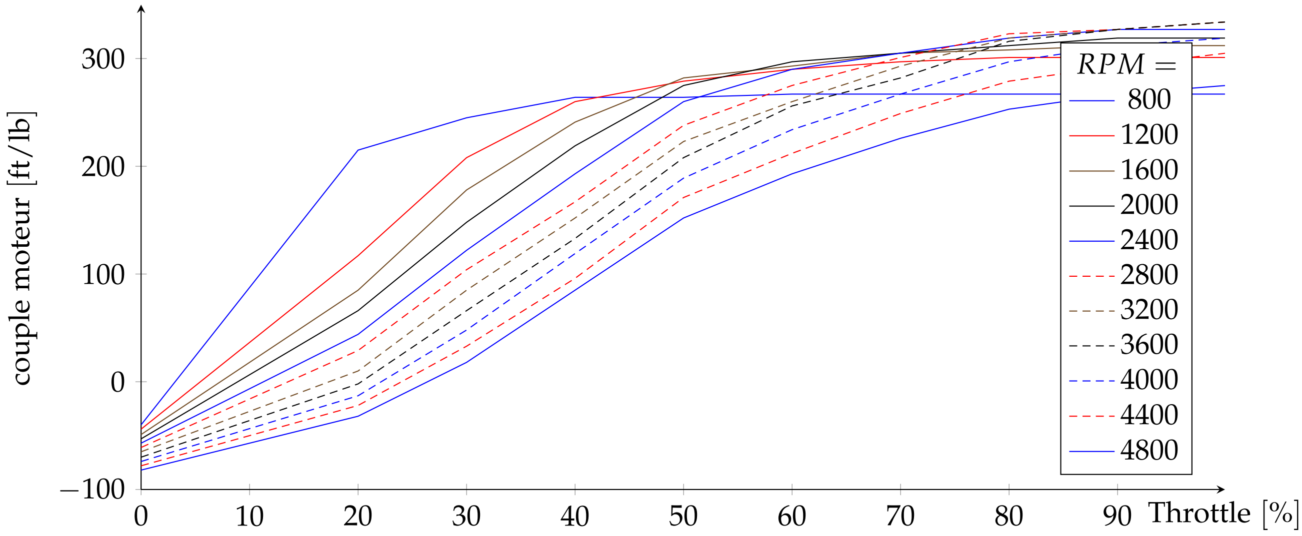


Figure A.1 – *Table de correspondance décrivant* f_1

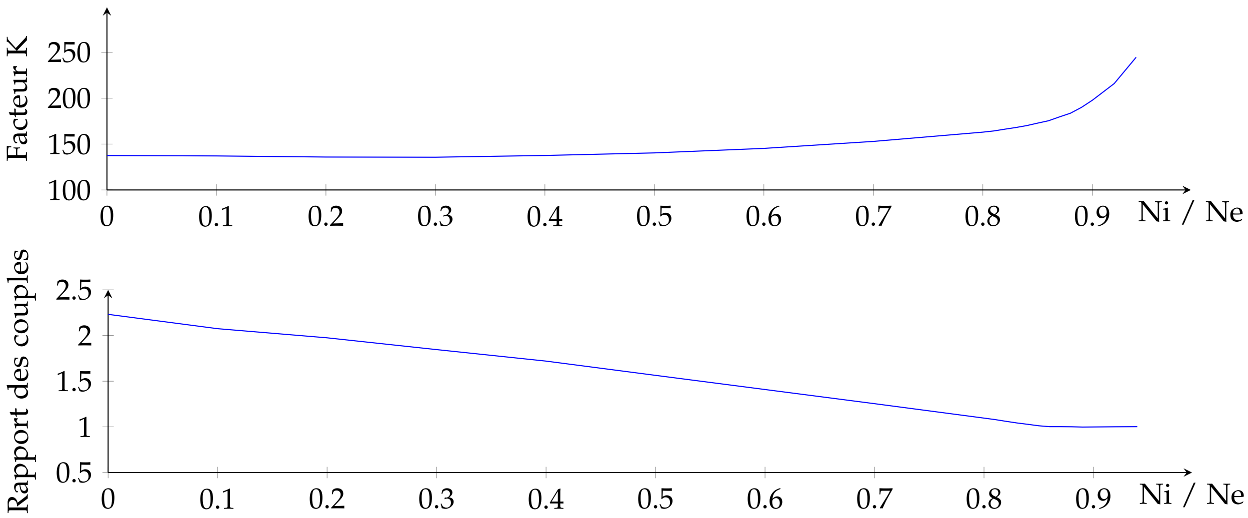


Figure A.2 – *Table de correspondance décrivant* f_2 *et* f_3

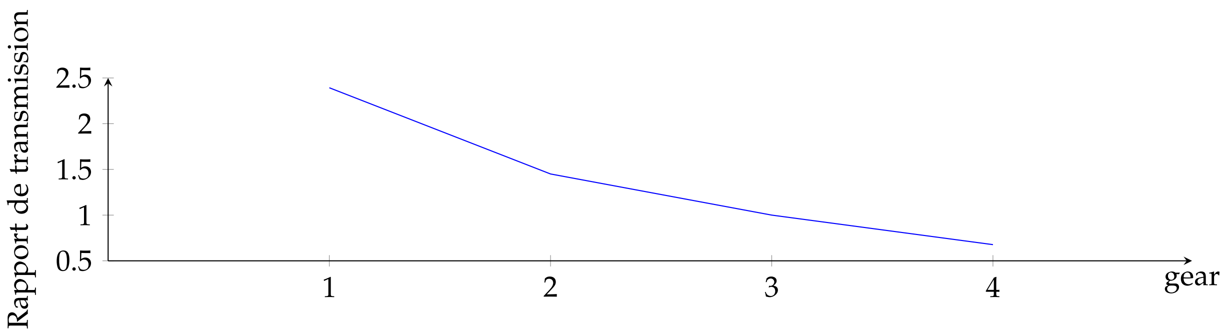
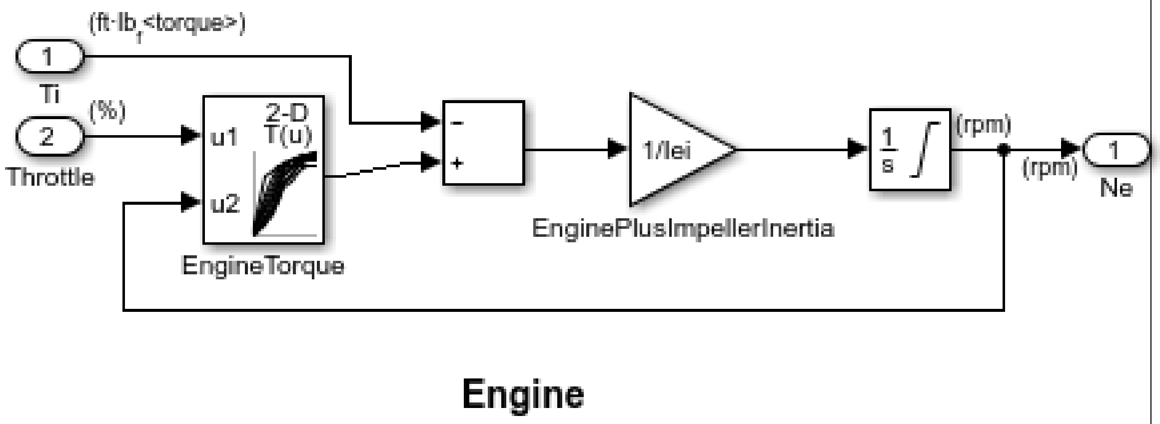
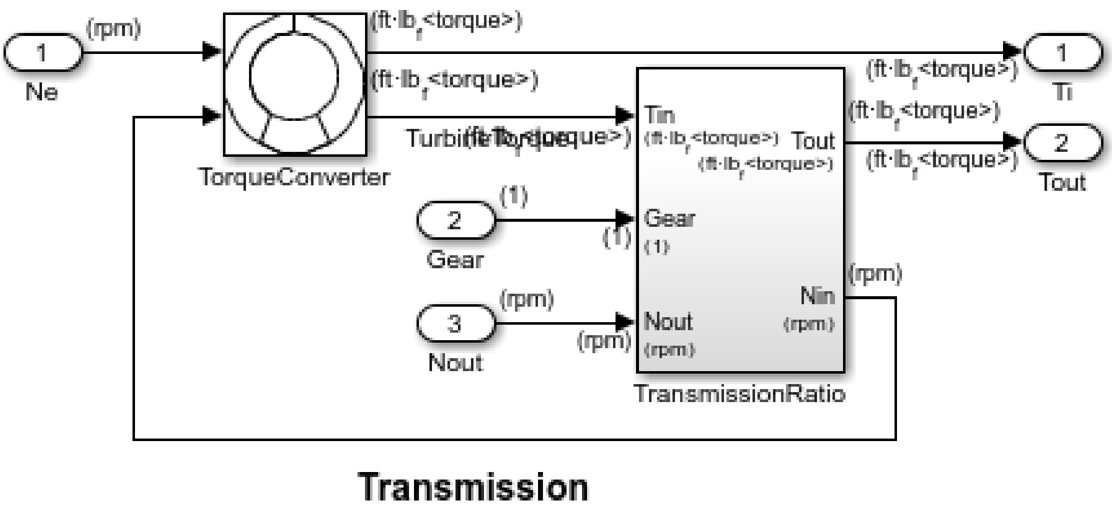


Figure A.3 – *Table de correspondance décrivant* f_4

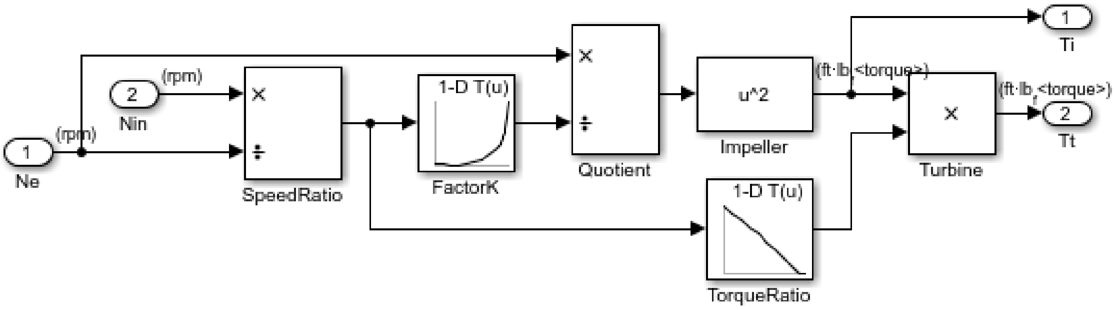
Implémentations Simulink et Zélus




```
let hybrid engine(ti, throttle) = rpm where
  rec te = lookup_engine(throttle, rpm)
  and drpm = te -. ti
  and rpm = integrate_and_saturate (inertia *. drpm, 1000., 600.,6000.)
```

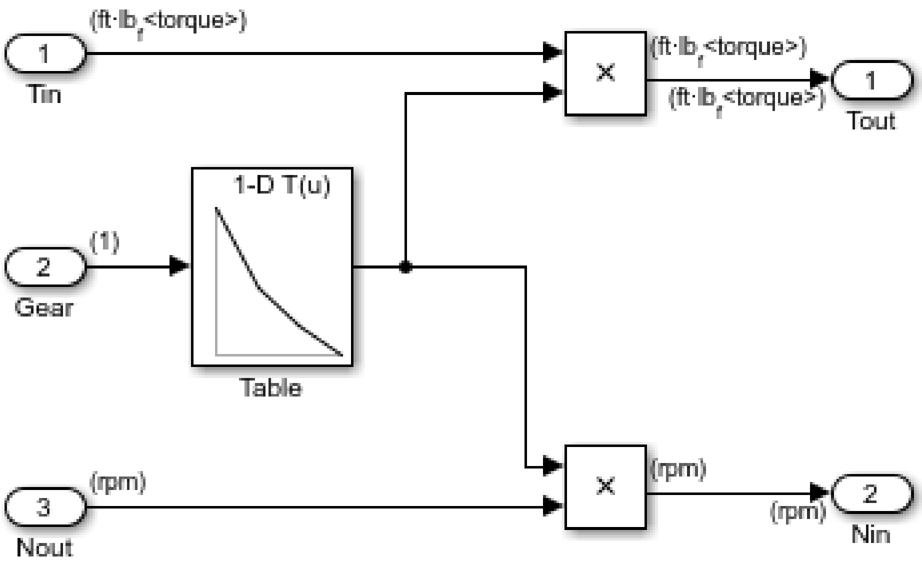



```
let transmission(ne, gear, nout) = (ti, tout) where
  rec ti, turbine_torque = torque_converter(ne, nin)
  and tout, nin = transmissionratio(turbine_torque, gear, nout)
```



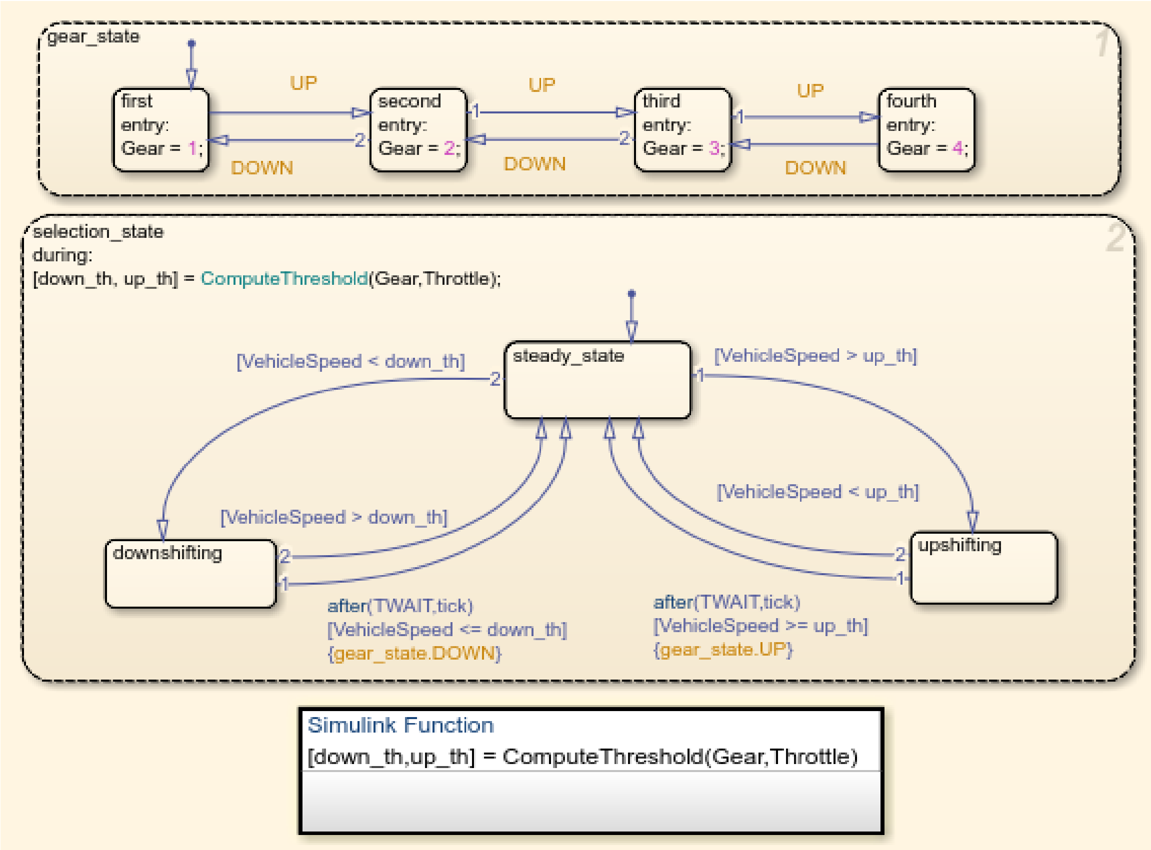
Torque Converter

```
let torque_converter(ne, nin) = (ti, tt) where
  rec speedratio = nin /. ne
  and factorK = lookup_factork speedratio
  and torqueratio = lookup_torqueratio speedratio
  and quotient = ne /. factorK
  and impeller = quotient *. quotient
  and turbine = impeller *. torqueratio
  and ti = impeller and tt = turbine
```

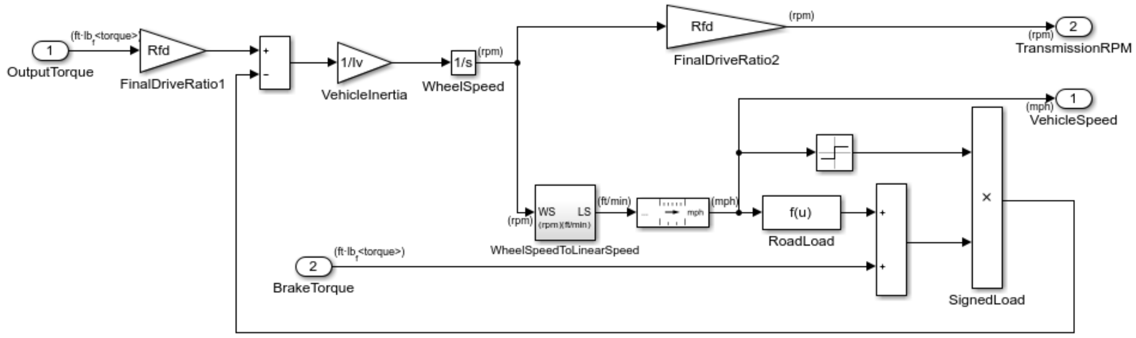


Transmission Gear Ratio

```
let transmissionratio(tin, gear, nout) = (tout, nin) where
  rec trans_ratio = lookup_gear gear
  and tout = tin *. trans_ratio
  and nin  = nout *. trans_ratio
```




```
let hybrid shiftlogic(speed, throttle) = gear where
  rec up_th = lookup_interpup(throttle, gear)
  and dn_th = lookup_interpdn(throttle, gear)

  and init gear = 1.
  and automaton
  | First  -> do
          until shift(UP) then do next gear = 2.
          in Second

  | Second -> do
          until shift(DOWN) then do next gear = 1.
          in First
          else shift(UP) then do next gear = 3.
          in Third

  | Third  -> do
          until shift(DOWN) then do next gear = 2.
          in Second
          else shift(UP) then do next gear = 4.
          in Fourth

  | Fourth -> do
          until shift(DOWN) then do next gear = 3.
          in Third

and automaton
| SteadyState  -> do
          until
            up(dn_th -. speed)
          | (init) on (speed < dn_th)
          then DownShifting

          else
            up(speed -. up_th)
          | (init) on (speed > up_th)
          then UpShifting

| DownShifting -> local t in do der t = 1. init 0.
          until
          up(speed -. dn_th) then SteadyState

          else
          up(t -. twait) then
          do emit shift = DOWN
          in SteadyState

| UpShifting   -> local t in do der t = 1. init 0.
          until
          up(up_th -. speed) then SteadyState

          else
          up(t -. twait) then
          do emit shift = UP
          in SteadyState
```

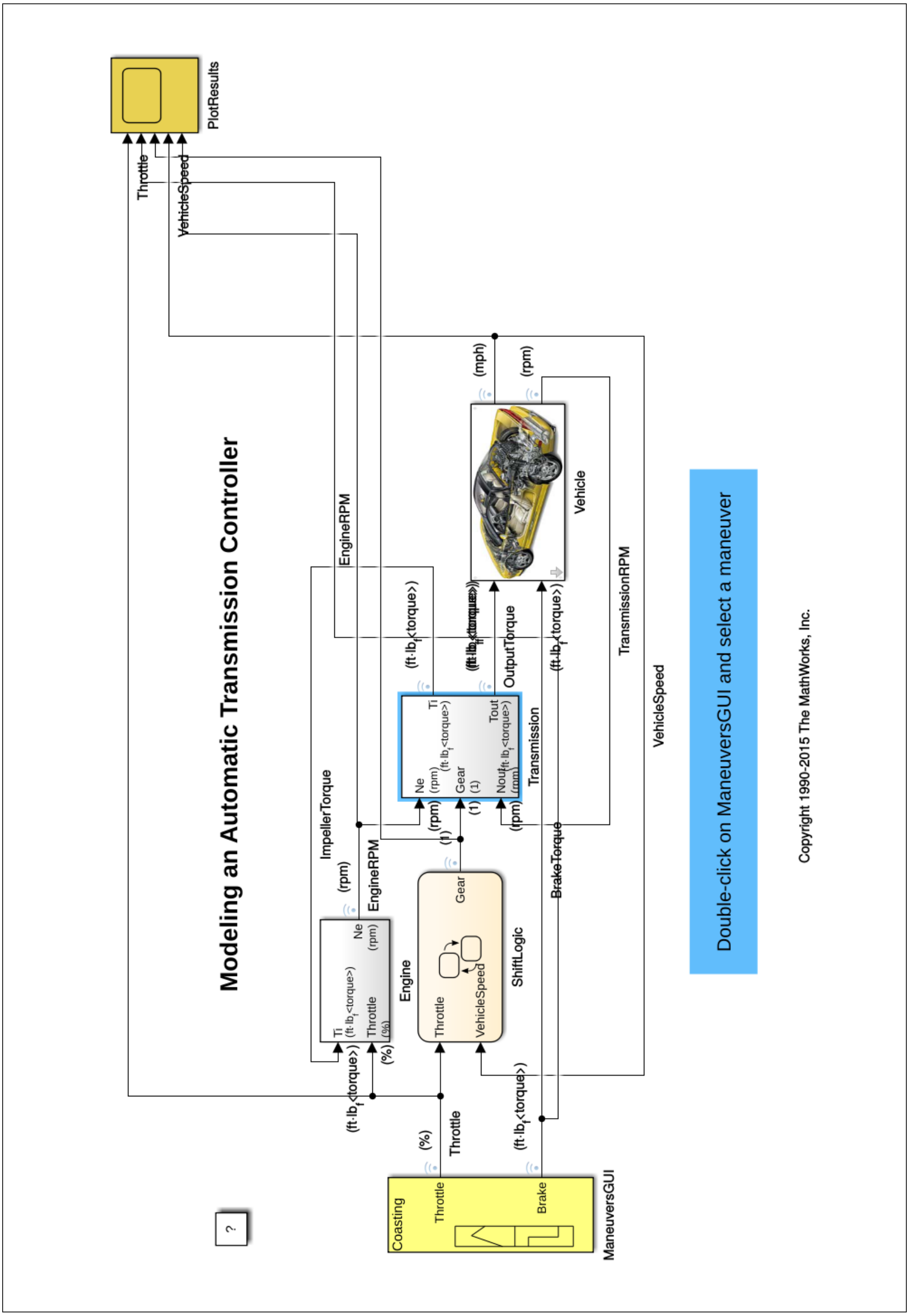


Vehicle

```
let hybrid vehicle(out_torque, brake_torque) = (speed, trans_rpm) where
rec final_drive_ratio = p_final_drive_ratio *. out_torque
and aux1 = final_drive_ratio -. signed_load
and vehicle_inertia = iv_inv *. aux1
and der wheel_speed = vehicle_inertia
    init p_initial_transmission_out_speed
and linear_speed = l_speed_coef *. wheel_speed
and mph = p_mph *. linear_speed
and brake = brake_torque +. road_load
and road_load = p_drag_friction +.
        p_aerodynamic_drag *.
        mph *. mph
and sgn = Basics.bad_sgn mph
and signed_load = sgn *. brake
and trans_rpm = p_final_drive_ratio *.
        wheel_speed
and speed = mph
```

```
let hybrid autotrans(throttle, brake_torque) = (rpm, gear, speed) where
  rec rpm = engine(ti, throttle)
  and gear = shiftlogic(speed, throttle)
  and ti, out_torque = transmission(rpm, gear, trans_rpm)
  and speed, trans_rpm = vehicle(out_torque, brake_torque)
```

Modèle final Zélus

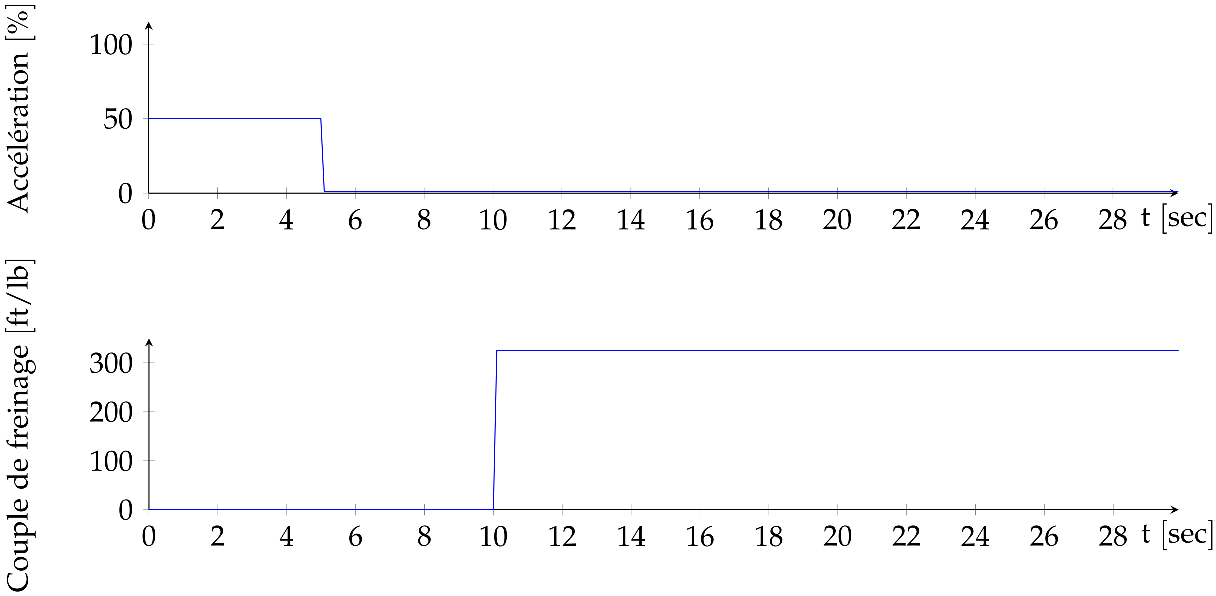


Modèle final Simulink

B. Simulations du système de transmission automatique

Scénario freinage sec

Entrées



Sorties

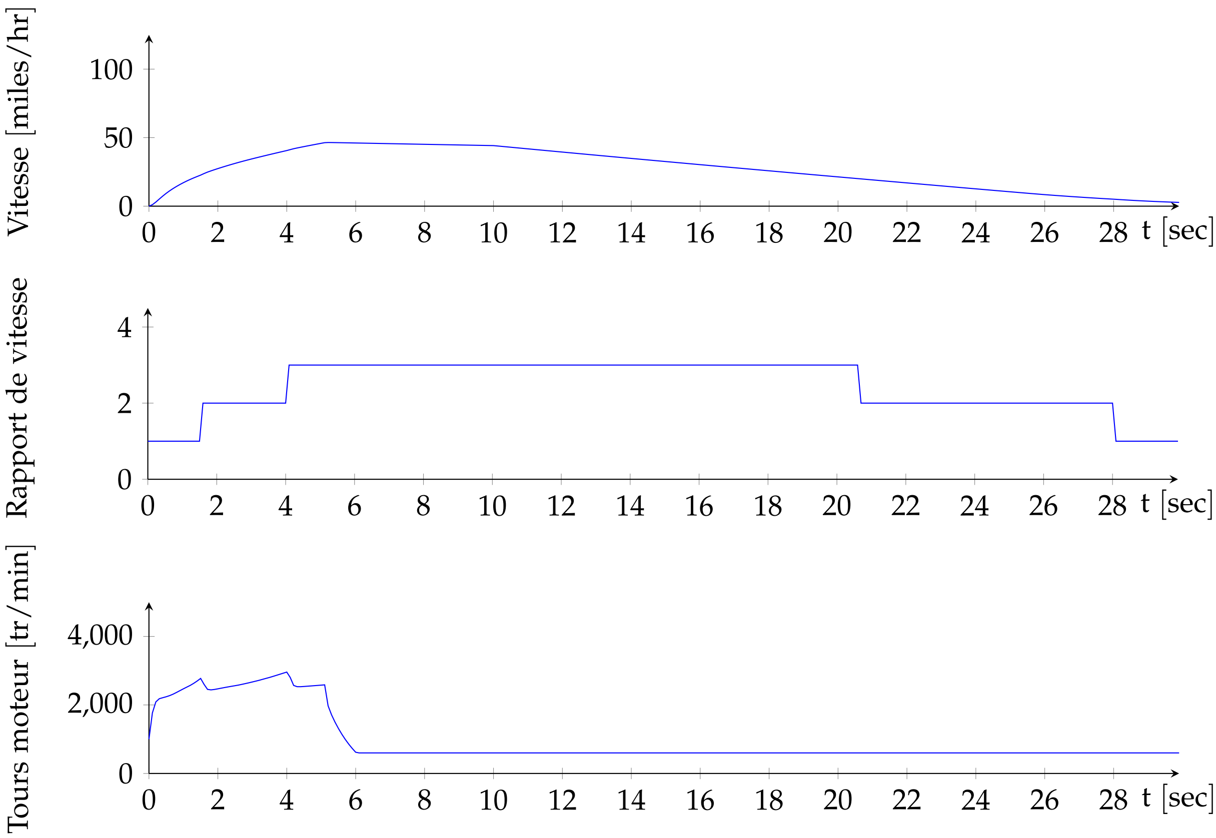
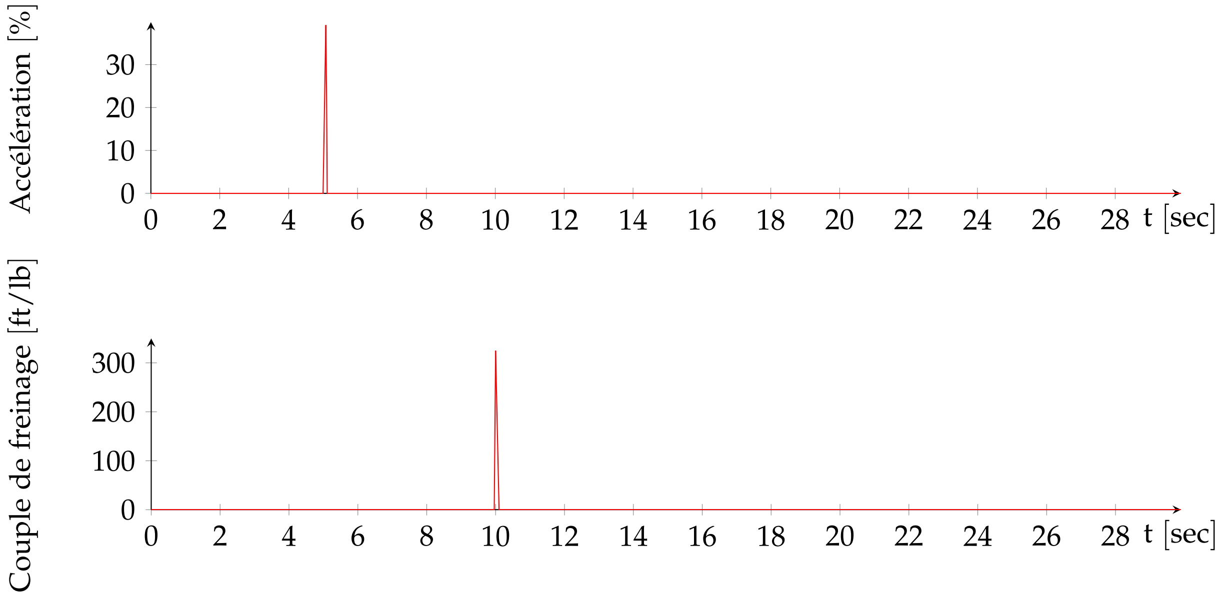


Figure B.1 – *Simulation du système (implémentation Zélus)*

Scénario freinage sec

Entrées



Sorties

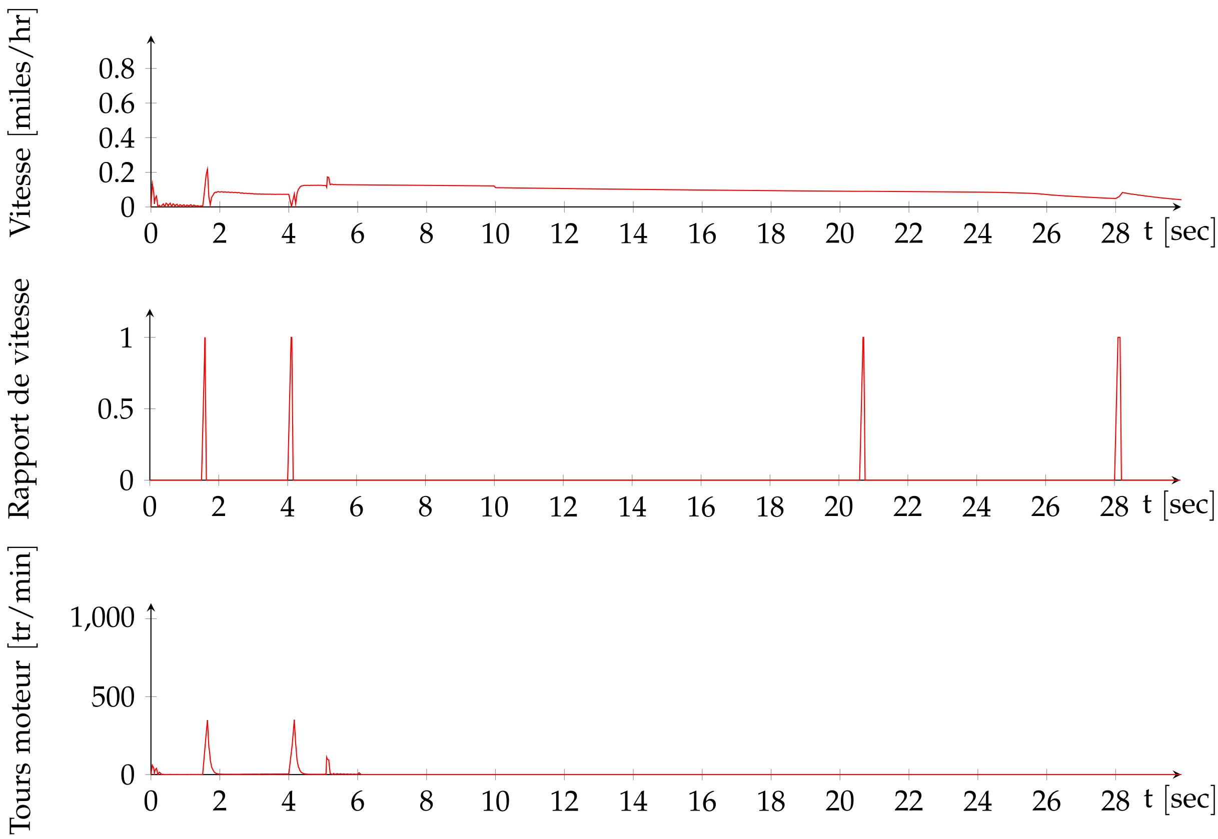


Figure B.2 – *Différences entre les signaux simulés par Zélus et les signaux simulés par Simulink (en valeur absolue)*

Scénario dépassement

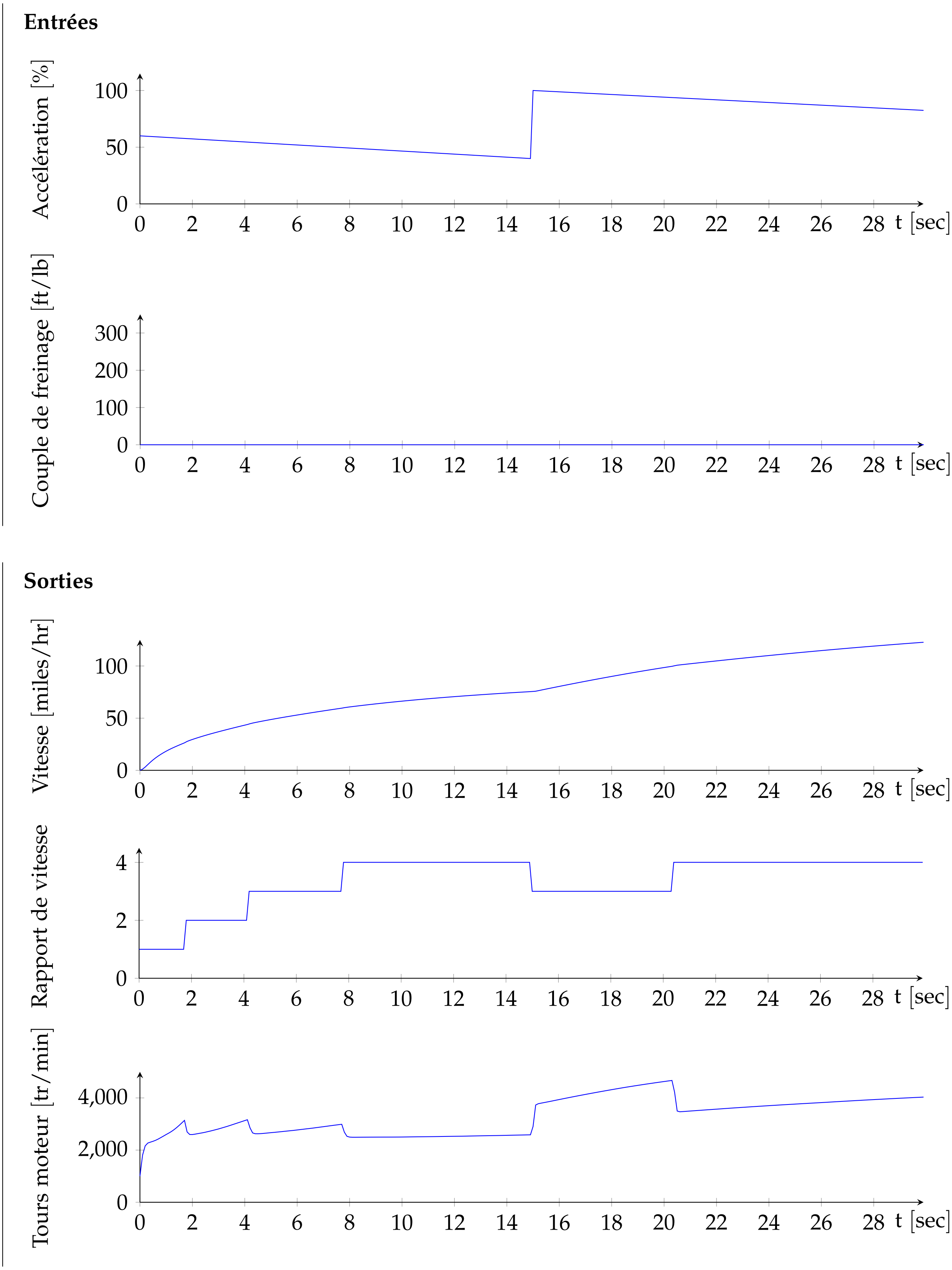


Figure B.3 – *Simulation du système (implémentation Zélus)*

Scénario dépassement

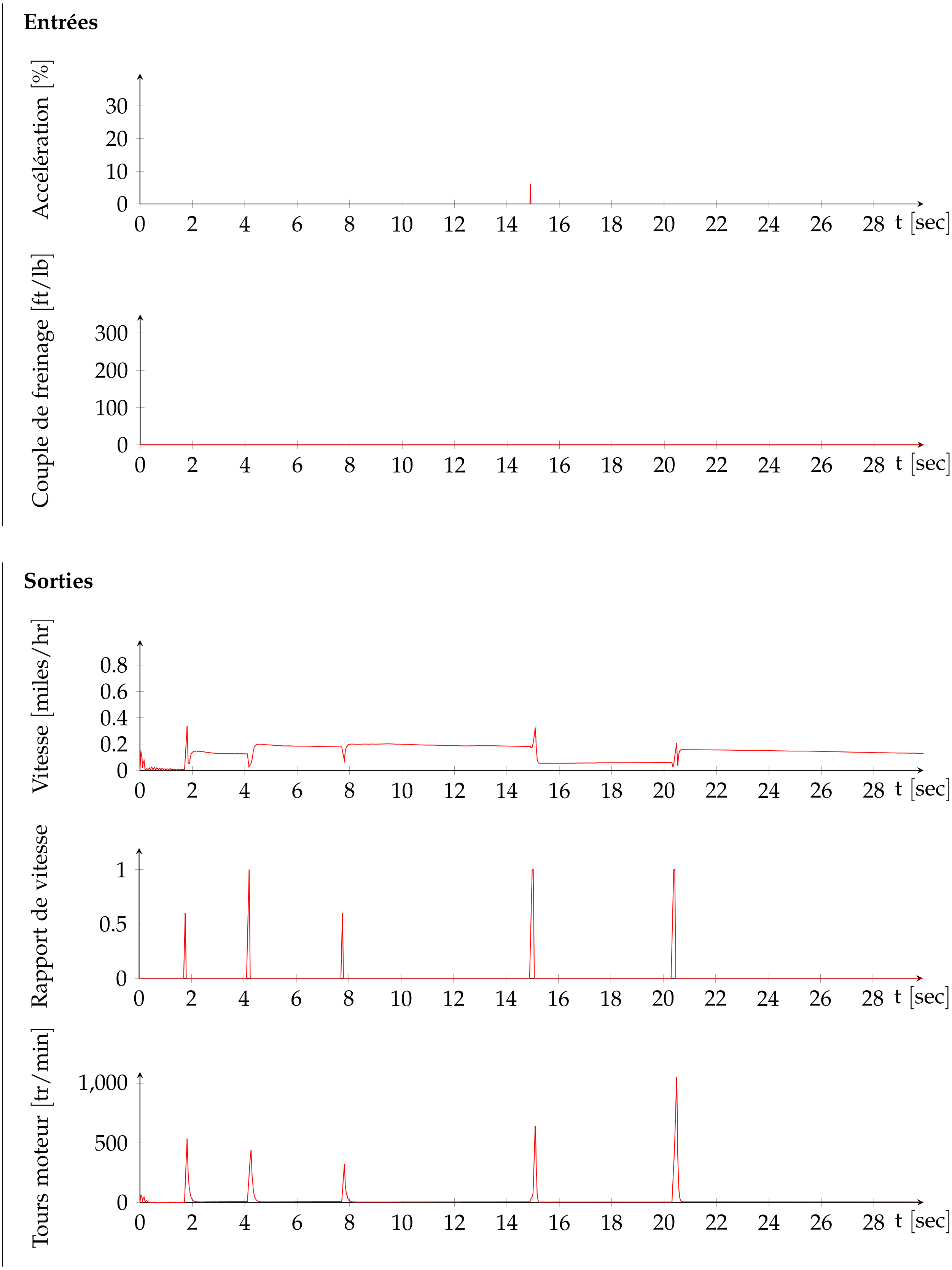


Figure B.4 – *Différences entre les signaux simulés par Zélus et les signaux simulés par Simulink (en valeur absolue)*

Scénario accélération progressive

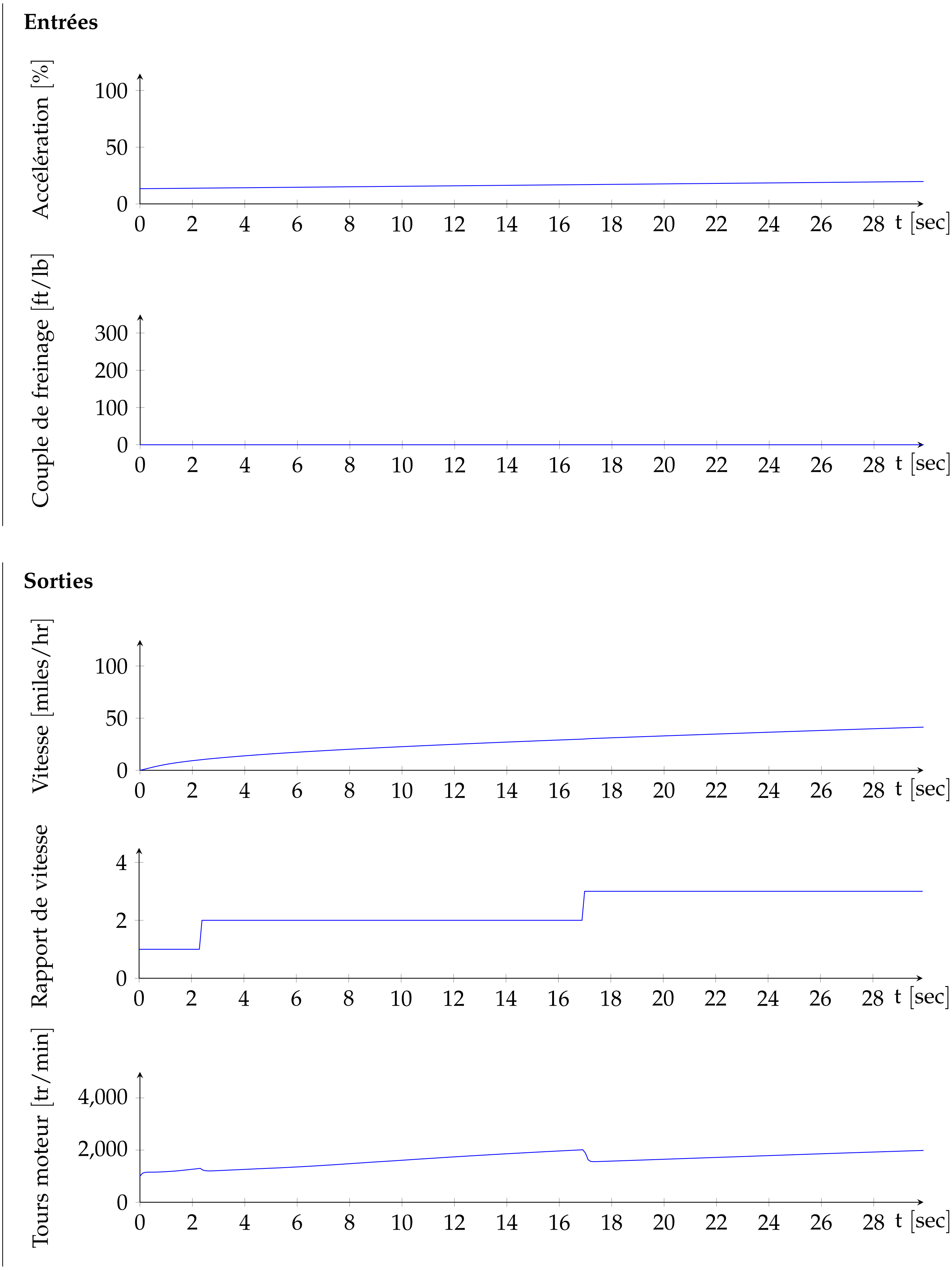


Figure B.5 – *Simulation du système (implémentation Zélus)*

Scénario accélération progressive

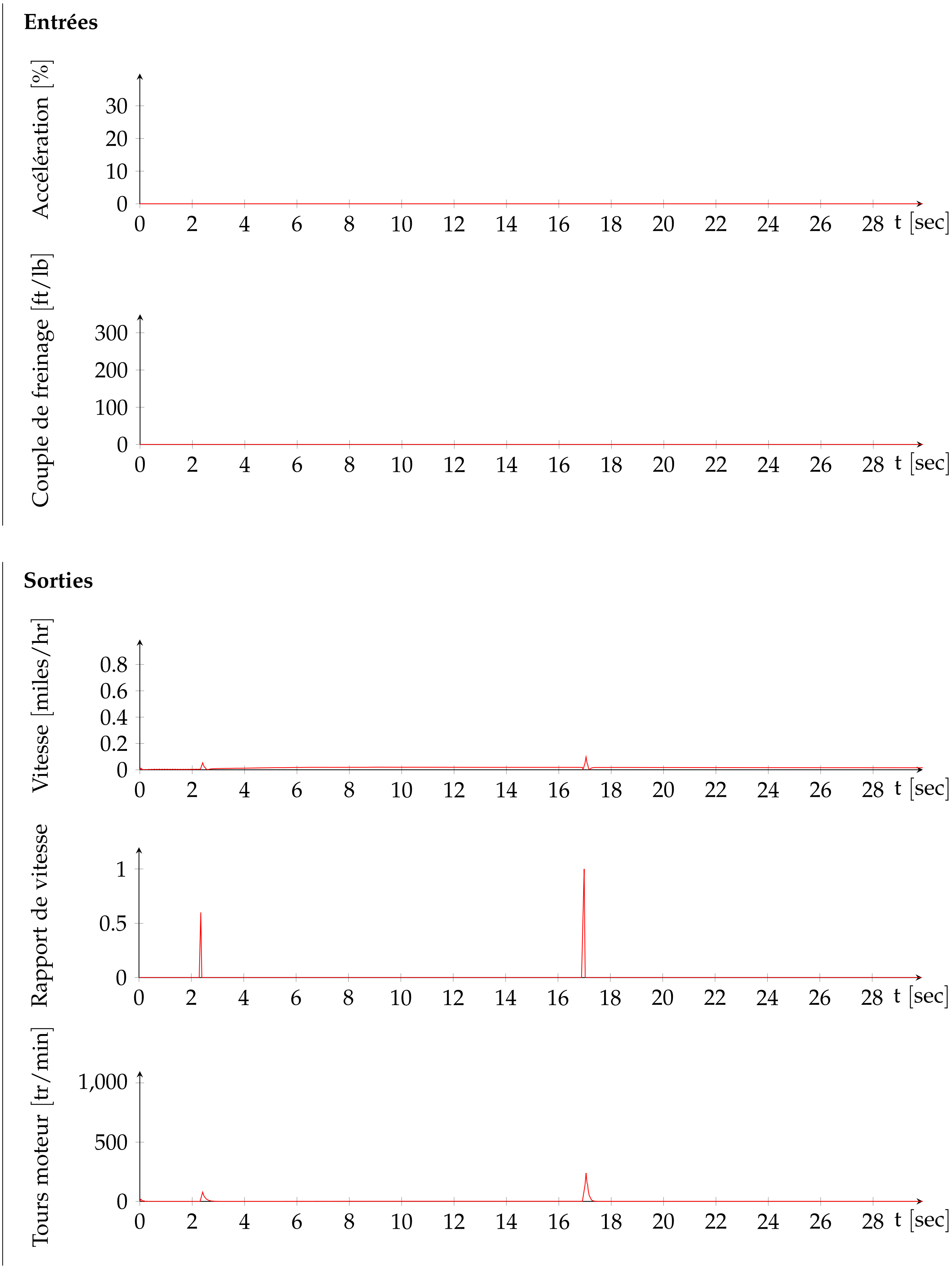


Figure B.6 – *Différences entre les signaux simulés par Zélus et les signaux simulés par Simulink (en valeur absolue)*

Scénario roue libre

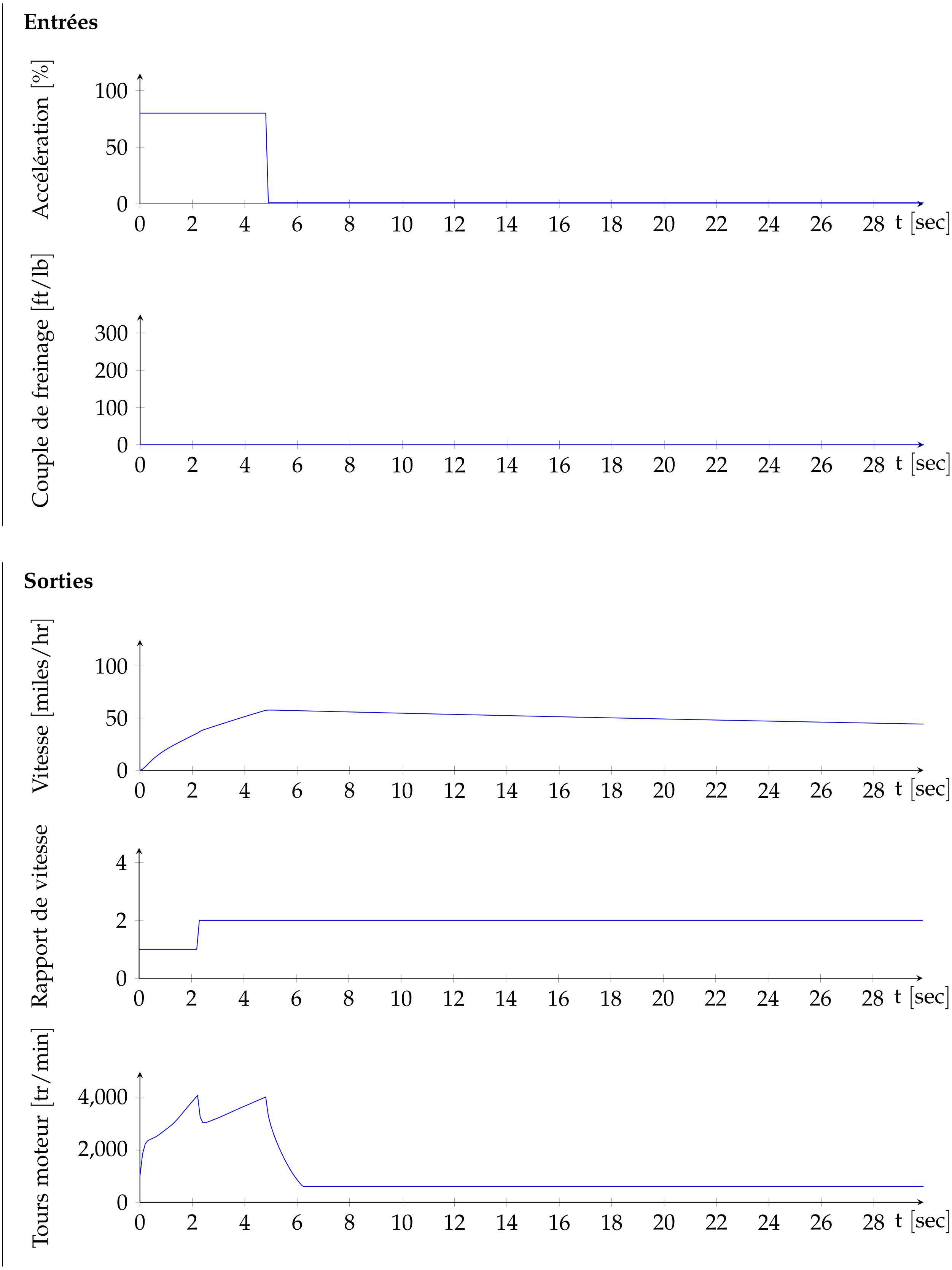


Figure B.7 – *Simulation du système (implémentation Zélus)*

Scénario roue libre

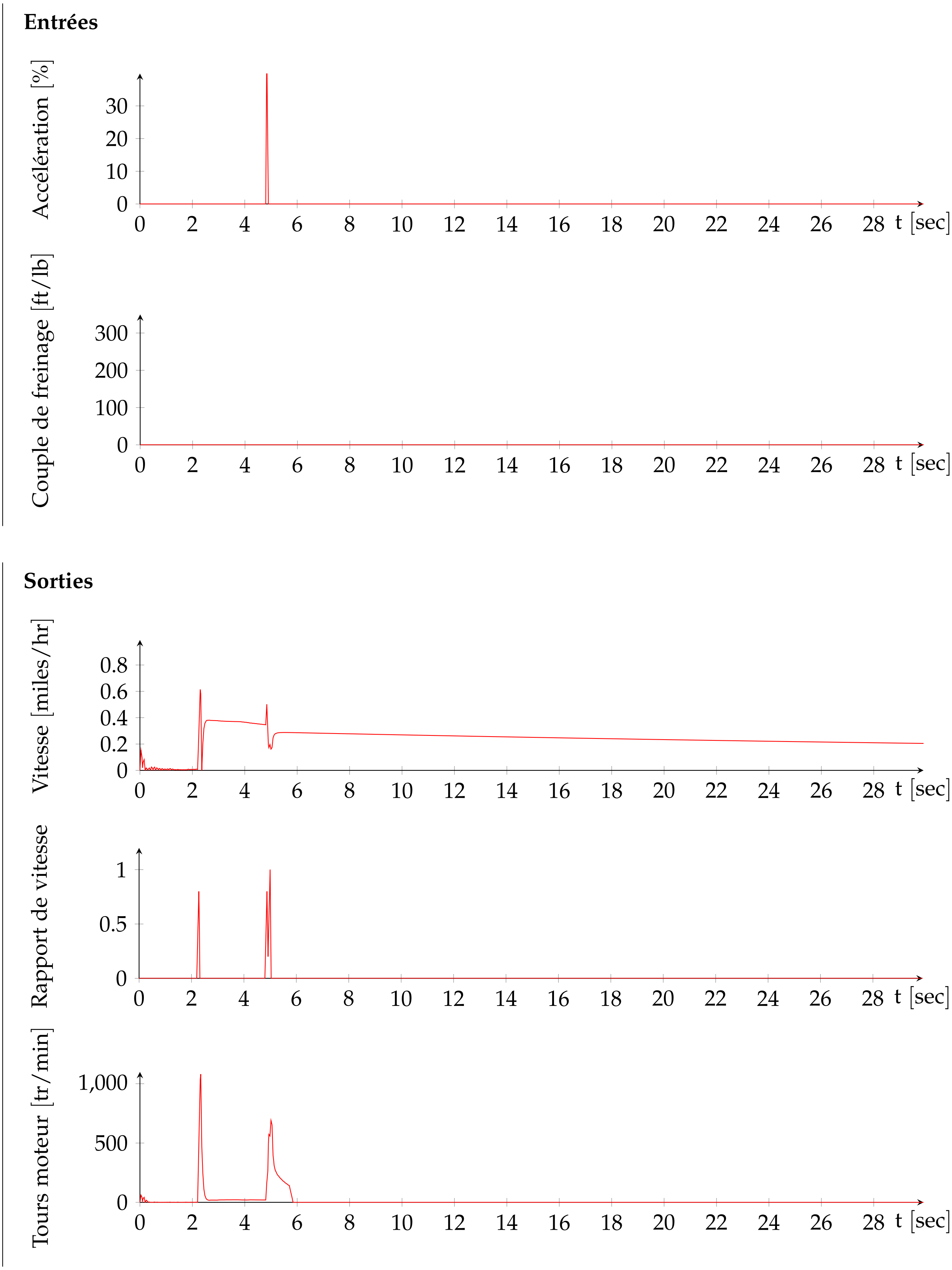


Figure B.8 – *Différences entre les signaux simulés par Zélus et les signaux simulés par Simulink (en valeur absolue)*

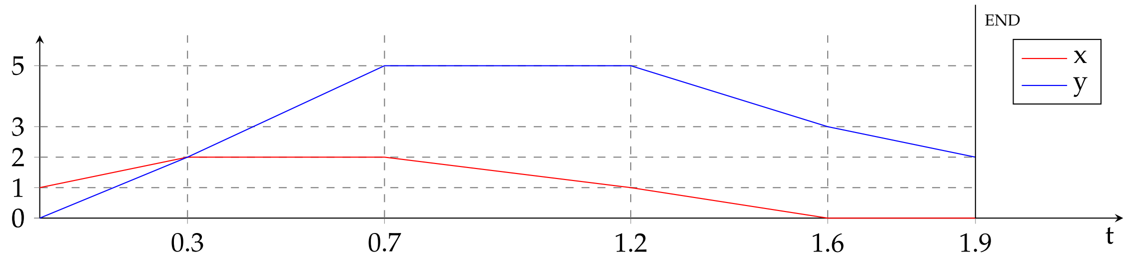
C. Démonstration de la propriété page 11

$$
\begin{aligned}
[\![\varphi_1 \; U_I \; \varphi_2]\!]_{MITL}(s,t_i) &= \exists\, t' \in \; t_i +_{\mathbb{R}} I, (\forall\, t'' \in [t_i,t'], [\![\varphi_1]\!]_{MITL}(s,t'')) \wedge [\![\varphi_2]\!]_{MITL}(s,t') \\
&= \bigvee_{j \,\mid\, t_j \in t_i +_{\mathbb{R}} I} \Big(\bigwedge_{i \le k \le j} [\![\varphi_1]\!]_{MITL}(s,t_k) \wedge [\![\varphi_2]\!]_{MITL}(s,t_j) \Big) \\
&= \bigvee_{j \ge i} \Big((t_j \in t_i +_{\mathbb{R}} I) \wedge \bigwedge_{i \le k \le j} [\![\varphi_1]\!]_{MITL}(s,t_k) \wedge [\![\varphi_2]\!]_{MITL}(s,t_j) \Big) \\
&= \Big((t_i \in t_i +_{\mathbb{R}} I) \wedge [\![\varphi_1]\!]_{MITL}(s,t_i) \wedge [\![\varphi_2]\!]_{MITL}(s,t_i) \Big) \\
&\quad \vee \bigvee_{j > i} \Big((t_j \in t_i +_{\mathbb{R}} I) \wedge \bigwedge_{i \le k \le j} [\![\varphi_1]\!]_{MITL}(s,t_k) \wedge [\![\varphi_2]\!]_{MITL}(s,t_j) \Big) \\
&= [\![\varphi_1]\!]_{MITL}(s,t_i) \wedge \Big((t_i \in t_i +_{\mathbb{R}} I) \wedge [\![\varphi_2]\!]_{MITL}(s,t_i) \\
&\quad \vee \bigvee_{j > i} \Big((t_j \in t_i +_{\mathbb{R}} I) \wedge \bigwedge_{i < k \le j} [\![\varphi_1]\!]_{MITL}(s,t_k) \wedge [\![\varphi_2]\!]_{MITL}(s,t_j) \Big) \Big) \\
&= [\![\varphi_1]\!]_{MITL}(s,t_i) \wedge \Big((0 \in I) \wedge [\![\varphi_2]\!]_{MITL}(s,t_i) \\
&\quad \vee \bigvee_{j \ge i+1} \Big((t_j \in t_{i+1} +_{\mathbb{R}} (I -_{\mathbb{R}} \delta t_i) \wedge \bigwedge_{i+1 \le k \le j} [\![\varphi_1]\!]_{MITL}(s,t_k) \wedge [\![\varphi_2]\!]_{MITL}(s,t_j) \Big) \Big) \\
&= [\![\varphi_1]\!]_{MITL}(s,t_i) \wedge \Big([\![\varphi_2]\!]_{MITL}(s,t_i) \wedge (0 \in I) \vee [\![\varphi_1 \; U_{I -_{\mathbb{R}} \delta t_i} \; \varphi_2]\!]_{MITL}(s,t_{i+1}) \Big)
\end{aligned}
$$

D. Exécution de l'algorithme d'évaluation en ligne de la robustesse d'une formule MITL

Exemple Exécution de l'algorithme

Simulation



Formule initiale

$\varphi = x > 0 \; U_{[1,2]} \; y > 5$

Étapes d'exécution

	$t = 0$, $\delta t = 0.3$	$t = 0.3$, $\delta t = 0.4$	$t = 0.7$, $\delta t = 0.5$
Entrée	$x > 0 \; U_{[1,2]} \; y > 5$	$1 \wedge x > 0 \; U_{[0.7,1.7]} y > 5$	$1 \wedge x > 0 \; U_{[0.3,1.3]} \; y > 5$
Sortie	$1 \wedge x > 0 \; U_{[0.7,1.7]} y > 5$	$1 \wedge x > 0 \; U_{[0.3,1.3]} \; y > 5$	$1 \wedge x > 0 \; U_{[-0.2,0.8]} \; y > 5$

	$t = 1.2$, $\delta t = 0.4$	$t = 1.6$, $\delta t = 0.3$
Entrée	$1 \wedge x > 0 \; U_{[-0.2,0.8]} \; y > 5$	$1 \wedge (4 \vee x > 0 \; U_{[-0.6,0.4]} \; y > 5)$
Sortie	$1 \wedge (4 \vee x > 0 \; U_{[-0.6,0.4]} \; y > 5)$	$1 \wedge (4 \vee (0 \wedge (9 \vee x > 0 \; U_{[-0.9,0.1]} \; y > 5)))$

	$t = 1.9$, $last = true$
Entrée	$1 \wedge (4 \vee (0 \wedge (9 \vee x > 0 \; U_{[-0.9,0.1]} \; y > 5)))$
Sortie	1

E. Librairie de génération d'entrées (en Zélus)

Interface

```
type inputs
type trace = inputs × float → float
type event = inputs × float → unit signal

type interval = float × float

val t_fby : (trace × event × trace) -S-> trace
val t_loop : (trace × event) -S-> trace
val t_switch : (float -C-> bool) × trace × trace -S-> trace

val flatten : trace
val const : interval -S-> trace
val slope : interval -S-> trace

val horizon : float -S-> event
val val_gt : float -S-> event
val val_lt : float -S-> event

val e_and : event × event -S-> event
val e_or : event × event -S-> event
```

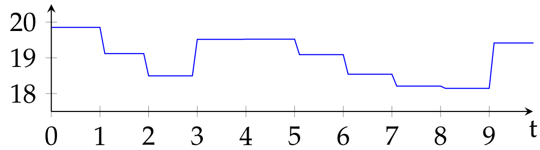
Implémentation

```
(* TRACE COMBINATORS *)

let hybrid t_fby(t1, e, t2)(params, initial) = t where
  rec init last_val = initial
  and automaton
  | NotSeenE → do t = run t1 (params, last last_val)
               until (run e (params, t))() then
               do last_val = t in SeenE
  | SeenE    → do t = run t2 (params, last last_val) done

let hybrid t_loop(t_in, e)(params, initial) = t_res where
  rec init last_val = initial
  and automaton
  | SeenE → do t_res = run t_in (params, last last_val)
            until (run e (params, t_res))() then
            do last_val = t_res in SeenE

let hybrid t_switch(cond, t1, t2)(params, initial) =
    if (run cond initial) then
      (run t1 (params, initial))
    else (run t2 (params, initial))

(* TRACE GENERATION *)

let hybrid const(i1, i2)(params, initial) = res where
  init res = pick_float(i1, i2)

let hybrid flatten(params, initial) = res where
  init res = initial

let hybrid slope(i1, i2)(params, initial) = res where
  rec init new_slope = pick_float(i1, i2)
  and der res = new_slope init initial

(* EVENT FUNCTIONS *)

let hybrid horizon(h)(_) = e where
  rec der t = 1. init 0.
  and present up(t -. h) → do emit e = () done

let hybrid val_gt0(x) = e where
  present up(x) → do emit e = () done
let hybrid val_gt(f)(params, x) = val_gt0(x -. f)
let hybrid val_lt(f)(params, x) = val_gt0(f -. x)

let hybrid e_and(e1, e2)(f) = e where
  rec e1_listener = (run e1 f)
  and e2_listener = (run e2 f)
  and init got1 = false
  and init got2 = false
  and init is_done = false
  and present
  | e1_listener() on (got2 && (not (is_done)))
  | e2_listener() on (got1 && (not (is_done))) →
    do emit e = () and next is_done = true done
  | e1_listener() on (not (got2)) → do next got1 = true done
  | e2_listener() on (not (got1)) → do next got2 = true done

let hybrid e_or(e1, e2)(f) = e where
  rec e1_listener = (run e1 f)
  and e2_listener = (run e2 f)
  and init is_done = false
  and present
  | e1_listener() on (not (last is_done))
  | e2_listener() on (not (last is_done)) →
  do emit e = () and is_done = true done
```

QUELQUES EXEMPLES

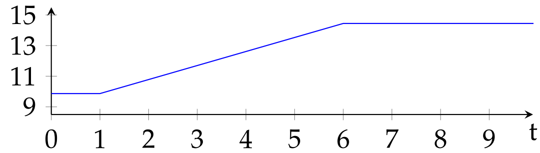
Noeud	Simulation

```
let hybrid gen_trgt2() =
  t_loop (const((18., 20.)), horizon(1.))(0., 0.)
```



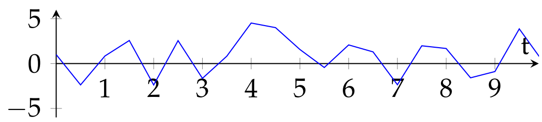
Une fonction constante dans $[18, 20]$ qui change de valeur toutes les 1 unité de temps.

```
let hybrid gen_trgt1(inp, x0) =
  t_fby(const(9., 10.), horizon(1.),
  t_fby(slope(0.9, 1.), horizon(5.), flatten))(0., 0.)
```



Une fonction constante dans $[9, 10]$ pendant 1 unité de temps puis une fonction affine de pente dans $[0.9, 1]$ pendant 5 unités de temps puis une fonction à partir de là.

```
let hybrid cond(f) = b where
  rec init b = (f < 1.)
  and present
  | up(f -. 1.) -> do b = false done
  | up(1. -. f) -> do b = true done

let hybrid gen_trgt4(inp, prev) =
  t_loop(
  t_switch(cond, slope(0.5, 10.), slope(-10., -0.5)),
  horizon(0.5))(0., 1.)
```



Une fonction de valeur initiale 1 qui change de pente toutes les 0.5 unités de temps. Si la valeur courante de la fonction est plus grande que 1 la pente est choisie dans $[-10, -0.5]$, sinon elle est choisie dans $[0.5, 10]$

F. Librairie d'observateurs synchrones (en Zélus)

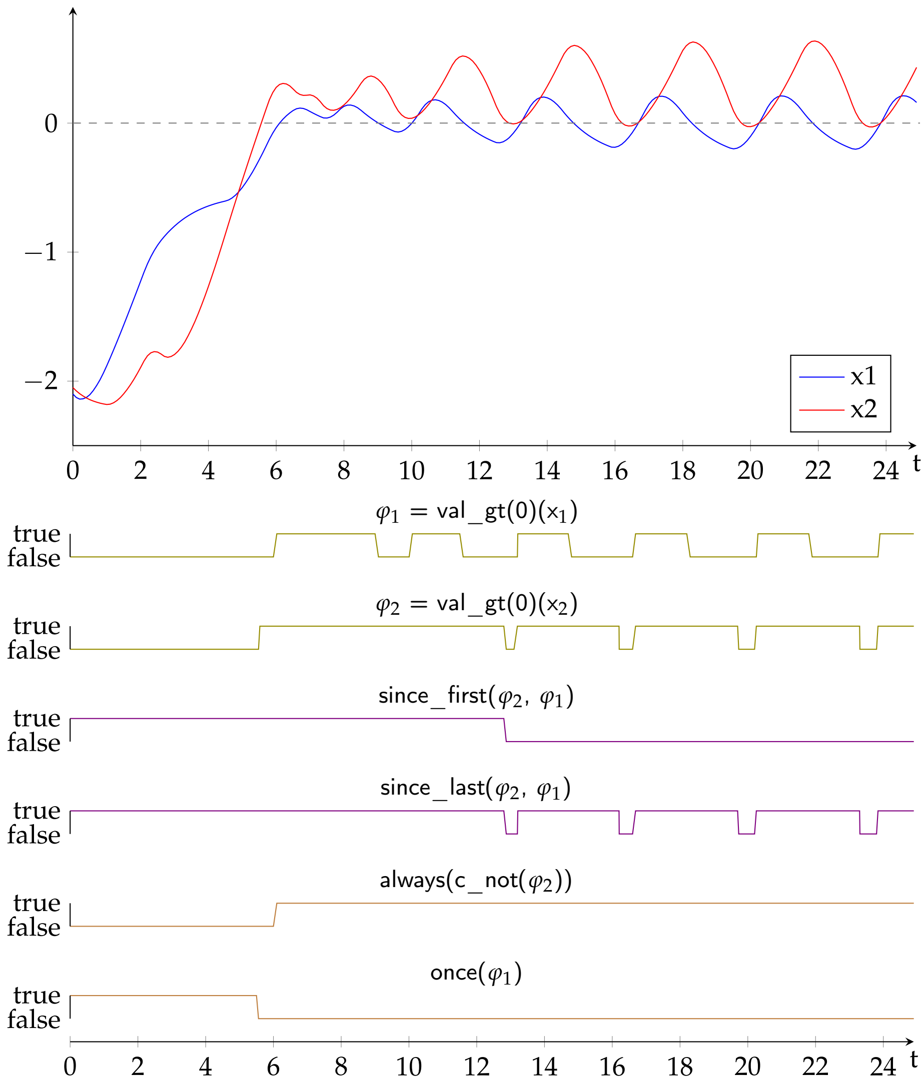
Interface

```
type state = bool
type state_change = (float × bool) signal
type sig = state × state_change

val val_gt : float -S-> float -C-> sig
val val_lt : float -S-> float -C-> sig

val c_not : sig -C-> sig

val c_and : sig × sig -C-> sig
val c_or : sig × sig -C-> sig

val once : sig -C-> sig
val always : sig -C-> sig

val since_first : sig × sig -C-> sig
val since_last : sig × sig -C-> sig
```

Implémentation

```
let hybrid val_gt0(t, x) = r, e where
  rec init r = (x > 0.)
  and present
    up(x) →
      do r = true
      and emit e = (t, true) done
    | up(−. x) →
      do r = false
      and emit e = (t, false) done
    | (disc(x)) →
      do r = (x > 0.)
      and emit e = (t, x > 0. done

let hybrid val_gt(f)(t, x) = val_gt0(t, x −. f)
let hybrid val_lt(f)(t, x) = val_gt0(t, f −. x)

let hybrid c_not((rA, eA)) = r, e where
  r = not rA
  and present eA(tA, b) →
    do emit e = (tA, not b) done

let hybrid c_and((rA, eA), (rB, eB)) = r, e where
  rec r = rA && rB
  and present
  | eA(t, true) on (rB) | eB(t, true) on (rA) →
    do emit e = (t, true) done
  | eA(t, false) | eB(t, false) →
    do emit e = (t, false) done

let hybrid c_or((rA, eA), (rB, eB)) = r, e where
  rec r = rA || rB
  and present
  | eA(t, true) | eB(t, true) →
    do emit e = (t, true) done
  | eA(t, false) on (not rB) | eB(t, false) on (not rA) →
    do emit e = (t, false) done

let hybrid once(rA, eA) = r, e where
  rec init r = rA
  and present eA(tA, true) on (not last r) →
    do emit e = (tA, true) and r = true done

let hybrid always(rA, eA) = r, e where
  rec init r = rA
  and present eA(tA, false) on (last r) →
    do emit e = (tA, false) and r = false done

let hybrid since_first((rA, eA), (rB, eB)) = r, e where
  rec rseenB, eseenB = once(rB, eB)
  and init r = rA || (not rseenB)
  and present
  | eseenB(t, true) on (rA)  →
    do r = true  and emit e = (t, true)  done
  | eA(t, false) on (rseenB) →
    do r = false and emit e = (t, false) done

let hybrid since_last((rA, eA), (rB, eB)) = r, e where
  rec init r = rA || (not rB)
  and present
  | eB(t, true) on (rA)       →
    do r = true and emit e = (t, true) done
  | eA(t, false) on (last r) →
    do r = false and emit e = (t, false) done
```

Quelques exemples



G. Quelques formules MITL écrite grâce à la librairie d'observateurs

— La vitesse n'est jamais inférieure à 30 mph lorsqu'on est en 3e vitesse

$\hookrightarrow$ MITL : $\Box\Big(\neg\ (gear(t) = 3 \wedge speed(t) < 30)\Big)$

$\hookrightarrow$ Zélus :

```
let hybrid never_gear3_and_speed_low(t, throttle, brake_torque, rpm, gear, speed) =
let speed_low = val_lt(v_low)(t, speed) in
let gear3 = c_and(val_in(2.5, 3.5)(t, gear)) in
let gear3_and_speed_low = c_and(gear3, speed_low) in
always (c_not gear3_and_speed_low)
```

— Lorsqu'on passe en 2e vitesse, on y reste pendant au moins 1 seconde

$\hookrightarrow$ MITL : $\Box\Bigg(\Big((\neg\ gear(t) = 2) \wedge \Diamond_{[0.01,0.02]} gear(t) = 2\Big) \Rightarrow \Box_{[0,1]} gear(t) = 2\Bigg)$

$\hookrightarrow$ Zélus :

```
let hybrid alw_stay2_for_t1(t, throttle, brake_torque, rpm, gear, speed) =
let gear2 = val_in(1.5, 2.5)(t, gear) in
for_since_last(t1)(t, gear2, gear2)
```

— Pendant les 25 premières secondes, la vitesse est supérieure à 30 mph, pendant les 25 secondes suivantes, la vitesse est inférieure à 150 mph

$\hookrightarrow$ MITL : $\Big((\Box_{[0,25]} speed(t) < 150) \wedge (\Box_{[25,50]} speed(t) > 30)\Big)$

$\hookrightarrow$ Zélus :

```
let hybrid vmaxmin(t, throttle, brake_torque, rpm, gear, speed) =
always(
c_and(
c_imp(
val_lt(25.)(t, t),
val_lt(150.)(t, speed)),
c_imp(
val_in(25., 50.)(t, t),
val_gt(30.)(t, speed))))
```

— Soit la vitesse est toujours inférieure à 100 mph, soit elle est toujours supérieure à 30 mph et est supérieure à 100 mph dans les 25 premières secondes

$\hookrightarrow$ MITL : $\Big((\Box\ speed(t) < 100) \vee (\Diamond_{[0,25]}(speed(t) > 100) \wedge (\Box\ speed(t) > 30))\Big)$

$\hookrightarrow$ Zélus :

```
let hybrid brake(t, throttle, brake_torque, rpm, gear, speed) =
c_or(
c_and(
once(
c_and(
val_gt(100.)(t, speed),
val_lt(25.)(t, t))),
always(val_gt(30.)(t, speed))),
always(val_lt(100.)(t, speed)))
```

— L'assertion "la vitesse est supérieure à 100 mph dans la première seconde et le moteur tourne toujours à moins de 4000 rpm" est fausse
$\hookrightarrow$ MITL : $\neg\Big((\Diamond_{[0,1]} speed(t) > 100) \wedge (\Box\, rpm(t) < 4000)\Big)$
$\hookrightarrow$ Zélus :

```
let hybrid phi100(t, throttle, brake_torque, rpm, gear, speed) =
c_not(
c_and(
once(c_and(
val_gt(100.)(t, speed),
val_lt(1.)(t, t))),
always(val_lt(4000.)(t, rpm))))
```