Données formalisées intégrant les exigences fonctionnelles

Client :	<->
Affaire/Projet :	COMON
Référence :	COMON/SPF5
Révision :	1.2
État :	PRE
Date :	1er octobre 2012
Classification :	Public
Nombre d'annexes :	4

1 Formalisation des données et conception dirigée par les modèles

1.1 Introduction

L'un des objectifs scientifiques du projet COMON (COnception dirigée par les MOdèles pour le Nucléaire) est de s'inspirer des préceptes de la conception basée sur les modèles pour mettre en œuvre des systèmes de contrôle-commande de centrales nucléaires (cf. section 4.1 de l'annexe technique [AT]). L'approche *dirigée par les modèles* s'appuie sur la construction précoce d'un modèle exécutable et sur son enrichissement continu au cours de la conception permettant la validation de chaque étape de conception (cf. section 2 de [AT]).

La formalisation des données est une étape primordiale dans cette approche, car c'est en s'appuyant sur cette formalisation que l'on va obtenir un modèle simulable du système de contrôle-commande en cours de conception. Par ailleurs, ces données formalisées vont également permettre d'automatiser des tests de conformité aux exigences fonctionnelles à chaque étape de la conception.

1.2 Correction fonctionnelle d'un système

Prouver que, ou tester si, un système est correct, c'est *confronter* une mise en œuvre et une spécification. On parle aussi de vérification de la conformité d'une mise en œuvre à sa spécification. Quand la spécification est *formelle*, la vérification de la conformité peut-être *automatisée*. Mais quand bien même une spécification formelle existe, celle-ci est nécessairement issue d'une spécification en langue naturelle, elle-même issue d'une analyse des besoins. C'est cette spécification en langue naturelle qui va servir de base à la construction de tout l'édifice. Afin de pouvoir automatiser la vérification de la conformité, l'étape la plus importante, et la plus difficile, va être d'obtenir une formalisation de ces exigences fonctionnelles la plus fidèle possible au document de référence. Pour cela, on doit chercher à obtenir une spécification la plus lisible et la plus concise possible. On veut, dans un premier temps, une formalisation des exigences qui ne se préoccupe pas du *comment* mais du *quoi* (le cahier des charges). Ainsi, les différentes étapes du développement que l'on propose dans ce projet sont les suivantes :

1. Analyse des besoins.
2. Spécification des exigences fonctionnelles (cahier des charges) en langue naturelle et à l'aide de dessins informels.
3. Formalisation de ces exigences fonctionnelles.
4. Obtention d'un modèle abstrait et exécutable satisfaisant ces exigences.
5. Obtention d'une implémentation finale satisfaisant ces exigences.

Quelques remarques au sujet de ces 5 étapes.

- Les trois premières étapes relèvent d'un processus humain difficile – mais crucial. Dans le projet COMON, nous supposons posséder le document issu de l'étape 2. Cependant, un des apports de la méthodologie va être de proposer des compléments à ce document, et ainsi d'y détecter d'éventuelles incohérences.
- La différence entre les modèles formels obtenus en étapes 3 et 4 est la suivante : dans l'étape 3, on ne s'intéresse qu'au « quoi », alors que le modèle de l'étape 4 commence à parler du « comment ». Plus précisément, dans l'étape 3, on écrit un modèle qui, à partir d'une séquence d'entrées/sorties du système, répond (calcule) si oui ou non la séquence est conforme. À l'étape 4, on écrit un modèle capable de générer une séquence de sortie conforme à partir d'une séquence d'entrée, ce qui est plus difficile. L'intérêt de passer par le modèle intermédiaire de l'étape 3 est de faciliter la démonstration (informelle) de la conformité du modèle à la description de l'étape 2. Cette phase de vérification doit être le plus facile possible, car l'interprétation d'un texte en langue naturelle ne peut pas être entièrement automatisée.
- Les étapes 3 et 4 peuvent être démarrées en parallèle par des équipes différentes.
- La vérification de la conformité des modèles formels obtenus dans les étapes 3 et 4 peut être automatisée.
- L'étape 5 peut-être vue comme l'aboutissement d'un raffinement itératif et continu de l'étape 4. Le premier modèle exécutable obtenu en étape 4 n'est pas complet ; typiquement, le matériel est complètement abstrait dans les premières phases.

1.3 Le processus de test proposé avec Lurette

Lurette est un outil de test de systèmes réactifs, conçu et mis en œuvre à Verimag. C'est un outil de tests fonctionnels, ou boîte noire, c'est-à-dire qui n'analyse pas le programme source du code à tester. Lurette permet d'automatiser la génération d'entrées du système sous test, ainsi que le dépouillement du résultat de ces tests. Pour pouvoir automatiser ces deux étapes, Lurette a besoin d'une description formelle (c'est-à-dire exécutable ou simulable automatiquement) des spécifications du système sous test.

Pour automatiser les tests de conformité aux exigences fonctionnelles, Lurette a besoin de la formalisation issue de l'étape 3 du paragraphe précédent. Mais ce que ne fait pas apparaître la description de cette étape jusqu'à présent, c'est que ces données formalisées sont de deux natures : certaines concernent les attendus du système, et d'autres décrivent les hypothèses faites sur le comportement de son environnement. Dans la formalisation requise par Lurette, cette distinction doit être explicite.

1.3.1 Formalisation des attendus fonctionnels (oracles)

Une partie des attendus fonctionnels décrivent comment le système doit se comporter dans certaines situations, comme par exemple lever une alarme quand un seuil est dépassé. La formalisation de ces attendus permet d'automatiser le dépouillement des tests. On parle aussi d'*oracles* du test. Plus précisément, un oracle est un programme qui décide si une séquence d'entrées/sorties est conforme.

Dans ce document et dans les expérimentations menées pour COMON, nous avons utilisé le langage Lustre pour écrire les oracles. Lustre est un langage flot de données faisant partie de la famille des langages dit Synchrones (Esterel, Signal, Scade), conçus pour la modélisation et la mise en œuvre de systèmes réactifs critiques. Les concepts de temps et de mise en parallèle sont constitutifs de ces langages, ce qui les rend particulièrement adaptés à la description d'oracles de systèmes réactifs.

1.3.2 Formalisation des hypothèses faites sur l'environnement du système (stimulateurs)

Une autre partie des exigences fonctionnelles concernent les hypothèses faites, parfois implicitement, sur l'environnement du système ; en effet, aucun système de contrôle-commande n'est censé fonctionner correctement dans un environnement quelconque. L'exemple typique est celui du système de signalisation ferroviaire dont l'environnement est constitué de trains : ce système ne pourra pas garantir l'absence d'accident si on ne fait pas l'hypothèse que les trains s'arrêtent aux feux rouges. Ces hypothèses, une fois explicitées et exprimées sous forme de contraintes, vont permettre à Lurette de générer (pseudo-aléatoirement) des entrées réalistes au système sous test. On parle également de *générateurs de stimuli*. Le langage que nous utilisons pour écrire ces générateurs se nomme *Lutin* ; nous donnons des exemples commentés de programmes Lutin par la suite.

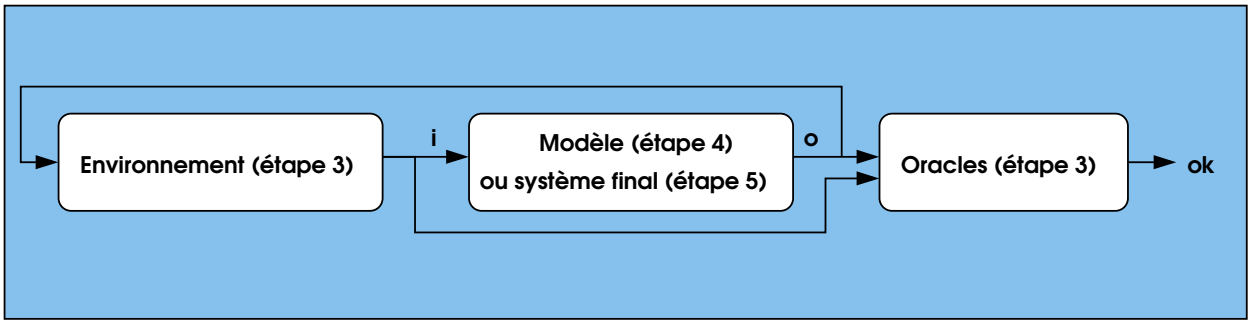


FIGURE 1 – Flot de données de test avec Lurette. Le modèle de l'environnement permet de fournir des séquences d'entrées (i) réalistes au modèle ou au système sous test. Ce dernier produit en retour une séquence de sorties (o). Ces séquences d'entrées/sorties sont fournies aux oracles qui décident si elles sont conformes.

Le flot des données de test est schématisé en Figure 1. Notez dans cette figure le rebouclage des sorties du système sous test sur son environnement. Cela constitue une des spécificités du test des systèmes réactifs, qui réagissent à

leur environnement, et cherchent à le contrôler : un environnement réaliste doit réagir aux commandes émises par le système, d'où ce rebouclage.

1.3.3 Couverture des tests fonctionnels

L'objectif de la méthodologie et de l'outillage proposé avec Lurette étant de vérifier la conformité d'une implémentation à sa spécification, il est naturel de définir un critère de couverture qui porte sur les oracles. L'idée est donc de s'assurer que chaque propriété définie dans un oracle est « couverte ». Considérons par exemple une propriété exprimée sous la forme d'une implication logique :

```
Cond => Obs
```

Les oracles ont cette forme dès que l'on veut exprimer des propriétés du style, « quand X a dépassé le seuil s, alors le booléen A doit être vrai », ou plus généralement, « quand on est dans tel état du système » (`Cond`), alors « on doit observer telles valeurs pour telles sorties » (`Obs`). Il y a 2 façons de constater la véracité d'une telle propriété : soit la variable `Cond` est fausse, soit elle est vraie et `Obs` doit l'être aussi. Du point de vue de la couverture de cette propriété, c'est manifestement le deuxième cas qui est intéressant. Pour vérifier que cette propriété est « couverte », il suffit donc de vérifier que le booléen `Cond` passe au moins une fois à vrai au cours d'une exécution.

Plus généralement, nous définissons la *couverture* d'un oracle par un ensemble de variables booléennes, dont on veut voir chacun des éléments passer à vrai au moins une fois au cours d'une exécution. Dans la méthodologie que nous proposons, c'est aux personnes qui formalisent les attendus fonctionnels de spécifier en même temps les attendus en terme de couverture.

D'un point de vue technique, dans Lurette, pour définir la couverture d'un oracle, il suffit de mettre les variables de couverture en sortie de l'oracle. La convention est que la première sortie indique si l'oracle est satisfait, et que les sorties suivantes définissent la couverture. L'outil se sert de ces sorties pour mettre à jour un taux de couverture d'une exécution à l'autre, via un fichier de couverture (`.cov`), et qui est ré-initialisé à chaque fois que l'implémentation ou l'oracle change.

1.3.4 Des scénarios de test pour améliorer la couverture

Considérons par exemple une propriété qui exprime que si une entrée numérique `I` est comprise entre `0.0` et `0.5` pendant au moins 2.5 secondes, alors la sortie S doit être vraie.

```
est_vrai_depuis_n_secondes(2.5, 0.5 < I < 0.0) => S
```

Pour couvrir une telle propriété, il faut générer une séquence d'entrée où `I` est dans cet intervalle pendant 2.5 secondes ; on a peu de chance d'y arriver sans écrire explicitement un scénario de test qui spécifie ce comportement.

Couvrir les critères de couverture qui portent sur les entrées du système à tester est facile. Le travail est plus ardu quand ces critères portent sur les sorties, ou sur des états internes du système. Dans ce cas il faut mettre en place des scénarios qui amènent le système dans des configurations particulières, ce qui nécessite une expertise sur l'application à tester (expertise nécessaire avec des méthodes de test traditionnelles également). C'est en particulier cette volonté de pouvoir exprimer facilement des scénarios qui a motivé la conception du langage Lutin ; en effet, les langages purement flot de données comme Lustre ne sont pas adaptés pour exprimer des scénarios.

1.3.5 Des oracles contextuels à la phase du scénario

On vient d'expliquer comment l'écriture de scénario de test permettait d'augmenter la couverture de test. Mais on peut voir les choses selon une perspective opposée. En effet, dans le document qui décrit en langue naturelle les exigences fonctionnelles du système à mettre en œuvre, les propriétés sur les entrées du système (hypothèse sur l'environnement) et sur ses sorties (exigences fonctionnelles) peuvent être entremêlées ; certains événements ne doivent être observés qu'après que certaines séquences de stimuli aient été observées. En d'autres termes, au cours du déroulement d'un test, la validité de certaines propriétés n'est exigée que dans des phases bien particulières du scénario de test.

Dans de tels cas, il peut être tentant de vouloir écrire les oracles et les séquences de stimuli conjointement, voire au sein du même fichier. Néanmoins, la séparation explicite entre hypothèses et attendus (ainsi qu'entre les équipes qui les formalisent) est saine. Nous nous sommes contenté dans nos expérimentations d'une approche où les stimulateurs et les oracle étaient séparés.

1.3.6 Un processus itératif

En résumé, la mise en œuvre du système, de ses oracles, et de ses générateurs de stimuli n'est nullement un processus linéaire. De multiples aller/retour (raffinement) entre ces différentes étapes sont nécessaires pour les raisons suivantes.

- Quand un oracle est invalidé, soit un vrai problème a été identifié et il faut remettre en cause le système sous test, soit il faut remettre en cause l'oracle. En effet, comme déjà mentionné, l'étape 3 dite de formalisation des exigences, est la plus délicate. L'interprétation d'un texte en langue naturelle parfois ambigu est difficile. Par ailleurs le document obtenu en étape 2 peut également contenir des incohérences corrigées lors d'étape (4) de mise en œuvre.
- Quand le taux de couverture n'est pas suffisant, il faut enrichir l'ensemble des comportements possibles de l'environnement par de nouveaux scénarios. Ce faisant, de nouvelles propriétés peuvent être falsifiées, amenant à des modifications du système sous test ou des oracles, modifications ayant elles-mêmes une influence sur le taux de couverture.

Ce processus itératif s'accommode tout à fait de l'essence même de l'approche dirigée par les modèles, qui consiste à définir le plus tôt possible un modèle exécutable du système, pour pouvoir le simuler et le tester à toutes les étapes du cycle de développement. Cette démarche est synthétisée en Figure 2.

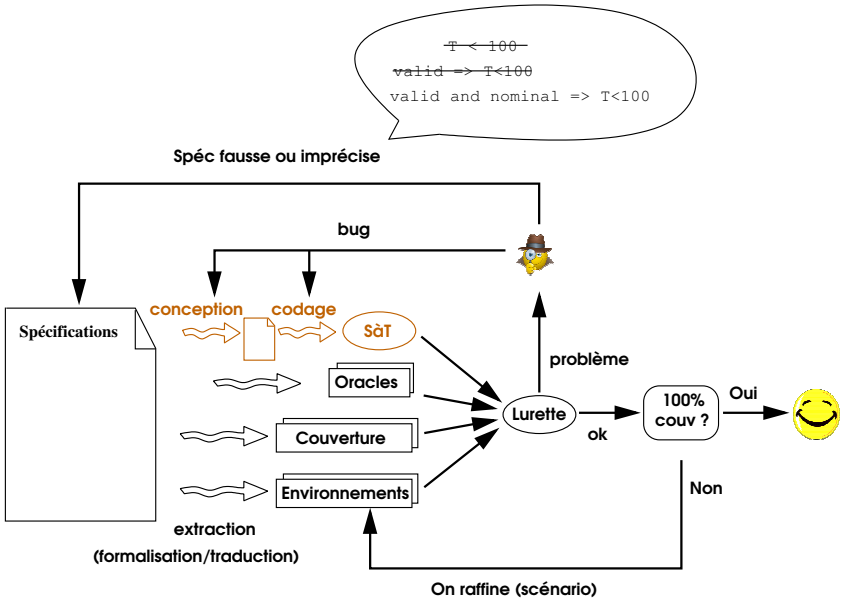


FIGURE 2 – Processus de test Lurette. Les oracles et l'environnement du système sous test sont extraits des spécifications généralement exprimées en langue naturelle. Quand un oracle est invalidé, il peut s'agir d'un mauvais codage (bug) ou bien d'une spécification pas assez précise. Par exemple, quand il est écrit dans la spécification que la température ne dois jamais dépasser 100 degrés, il n'est parfois pas précisé explicitement que cela ne peut être le cas que si le bit de validité associé au capteur est à `vrai`, et que le système est en mode `nominal`. Une fois que le système s'exécute sans invalider d'oracles, il faut regarder le taux de couverture de ces oracles ; pour améliorer ce taux de couverture, le testeur est amené à raffiner ses scénarios de tests.

1.4 Cahiers de test

Afin de vérifier la conformité de leur système, les industriels rédigent des cahier de test, en se basant sur les exigences fonctionnelles. Ces cahiers se présentent généralement sous forme de matrices dont les lignes représentent des étapes du test, et dont les colonnes spécifient les valeurs des entrées (à positionner), et des sorties (à observer). Ces cahiers sont destinés à être « joués » manuellement, c'est-à-dire que le testeur déroule séquentiellement chacune de ces étapes, en positionnant les entrées, et en vérifiant que les sorties sont conformes aux attendus.

Corys utilise un outil développé en interne qui automatise ce processus (I&C simulation), mais les instants et les valeurs sont trop précis – d'autres valeurs, à d'autres instants conviendraient. En d'autres termes, ces cahiers de test, bien que leur déroulement soit automatisé, sont trop déterministes – le testeur qui déroule sont cahier de test à la main introduit au moins un non-déterminisme temporel.

Les cahiers de tests peuvent être généralisés et traduits en termes :

- de scénarios aléatoires de stimulation versus un scénario précis (du point de vue des valeurs des entrées et du moment où elles changent),
- d'oracles,
- de critères de couverture (définis implicitement dans la colonne « observation » de ces cahiers).

Nous illustrons cette démarche dans une expérimentation relatée en annexe C, où nous montrons comment automatiser l'exécution d'un cahier de test existant. Nous montrons en particulier comment, en rendant moins déterministe le scénario de stimulation d'un cahier de test, nous avons en fait décrit le scénario de stimulation de 21 autres cahiers.

1.5 Non-régression

Si la mise en place initiale des stimulateurs et des oracles nécessite un investissement conséquent, la vérification de la non-régression au cours du cycle de développement, et surtout au cours des différentes évolutions mineures, est quant à elle quasiment gratuite. En effet, tous les oracles et stimulateurs concernant des parties du système qui ne changent pas peuvent être réutilisés tels quels.

Notons que l'on pourrait aussi utiliser Lurette pour mettre en place une stratégie de vérification de non-régression plus fine, en écrivant des oracles qui formalisent l'évolution entre la version « n-1 » et la version « n » du système. Ces oracles prendraient en entrée les sorties des deux versions du système, et décideraient si elles sont équivalentes.

1.6 Méthodes agiles

Remarquons que la méthodologie proposée ici s'accorderait particulièrement bien avec des méthodes de développement agiles. Trois équipes – une équipe développement, une équipe oracle, et une équipe scénario – peuvent travailler en parallèle, et ce dès les phases initiales du développement.

D'ailleurs, l'idée du développement orienté par les modèles, où l'on cherche à obtenir au plus tôt un modèle exécutable du système à mettre en œuvre, est tout à fait dans l'esprit des méthodes agiles.

1.7 Plan du document

Nous présentons dans les sections annexes, de manière linéaire, les expérimentations effectuées au cours du projet en terme de formalisation des exigences et de tests. Nous nous sommes appuyés sur ces expérimentations pour chacune des 2 sections du corps de ce document. La section 2 présente les oracles et les stimulateurs utilisés lors des expérimentations qui ont une portée plus générale, et qui sont donc susceptibles de faire partie d'une bibliothèque. La section 3 reprend certaines parties de ces expérimentations afin d'illustrer les concepts et les processus décrit dans la section 1.3. La section 3, moins technique, peut être lue avant la section 2 ; nous avons malgré cela choisi de la présenter après car elle fait référence à des oracles et des stimulateurs définis en section 2.

2 Une ébauche de Bibliothèque d'oracles et de stimulateurs génériques

Certaines propriétés, faisant intervenir des combinaisons de variables booléennes et du temps, peuvent être délicates à définir. Cela étant, ces propriétés « booléanno-temporelles » dont on a besoin sont toujours un peu les mêmes. C'est pourquoi il est important du point de vue de la formalisation des exigences fonctionnelles, de pouvoir s'appuyer sur une bibliothèque de propriétés correctes et faciles à utiliser. Leur correction peut être garantie par le fait qu'elle soient réutilisées, mais aussi par model-checking (cf Section 2.4).

Nous présentons ici, parmi les oracles et les stimulateurs utilisés lors des expérimentations (cf. sections annexes), ceux d'entre eux qui ont une portée plus générale, et qui sont donc susceptibles de faire partie d'une bibliothèque. Nous mentionnons également en fin de section une série d'opérateurs utiles élaborés avant le projet COMON.

Plutôt que d'avoir une section « oracles », et une section « stimulateurs », nous avons choisi dans cette section de faire une présentation thématique qui alterne la description des uns et des autres. Nous avons fait ce choix tout d'abord parce que cette formalisation est issue d'un même document décrivant les exigences fonctionnelles du système à réaliser (cf. étape 2 du processus décrit en section 1.2). Par ailleurs, beaucoup de propriétés issues de cette formalisation servent à la fois à la définition des oracles et des stimulateurs. Un programme Lutin déterministe ressemble d'ailleurs comme deux gouttes d'eau à son équivalent en Lustre, car ils sont basés sur les mêmes principes : flot de données, temps logique, composition synchrone.

2.1 Gérer la redondance matérielle

Pour formaliser les environnements et les oracles où de la redondance matérielle est présente, nous sommes toujours amenés à vouloir exprimer des propriétés du style « au moins 2 parmi ces 3 booléens doivent être vrais », ou bien « au plus 1 parmi 4 doivent être vrais », etc. Bien sûr, ces nœuds ne sont pas très difficiles à écrire ; néanmoins, il est relativement facile de se tromper en les écrivant. Il est donc important que ce type de nœuds soient dans des bibliothèques.

En Lustre, il est possible de définir de tels nœuds de façon générique en utilisant des tableaux. Définissons donc le nœud `au_plus_n_parmi_m`, qui prend un tableau de booléen de taille `m`, et qui rend vrai si et seulement si ce tableau contient au plus `n` éléments de vrai. Pour cela, nous définissons d'abord un nœud qui incrémente un compteur si une certaine condition `c` est vraie.

```
-- incremente le compteur si c
node incr_cond(cpt:int; c:bool) returns (n_cpt:int);
let
  n_cpt = cpt + if c then 1 else 0;
tel;
```

Ensuite, il ne reste plus qu'à utiliser ce nœud pour compter le nombre de booléens qui sont vrais dans le tableau de booléens, et de comparer ce nombre obtenu à `n`. En utilisant l'itérateur Lustre V6 `red`, qui permet d'itérer un nœud sur tous les éléments d'un tableau, on peut définir le nœud `au_plus_n_parmi_m` en une équation :

```
node au_plus_n_parmi_m<<const m:int>>(n:int; t:bool^m) returns (res:bool);
let
  res = (red<<incr_cond, m>>(0, t) <= n);
tel;
```

Les arguments entre << >> sont des paramètres statiques qui doivent être connus à la compilation. Un tel nœud générique Lustre V6 peut s'instancier ainsi :

```
node au_plus_n_parmi_10(n: int; b: bool^10) returns(ok: bool)
let
  ok = au_plus_n_parmi_m<<10>>(n, b);
tel
```

qui s'écrit aussi plus succinctement ainsi :

```
node au_plus_n_parmi_10 = au_plus_n_parmi_m<<10>>;
```

Les nœuds `au_moins_n_parmi_m` et `exactement_n_parmi_m` peuvent s'écrire tout aussi aisément, en remplaçant le « <= » par « >= » et par « = » respectivement.

En Lutin, nous n'avons pas à ce jour implémenté les tableaux et leurs itérateurs dans le langage, mais en attendant, il est bien sûr possible de définir « manuellement » toutes les instances utiles.

```
node lut_au_plus_n_parmi_10(n:int;b_0,b_1,b_2,b_3,b_4,b_5,b_6,b_7,b_8,b_9:bool)
returns (ok:bool)=
  let b2i(b:bool):int = if b then 1 else 0 in
  let cpt = b2i(b_0)+b2i(b_1)+b2i(b_2)+b2i(b_3)+b2i(b_4)+
            b2i(b_5)+b2i(b_6)+b2i(b_7)+b2i(b_8)+b2i(b_9)
  in
    loop ok = (cpt <= n)
```

La version générateur de ce nœud peut être tout aussi utile. Pour la mettre en œuvre, il suffit de déclarer les entrées comme des sorties.

```
node au_plus_n_parmi_10_gen(n:int)
returns (b_0,b_1,b_2,b_3,b_4,b_5,b_6,b_7,b_8,b_9:bool)=
  let b2i(b:bool):int = if b then 1 else 0 in
  let cpt = b2i(b_0)+b2i(b_1)+b2i(b_2)+b2i(b_3)+b2i(b_4)+
            b2i(b_5)+b2i(b_6)+b2i(b_7)+b2i(b_8)+b2i(b_9)
  in
    loop (cpt <= n)
node gen_n() returns(n:int) = loop (0 <= n and n <= 10)
```

À chaque pas, ce nœud va générer un entier n entre 0 et 10, et 10 booléens dont exactement n d'entre eux sont vrais.

N.B. : une fois que l'on a défini 2 implémentations d'une même spécification, il peut être intéressant de chercher à les confronter. Pour cela, il suffit d'exécuter le générateur Lutin `au_plus_n_parmi_10_gen` dans Lurette et d'utiliser le nœud `au_plus_n_parmi_10` comme oracle.

```
lurettetop -go -l 20 -rp "sut:lutin:bool-utils.lut:au_plus_n_parmi_10_gen" \
  -rp "env:lutin:bool-utils.lut:gen_n"
  -rp "oracle:v6:bool-utils.lus:au_plus_n_parmi_10"
```

2.2 Parler du temps

Beaucoup d'attendus fonctionnels font référence au temps. Pour formaliser des propriétés temporelles, on peut soit compter le nombre de cycles (pour les programmes activés périodiquement), soit passer en paramètre la date à laquelle la machine est activée. Quand la machine réactive est activée de façon périodique (comme c'est le cas sur le banc de test COMON), pour pouvoir parler du temps, il suffit de compter le nombre de cycles. Par exemple, pour savoir si un signal est vrai depuis un certain temps, on peut utiliser le nœud Lustre suivant :

```
node vrai_depuis_n_secondes(n: real; signal: bool) returns (res: bool)
var
  tempo : real;
let
  tempo = n -> if not signal then n else max (0.0, pre tempo - cycle_time);
  res = (tempo = 0.0);
tel
```

où `cycle_time` est une constante exprimée en seconde. L'intérêt de passer par une telle constante est de pouvoir exprimer des propriétés temporelles indépendemment du cadencement choisi.

Une autre façon de compter le temps, qui est d'ailleurs la seule possible pour des processus non périodiques, est de passer en argument l'horodatage. Ainsi, le nœud qui calcule si un booléen est vrai depuis un certain temps peut s'écrire comme suit.

```
node vrai_depuis_n_secondes_bis(n, t : real; signal: bool) returns (res: bool)
var
  tempo : real;
  pt : real; -- un registre pour memoriser l'horodate de l'instant precedent
let
  pt = 0.0 -> pre t;
  tempo = n -> if not signal then n else max (0.0, pre tempo - (t-pt));
  res = (tempo = 0.0);
tel
```

La seule différence est que plutôt que de retrancher à `tempo` le temps de cycle, on retranche le temps écoulé entre l'instant courant et l'instant précédent.

De même pour les stimulateurs Lutin, par défaut le langage manipule des nombres de cycles.

```
node main() returns(x:int) = loop ~100 x=42
```

Ce programme génère pendant environ 100 cycles la valeur 42. Là encore, il peut parfois être plus commode de raisonner en temps réel, par exemple en définissant une macro `sec` qui traduit les secondes en nombre de cycles. Ainsi, pour générer la valeur 42 pendant 5 minutes, on peut procéder de la façon suivante.

```
let minutes(x:int):int = x*60*1000/cycle_time
node main() returns(x:int) = loop ~minutes(5) x=42
```

De la même manière, on peut définir une fois pour toute de telles macros pour les millisecondes, les secondes, les heures, les jours.

2.3 Détecter la stabilité d'une variable numérique

Pour stimuler ou surveiller des systèmes de régulation qui mettent en jeu des grandeurs numériques, il est utile de pouvoir détecter la stabilité d'une variable. Pour les oracles, cela permet d'exprimer des propriétés du type, « 2 minutes après un changement de consigne, la température doit être stable ». Pour les stimulateurs, cela permet d'écrire des scénarios dans lesquels on attend la stabilité avant de changer à nouveau la consigne.

La stabilité peut être définie en fonction de deux paramètres qui précisent la tolérance numérique, pour filtrer le bruit, et temporelle, pour préciser à partir de combien de temps on considère la variable comme stable. Plus précisément, on peut définir la *(d,ε)-stabilité* ainsi :

> On dit qu'une variable V est *(d,ε)-stable* à l'instant i s'il existe un intervalle I de taille ε tel que, $\forall$ t $\in$ [i-d, i], V(t) $\in$ I.

Pour mettre en œuvre un nœud qui satisfait cette définition, il faut autant de mémoires qu'il y a d'instants durant « d » secondes, afin de stocker les différentes valeurs prises par cette variable au cours de ce laps de temps, et ainsi pour pouvoir vérifier que l'écart entre ces valeurs ne dépasse jamais ε. En Lutin, pour d=3, cela donne :

```
node is_3_stable(eps, v:real) returns (v_average:real;res:bool) =
  exist v1,v2,v3 : real in
  assert v_average = (v1+v2+v3)/3.0 in
    { v1=v and v2=v and v3=v fby loop v1=v and v2=pre v1 and v3=pre v2
  &>
     loop [3] res = false
     fby loop res = abs(v1-v2)<eps and abs(v1-v3)<eps and abs(v2-v3)<eps
    }
```

Pour calculer la (m,ε)-stabilité avec « m » grand, ce nœud étant gourmand en mémoire (il requiert « m » cases mémoires), on peut calculer la (d,ε)-stabilité avec « d » de taille acceptable ; ensuite on vérifie que l'on est (d,ε)-stable pendant « m-d » cycles en incrémentant un compteur (cf. nœud vrai_depuis_n_secondes défini en section 2.2). Un problème avec cette façon de procéder est que si on choisit un « d » trop petit, on va considérer comme stables des variables qui dérivent de moins de ε pendant « d × cycle_time » secondes. Pour remédier à ce problème, on peut par exemple mémoriser la valeur de la moyenne des « d » dernières valeurs dès que v devient (d,ε)-stable, puis vérifier que v ne s'écarte pas de cette valeur de référence (vref) de plus de ε.

```
node is_m_stable_approx(m:int ; epsilon, v:real) returns (res:bool) =
  exist v_average : real in
  exist vref : real in
  exist looks_stable : bool in
  run v_average, looks_stable := is_3_stable(epsilon,v) in
  myloop (
   loop not looks_stable and not res and vref = v_average
    fby
    assert
      vref = pre vref and -- memorize v_average when entering the loop
      abs(v-vref) < epsilon -- start from the beginning when false
    in
      loop [m-3] not res fby loop res
  )
```

En Lustre, grâce aux tableaux, il est possible de définir une version générique (paramétrée par d) de la stabilité.

```
node is_d_stable<<const d: int; const eps: real>>(v: real)
returns (res:bool);
var
  v_mem: real^d; -- circular array
  cpt: int;
let
  cpt = 0 -> (pre cpt + 1);
  v_mem = 0.0^d -> assign<<d>>(v, cpt mod d, pre v_mem);
  res = cpt>=d-1 and (get_range<<d>>(v_mem) < eps);
tel
```

où

– assign«d»(v,i,t) renvoie le tableau t de taille d pour qui le i-ème élément a été remplacé par v ;

– get_range renvoie la différence entre la valeur maximale et la valeur minimale d'un tableau.

Notons que si ce programme est moins fastidieux à écrire en Lustre qu'en Lutin pour de grandes valeurs de « d », il est tout aussi coûteux en mémoire. On peut remédier à cela de la même façon que nous venons de le montrer en Lutin.

Une autre façon économique d'approcher la (d,ε)-stabilité est de calculer à chaque cycle un décompteur initialisé à « d », ainsi que la valeur minimale et la valeur maximale du signal. Ensuite, on considère être stable quand le décompteur est inférieur à 1, tout en re-initialisant le tout à chaque fois que max-min dépasse ε.

```
node is_stable_approx(v: real) returns (res:bool);
var
  vmin, vmax : real;
  cpt : int;
  reset : bool;
let
  reset = true -> (vmax - vmin > epsilon);
  vmin = if reset then v else minr(v, pre vmin);
  vmax = if reset then v else maxr(v, pre vmax);
  cpt  = if reset then d else max(0, pre cpt - 1);
  valid = (cpt = 0);
tel
```

L'inconvénient de cette version est qu'elle détecte la stabilité avec un peu de retard. Par ailleurs, pour les valeurs qui dérivent, elle peut se mettre à osciller (en retournant vrai pendant un certain temps, puis faux pendant un certain temps, etc.).

2.4 Vérifier qu'un événement en suit un autre avec décalage temporel

Les propriétés attendues sur des systèmes temps-réels, sont généralement du type, « quand il se passe ceci, alors au bout d'un certain temps, il doit se passer cela ». Plus précisément, si on dit qu'un événement *arrive à un instant t* quand la variable booléenne correspondante passe à vrai lors de cet instant (typiquement, cela peut-être la sortie d'un front montant ou descendant), on veut exprimer que quand un événement `e1` arrive, `e2` doit arriver dans les « `delais` » millisecondes suivantes. Par ailleurs, `e2` ne doit pas arriver avant `e1`. Pour formaliser ce type de propriété, une façon de procéder est d'utiliser un automate à 3 états `AE1` (Attend `e1`), `AE2` (Attend `e2`), et `Erreur`, comme suit.

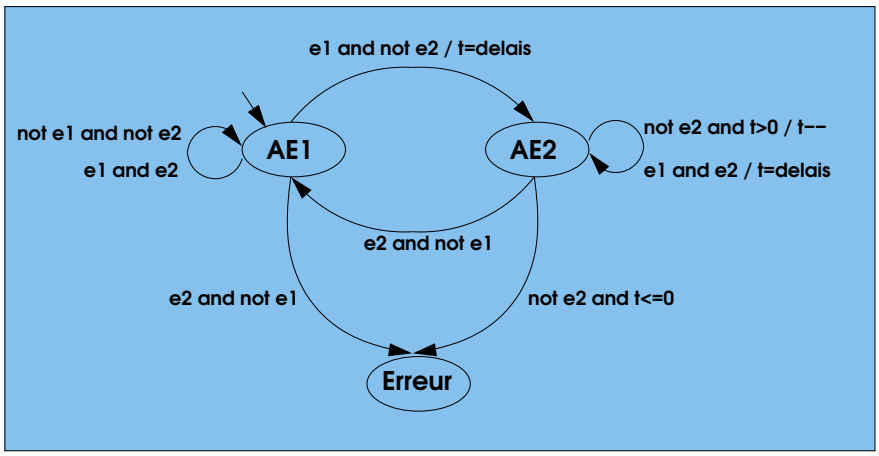


FIGURE 3 – Un oracle qui décide si `e2` arrive `delais` ms après `e1`.

- On attend `e1` (`AE1=true`) si
  - on attendait `e1` et qu'il n'y a ni `e1` ni `e2`
  - on attendait `e1` et qu'il y a `e1` et `e2`
  - on attendait `e2` et `e2` arrive (sans `e1`)
- On attend `e2` (`AE2=true`) si
  - on attendait `e1` et que `e1` arrive
  - on attendait `e2`, qui n'arrive pas, et le timer n'a pas expiré
  - on attendait `e2`, qui arrive, et `e1` arrive en même temps (discutable)
- On est dans l'état Erreur si
  - on attendait `e1`, et `e2` arrive sans `e1`
  - on attendait `e2`, et le timer expire.
- Le timer doit être réarmé à chaque fois que l'on devient (ou redevient dans l'instant) en attente de `e2`, et décrémenté quand on reste en attente de `e2`.

Cet oracle est schématisé en Figure 3, et encodé en Lustre en Figure 4. L'utilisation de ce nœud est illustré en section 3.1, pour vérifier que des alarmes sont levées lors de dépassement de seuils, et réciproquement.

N.B. : nous aurions pu ne pas explicitement définir l'état Erreur et les transitions qui y mènent, en définissant juste `AE1` et `AE2`, et en définissant le résultat de cet oracle comme le ou exclusif entre `AE1` et `AE2` : toutes les transitions absentes de ce programme menant implicitement vers un état `Erreur` lui-même implicite. L'avantage de passer comme nous l'avons fait par une définition explicite de l'état `Erreur` est que l'on peut exprimer précisément et clairement dans quels cas de figure l'oracle est faux. Par ailleurs, on peut tirer parti de cette redondance en vérifiant qu'elle est cohérente via l'utilisation d'un model-checker. Ainsi nous avons prouvé avec `lesar` (un outil de model-checking développé à Verimag) qu'exactement un des trois états est vrai à chaque instant.

```
node est_suivi_de(delais:real; e1,e2: bool) returns (ok:bool);
var
  AE1, pAE1: bool; -- l'etat "Attend E1", et son pre
  AE2, pAE2: bool; -- l'etat "Attend E2", et son pre
  Erreur,  pErreur : bool; -- l'etat Erreur, et son pre
  t, pt: real; -- la minuterie,  et son pre
let
  pAE1 = true  -> pre AE1; -- etat initial
  pAE2 = false -> pre AE2;
  pErreur = false -> pre Erreur;
  pt = delais -> pre t;
  t = if AE1 or e2 then delais else max_r(-temps_de_cycle, pt-temps_de_cycle);
  -- nb: 'temps_de_cycle' est une constante connue a la compilation
  AE1 = (pAE1 and (not e2) and not e1) or
        (pAE1 and e2 and e1) or
        (pAE2 and e2 and not e1);
  AE2 = (pAE1 and (not e2) and e1) or
        (pAE2 and t > 0.0 and not e2) or
        (pAE2 and e2 and e1);
  Erreur = pErreur or -- on reste en erreur pour toujours (discutable)
    (pAE1 and e2 and not e1) or -- on attend e1 et e2 arrive
    (pAE2 and not (t > 0.0) and not e2); -- on attend e2, qui n'arrive pas
                                         -- et le timer expire
   ok = not Erreur;
tel
```

FIGURE 4 – L'encodage en Lustre de l'automate de la Figure 3.

```
ok = exactement_1_parmi_n(3, [AE1, AE2, Erreur]);
```

2.5 Vérifier l'égalité de variables avec une tolérance temporelle

De la même manière qu'on peut vouloir exprimer des propriétés sur une variable avec une tolérance relative à sa valeur, il peut être intéressant de le faire avec une tolérance temporelle, c'est-à-dire qu'une variable y devient égal à x au bout d'au plus d'un certain temps.

Plus précisément, il s'agit de vérifier que les valeurs prises par une variable y sont dans l'intervalle des valeurs prises par une autre variable x au cours des d derniers instants. Pour ce faire, on peut utiliser un tableau circulaire de taille d, qui mémorise les valeurs prises pas x au cours des d derniers instants ; on en déduit, en calculant les extrema de cet intervalle, la plage des valeurs acceptables pour y.

```
node equal_i<<const d:int>>(x,y:int) returns (ok:bool);
var
  cpt:int;
  x_mem: int^d; -- tableau contenant les d dernieres valeurs de x
  min_x,max_x:int;
let
  cpt = 0 -> (pre cpt + 1) mod d;
  x_mem = x^d -> assign_i<<d>>(x, cpt, pre x_mem);
  min_x,max_x = get_extrema_i<<d>>(x_mem);
  ok = (min_x <= y and y <= max_x);
tel
```

Là encore, si on veut parler de tolérance temporelle en termes de delta_t millisecondes, il suffit de diviser par le temps de cycle, c'est-à-dire de prendre d=1+delta_t/tc. Si la tolérance temporelle est inférieure au temps de cycle, on utilisera donc ce noeud avec d valant 1, ce qui revient bien à tester l'égalité entre x et y.

Pour les booléens, le plus simple est d'utiliser la version pour les entiers (ce qui n'est pas optimal en espace, puisqu'un tableau de taille 2 suffirait).

```
node equal_b<<const d:int>>(x,y:bool) returns (ok:bool);
let
   ok = equal_i<<d>>(if x then 0 else 1, if y then 0 else 1);
tel
```

Remarquons que `equal_b«d»(e1,e2)` est différent de `est_suivi_de(d,e1,e2)`, car ce dernier n'accepte que des séquences où le nombre d'instants où `e1` et `e2` sont vraies sont les mêmes (à `d` près transitoirement).

Pour les réels, on peut en plus paramétrer ce noeud par une tolérance de `delta_v` sur les valeurs.

```
node equal_r<<const d:int>>(delta_v,x,y:real) returns (ok:bool);
var
  cpt:int;
  x_mem: real^d;
  min_x,max_x:real;
let
  cpt = 0 -> (pre cpt + 1) mod d;
  x_mem = x^d -> assign<<d>>(x, cpt, pre x_mem);
  min_x,max_x = get_extrema_r<<d>>(x_mem);
  ok = (min_x - delta_v < y and y < max_x + delta_v);
tel
```

Notez que tous ces noeuds ne sont pas symétriques (`y` prend la valeur de `x` au bout d'un certain temps). On peut facilement définir une version symétrique ainsi :

```
node sym_equal_r<<const d:int>>(delta_v,x,y:real) returns (ok:bool);
let
  ok = equal_r<<d>>(delta_v,x,y) or equal_r<<d>>(delta_v,y,x);
tel
```

2.6 Stimulation de variables numériques

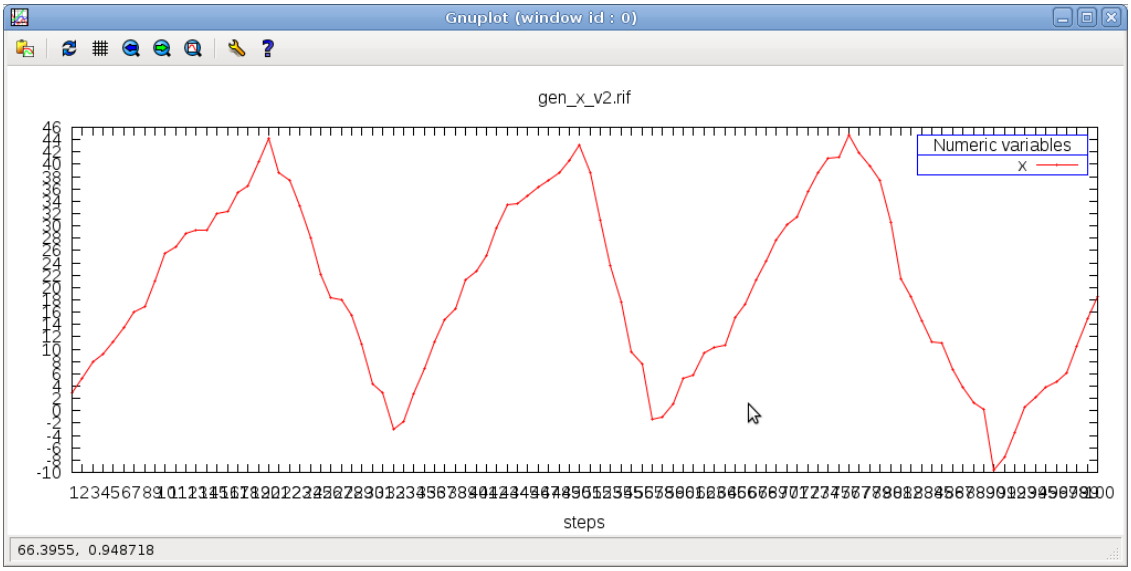
Faire varier une variable numérique dans un intervalle est extrêmement facile en Lutin.

```
node gen_x_v1() returns(x:real = 0.0) =
  loop -100.0 < x and x < 100.0
```

Néanmoins, il est souvent plus réaliste de borner la dérivée, et intéressant d'explorer de tout le domaine numérique.

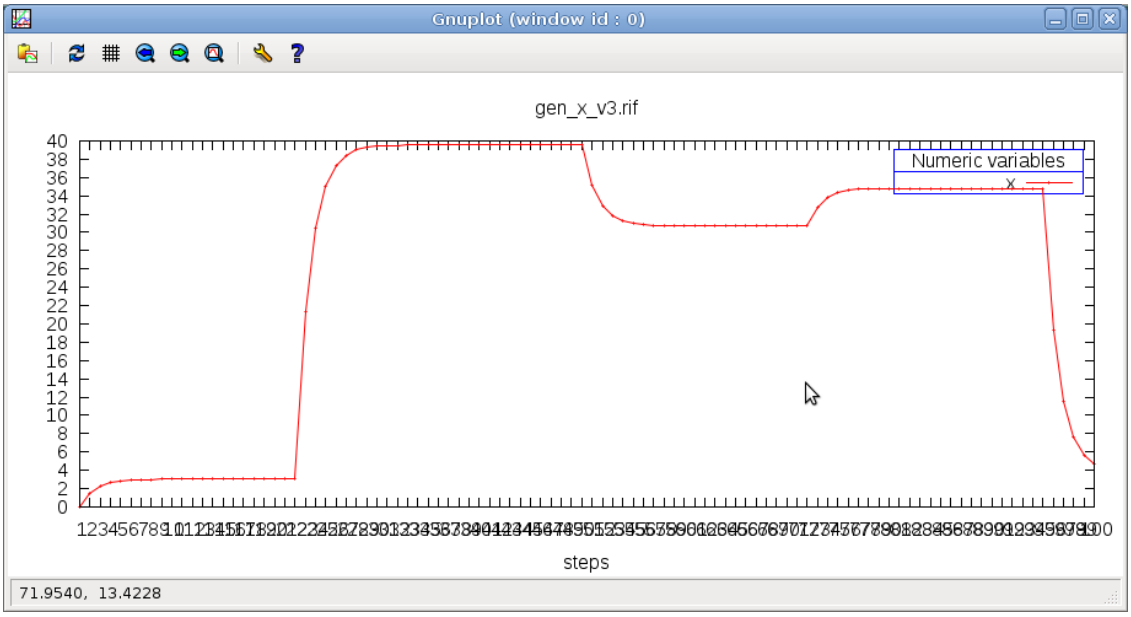
```
let between(x, min, max: real): bool = (min <= x) and (x <= max)
let up  (delta:real; x:real ref) : bool = between(x, pre x, pre x + delta)
let down(delta:real; x:real ref) : bool = between(x, pre x - delta, pre x)
node gen_x_v2() returns (x:real) =
   between(x, 0.0, 10.0) fby
   loop {
        | loop { up  ( 5.0, x) and pre x < 42.0 }
        | loop { down(10.0, x) and pre x > 0.0 }
      }
```

Ce programme choisit une valeur entre 0.0 et 10.0, puis il choisit de descendre (`down`) ou de monter (`up`). Pour monter, il choisit une valeur comprise entre sa valeur précédente (`pre x`) et sa valeur précédente plus `5.0`, et ce jusqu'à ce que la limite de 42.0 ait été dépassée. La descente est plus rapide (d'un décrément compris entre 0.0 et 10.0 à chaque pas), et s'arrête quand `0.0` est dépassé. Voici une représentation graphique des données obtenues sur une exécution de ce programme pendant 100 cycles :



Pour certaines applications, il peut-être intéressant de maintenir la valeur constante pendant un certain nombre de cycles. Pour cela on peut par exemple écrire un programme qui choisit une cible a atteindre, puis qui l'atteint de façon « amortie » :

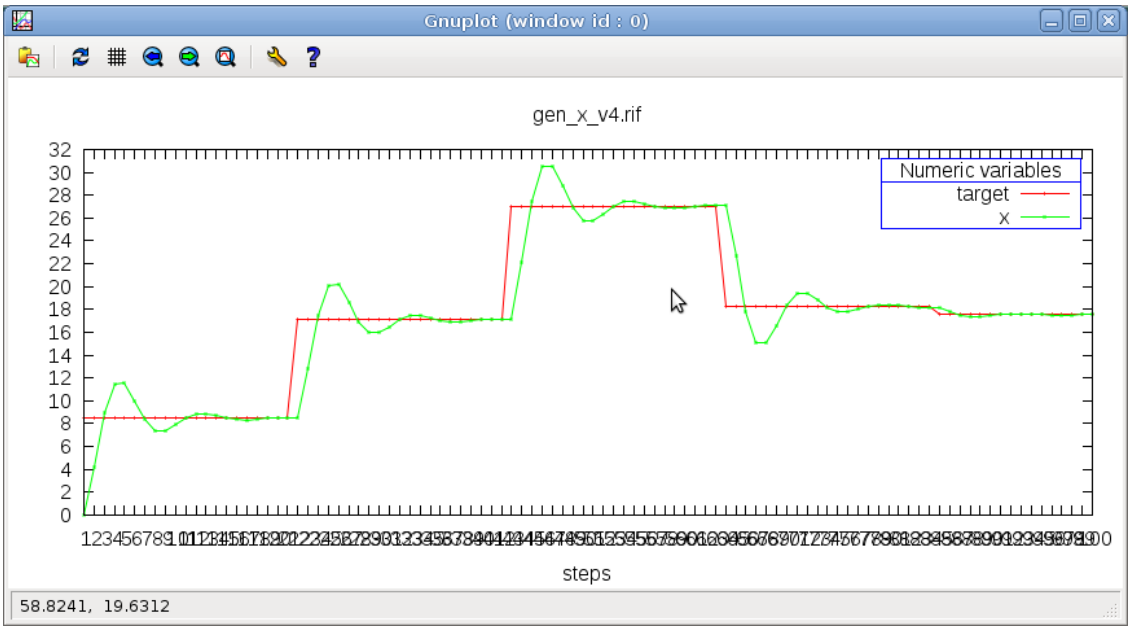
```
node gen_x_v3() returns(x:real = 0.0) =
  exist target : real in
  loop {
    0.0 < target and target < 42.0 and x = pre x
    fby loop [20,30] { x = (pre x + target) / 2.0 and target = pre target }
  }
```



On peut également modifier ce stimulateur en jouant sur le coefficient d'amortissement (ici, `2.0`), sur le nombre de cycle où on maintient la cible (ici, entre 20 et 30), ou sur l'intervalle dans lequel la cible est choisie (ici, entre `0.0` et `42.0`). On peut aussi rajouter un peu d'inertie dans la manière d'atteindre la cible de la manière suivante :

```
node gen_x_v4() returns(target, x:real = 0.0) =
```

```
exist px : real = 0.0 in -- Because pre pre x is currently not supported
assert px = pre x in
loop {
  0.0 < target and target < 42.0 and x = pre x
  fby loop [20] {
     x = (pre x + target) / 2.0 + 0.6*(px - pre px) -- adding inertia.
     and target = pre target
  }
}
```



2.7 Quelques utilitaires existants

Pour compléter cette ébauche de bibliothèque, nous proposons dans cette section un certain nombre d'opérateurs qui se sont avérés utiles dans d'autres études de cas menées par le passé. Voici en premier lieu une série d'opérateurs booléens temporels standards qui prennent un booléen en entrée, et retournent un booléen.

```
node after  (X:bool) returns (o:bool)
node jafter (X:bool) returns (o:bool);
node once   (X:bool) returns (o:bool);
node never  (X:bool) returns (o:bool);
node r_edge (X:bool) returns (o:bool);
node r_xedge(X:bool) returns (o:bool);
node f_edge (X:bool) returns (o:bool);
node f_xedge(X:bool) returns (o:bool);
node edge   (X:bool) returns (o:bool);
node xedge  (X:bool) returns (o:bool);
```

`jafter(X)` est faux au premier instant, et vrai si l'instant courant suit immédiatement un instant où `X` était vrai. `after(X)` est faux au premier instant, puis vrai si `X` a déjà été vrai dans le passé strict. `once(X)` est vrai si `X` est vrai ou bien a été vrai. `never(X)` est vrai si `X` n'est pas vrai et n'a jamais été vrai. `r_edge(X)` est égal à `X` au premier instant, puis vrai sur les fronts montants. `r_xedge(X)` est détecte aussi les fronts montants, mais il est toujours faux au premier instant. De même on définit `f_edge` et `f_xedge` pour les fronts descendants, ainsi que `edge` et `xedge` pour les fronts montants ou descendants.

```
node atlast(A,B:bool) returns (o:bool);
```

```
node switch(past,on,off:bool) returns (o:bool);
```

`atlast(A,B)` est faux au premier instant, puis prend la valeur de `A` chaque fois que `B` est vrai, et conserve sa valeur sinon. `switch(past,on,off)` est un interrupteur à deux états qui est :

- égal à `past` à l'origine des temps ;
- vrai si `on` et s'il n'était pas vrai à l'instant précédent ;
- faux si `off` et s'il n'était pas faux à l'instant précédent ;
- dans son état précédent sinon.

Voici maintenant une série d'observateurs qui comparent les occurrences d'événements (c'est-à-dire les instants où des variables booléennes sont vraies) entre eux. Nous avons intentionnellement omis de préciser ce qui se passait aux bornes des intervalles définis par ces événements ; en fait on peut définir des versions strictes et non strictes pour chacun de ces noeuds (comme pour `edge` et `xedge`).

```
node once_since(A,B:bool)           returns (o:bool);
node once_from_to(A,B,C:bool)       returns (o:bool);
node cont_since(A,B:bool)           returns (o:bool);
node cont_from_to(A,B,C:bool)       returns (o:bool);
node never_between_and(A,B,C:bool) returns (o:bool);
node alternate(A,B:bool)            returns (o:bool);
```

`once_since(A,B)` est vrai tant que `B` n'est pas vrai, puis est vrai à partir du moment où `A` a été vrai après le dernier instant où `B` a été vraie. `once_from_to(A,B,C)` est faux chaque fois que `C` survient après la dernière occurrence de `B` sans que `A` soit intervenu entre temps. `cont_since(A,B)` est vrai si on a continuellement `A` après que `B` ait été vrai (quand `B` est vrai, `A` peut être faux). `cont_from_to(A,B,C)` est vrai si on a continuellement `A` entre une occurrence de `B` et une occurrence de `C`. `never_between_and(A,B,C)` est vrai si `A` n'est jamais vrai entre un instant où `B` est vrai, et un instant ultérieur où `C` est vrai. `alternate(A,B)` est vrai si les instants où `A` est vrai et ceux où `B` est vrai se succèdent.

Des opérateurs qui comptent le nombre d'instants où une certaine configuration est vraie sont également très utiles.

```
node monostable(event,start     :bool; cpt:int) returns (on:bool);
node bistable  (event,start,stop:bool; cpt:int) returns (on:bool);
```

`monostable(event,start,cpt)` devient vrai chaque fois que `start` est vrai, et reste vrai jusqu'à la `cpt`-ième occurrences pas nécessairement consécutives de `event`. `bistable(event,start,stop,cpt)` est identique, excepté qu'il devient faux à chaque occurrence de `stop`.

3 Morceaux choisis des expérimentations illustrant la démarche

Une description des travaux effectués sur les cas d'étude abordés dans le projet COMON est présentée dans les sections annexes. Dans cette section, nous en avons extrait (parfois en simplifiant les noms des variables) quelques exemples illustrant la démarche présentée en section 1.

3.1 Formalisation des attendus

Nous allons illustrer maintenant le passage de l'étape 2 à l'étape 3 de la section 1.2, c'est-à-dire comment formaliser en Lustre une spécification exprimée en langue naturelle ou à l'aide de dessins informels. Nous avons choisi un exemple par cas d'étude. Ces exemples illustrent la démarche proposée dans le cadre de tests unitaires, de tests du système complet, et de tests de niveaux intermédiaires.

3.1.1 Un test unitaire au N1 classé

Parmi la liste des exigences fonctionnelles exprimées en langue naturelle dans le cas d'étude fournie par Rolls-Royce (annexe A), il y avait celle-ci :

> La sortie positionnant l'actionneur (`EU01`) doit être vraie si et seulement si :
>
> 1. l'opérateur en a fait la demande (l'entrée `Manual_EU01`).
> 2. et/ou la pression (`P`) dépasse sa valeur maximale alors que la température maximale (`T`) a été dépassée

La première condition est facile à formaliser, car elle ne dépend que de l'entrée `Manual_EU01`. La seconde semble moins directe, mais le système produit en sortie la variable booléenne `HzPsMax2_et_MemoTsMax1` qui correspond exactement à cette définition. Nous pouvons donc formaliser cette propriété sous forme d'un oracle qui vérifie que la sortie de l'actionneur est bien calculé ainsi :

```
node check_EU01(P,T:real;HzPsMax2_et_MemoTsMax1,EU01:bool) returns (ok:bool);
 ok = if (HzPsMax2_et_MemoTsMax1 or Manual_EU01_valeur) then EU01 else not EU01;
```

En terme de couverture de cet oracle, les 3 cas intéressants sont :
- quand `HzPsMax2_et_MemoTsMax1` est vrai,
- quand `Manual_EU01_valeur` est vrai,
- ou quand les 2 sont faux.

Ainsi si on écrit l'oracle précédent ainsi :

```
node check_EU01(P,T:real;HzPsMax2_et_MemoTsMax1,EU01:bool)
returns(ok,C1,C2,C3:bool);
  C1 = HzPsMax2_et_MemoTsMax1;
  C2 = Manual_EU01_valeur;
  C3 = not (C1 or C2);
  ok = (if (C1 or C2) then EU01 else not EU01);
```

Lurette pourra calculer la couverture associée à cet oracle.

Il est également intéressant de vérifier que la sortie `HzPsMax2_et_MemoTsMax1` est calculée correctement. Pour établir si température maximale (`T`) a été dépassée, on peut utiliser l'interrupteur `switch` de la section 2.7. `switch(false,C,false)` est faux à l'origine des temps, devient vrai au premier instant où `C` est vrai, puis reste vrai pour toujours. Ainsi on peut vérifier que `HzPsMax2_et_MemoTsMax1` est calculée conformément à sa spécification ainsi.

```
node check_HzPsMax2_et_MemoTsMax1(P,T:real;HzPsMax2_et_MemoTsMax1:bool)
returns(ok:bool);
var
  TMax1_franchit:bool;
```

```
let
  TMax1_franchit = switch(false,T>Max1,false);
  ok = (HzPsMax2_et_MemoTsMax1 = TMax1_franchit and P>Max2);
tel
```

En terme de couverture, on veut au moins voir `HzPsMax2_et_MemoTsMax1` passer à vrai et à faux. Éventuellement, il peut être intéressant de vérifier un troisième cas correspondant au cas où `P` dépasse `Max2` avant que `T` ne dépasse `Max1`, ce qui peut s'exprimer par :

```
C3 = P>Max2 and not TMax1_franchit;
```

3.1.2 Vérification d'une alarme au Niveau 2

Voici une propriété classique du niveau 2 et issu du SPF1, et de plus haut niveau que la précédente (cf annexe D).

> Quand un seuil haut est franchi pour la valeur `v`, le niveau 2 doit afficher l'alarme correspondant au bout de 5 secondes, et réciproquement.

En Lustre, cette propriété peut se formaliser ainsi :

```
ok1 = est_suivi_de(5.0, r_edge(v>seuil_haut), r_edge(alarm));
ok2 = est_suivi_de(5.0, f_edge(alarm), f_edge(v>seuil_haut));
```

où `r_edge` et `f_edge` sont des nœuds qui détectent les fronts montants et descendants de variables booléennes (cf. section 2.7), et où `est_suivi_de(5.0,e1,e2)` est vrai si et seulement si quand `e1` devient vrai, `e2` devient dans les 5 secondes (cf. section 2.4). On vérifie donc à la fois qu'un dépassement de seuil est suivi d'une alarme dans les 5 secondes, et que quand le seuil haut n'est plus dépassé, l'alarme retombe dans les 5 secondes.

Pour couvrir cet oracle, il suffit de voir passer un front descendant de l'alarme (`f_edge(alarm)`). En effet, si on voit passer un tel front descendant sans invalider l'oracle ci-dessus, c'est bien que les fronts montants et descendants des 2 signaux booléens concernés ont eu lieu en temps et en heure.

3.1.3 Surveillance de la stabilité du système complet

Voici une autre propriété exprimée en langue naturelle dans le SPF1 (annexe D) :

> En mode nominal, après tout changement de consigne, toutes les valeurs issues des capteurs doivent être stables au bout de cinq minutes.

Une façon de formaliser cette propriété en Lustre est la suivante :

```
C = vrai_depuis_n_secondes(300.0, aucun_chgt_consigne and nominal);
ok = (C => est_stable);
```

Bien entendu, cette formalisation est aisée et concise car nous supposons que les nœuds qui (1) calculent si une variable est vraie depuis un certain temps, et (2) calculent si une variable est stable existent et sont corrects. En l'occurrence, les nœuds `vrai_depuis_n_secondes` ainsi que celui qui calcule la valeur de la variable `est_stable` sont dans notre ébauche de bibliothèque et sont détaillés en section 2.2 et 2.3 respectivement.

Concernant la couverture de cet oracle, on peut remarquer qu'il y deux cas de figure pour que l'oracle surveillant la stabilité du système soit tout le temps satisfait.

1. Tout d'abord, on peut rendre la prémisse de l'implication fausse, par exemple en changeant de consigne à chaque cycle.
2. Une autre façon est de mettre en œuvre un système qui réellement se stabilise en mode nominal quand aucun changement de consigne n'a lieu pendant cinq minutes.

Du point de vue du testeur, c'est le deuxième cas qui est intéressant. Ainsi on définit la couverture associé à cet oracle par la variable C. Pour couvrir ce critère, il faudra générer une séquence de test où la consigne ne change pas toutes les secondes. Nous montrons comment spécifier un stimulateur générant ce type de séquences en section 3.2.

3.2 Formalisation de l'environnement

Nous allons illustrer à nouveau le passage de l'étape 2 à l'étape 3 de la section 1.2, mais en nous focalisant cette fois sur l'environnement du système à tester.

3.2.1 Exploiter les contraintes pour explorer tous les cas réalistes

Pour illustrer comment extraire d'une spécification informelle des contraintes permettant d'explorer pseudo-aléatoirement un grand nombre de comportements réalistes de l'environnement d'un système, nous nous proposons de décrire un générateur de pannes conçu pour stimuler l'étude de cas COMON (annexe D).

Il s'agit d'un stimulateur dont l'objectif est de générer `n` pannes sans déclencher d'action classée. Une action classée doit être déclenchée par exemple si 2 éléments redondés sont en défaut, ou bien si 3 éléments quadri-redondés sont en défaut.

Considérons pour simplifier le cas d'un capteur redondé et d'un capteur quadri-redondé, dont les pannes sont associées aux 6 variables booléennes `P1`, ..., et `P6`. Par défaut en Lutin, une variable non contrainte est choisie au hasard. Ainsi le nœud suivant choisit au hasard des valeurs pour chacune des 6 pannes possibles :

```
node gen_pannes() returns (P1,P2,P3,P4,P5,P6:bool) =
 loop { true }
```

Pour vérifier certains aspects du fonctionnement du système sous test en cas de pannes, il peut être intéressant de rendre un peu moins aléatoire ce stimulateur en l'empêchant de changer de pannes à chaque cycle. Pour modifier le stimulateur précédent pour le forcer à ne changer de pannes qu'au bout d'un temps compris entre 2 et 5 minutes, il suffit de remplacer `loop true` par :

```
loop {
 true -- on choisit des pannes
 fby loop [minutes(2),minutes(5)] -- puis on les conserve quelques temps
    P1=pre P1 and P2=pre P2 and P3=pre P3 and P4=pre P4 and P5=pre P5 and P6=pre P6
}
```

Notez qu'en Lutin, pour obliger une variable `P` à garder sa valeur précédente, on doit explicitement écrire la contrainte (`P=pre P`). On utilisera souvent par la suite le combinateur suivant pour faciliter la définition de telles contraintes.

```
let maintient(x: bool ref) = (x = pre x)
```

Quand une action classée est déclenchée, le système est mis dans une position sûre. Pour pouvoir tester le système en fonctionnement nominal, il peut être intéressant de déclencher des pannes tout en évitant le déclenchement d'actions classées. Pour cela, il suffit de formaliser les conditions qui doivent provoquer ces actions, par exemple à l'aide des nœuds `au_moins_un_parmi_deux` et `au_moins_trois_parmis_quatre` décrits en section 2.1.

```
let action_classee = au_moins_un_parmi_deux(P1,P2) or
                     au_moins_trois_parmis_quatre(P3,P4,P5,P6)
in
```

Ensuite, il suffit de rajouter cette contrainte au générateur de pannes.

```
loop (not action_classee)
```

On peut aussi vouloir définir des sessions de test plus interactives, qui laissent à l'utilisateur (ou à un autre stimulateur l'utilisant) le soin de spécifier le nombre de pannes. Pour cela, il suffit d'ajouter en entrée de ce nœud un entier `n` spécifiant le nombre de pannes à générer, puis de contraindre ce stimulateur à générer exactement `n` pannes.

```
let b2i(b:bool) = if b then 1 else 0 in
let nb_pannes = b2i(P1)+b2i(P2)+b2i(P3)+b2i(P4)+b2i(P5)+b2i(P6) in
loop {
  (not action_classee and nb_pannes = n) |> nb_pannes = n
}
```

Notez l'utilisation de l'opérateur Lutin « |> » qui permet de demander prioritairement d'essayer une contrainte plutôt qu'une autre. Ici, on essaie de générer n pannes sans déclencher d'action classée. Bien sûr, si « n » est plus grand que 5, la contrainte « `not action_classee and nb_pannes = n` » est insatisfiable. Dans ce cas on se contente de choisir des valeurs qui satisfont la contrainte « `nb_pannes = n` ».

Si on résume l'ensemble des évolutions du nœud `gen_pannes`, et que l'on rajoute la possibilité de redéclencher le choix de la panne via une entrée supplémentaire « `reset` », on obtient le stimulateur suivant :

```
node gen_pannes(n:int; reset:bool) returns(P1,P2,P3,P4,P5,P6:bool) =
let action_classee = au_moins_un_parmi_deux(P1,P2) or
                     au_moins_trois_parmis_quatre(P3,P4,P5,P6)
in
let b2i(b:bool) = if b then 1 else 0 in
let nb_pannes = b2i(P1)+b2i(P2)+b2i(P3)+b2i(P4)+b2i(P5)+b2i(P6) in
loop {
 { (not action_classee and nb_pannes = n) |> nb_pannes = n }
 fby loop [minutes(2),minutes(5)] {
       not reset and maintient(P1) and maintient(P2) and maintient(P3)
                 and maintient(P4) and maintient(P5) and maintient(P6)}
}
```

3.2.2 Écrire des scénarios pour améliorer la couverture

L'exemple précédent illustre la capacité de Lutin à générer aléatoirement des valeurs de variables à stimuler, tout en respectant certaines contraintes permettant de générer des séquences réalistes, ou intéressantes. Nous allons maintenant décrire un exemple illustrant la capacité du langage à exprimer des scénarios de test plus élaborés. En effet, si l'on veut couvrir des critères du style « quand aucune consigne ne change pendant 2 minutes alors ... », il faut explicitement concevoir des scénarios où aucune consigne ne change pendant 2 minutes. Nous nous proposons de programmer un opérateur (pseudo-)aléatoire qui change la consigne en surveillant la valeur de capteurs de niveaux : la procédure lui impose de changer la consigne pas à pas, en attendant à change fois que le système se stabilise.

Plus précisément, nous allons modéliser le comportement d'un opérateur qui : choisit une consigne Cible dans l'intervalle [0 ; 100] (phase 1) ; utilise un nœud qui va amener la consigne à la cible pas à pas (phase 2) ; quand la cible est atteinte, nous retournons dans la phase de choix d'une cible à atteindre (phase 1).

```
let between(x, min, max: real): bool = (min <= x) and (x <= max)
node operateur(C_init:real; est_stable:bool) returns(C,Cible:real;phase:int)=
 phase=0 and C=C_init
 fby
 loop {
     phase=1 and -- Choisit la prochaine consigne cible
     between(Cible, 0.0, 100.0) and
     maintient(C)
   fby -- Vise la cible pas a pas
     assert phase=2 and maintient(Cible) in
     run C := change_consigne_pas_a_pas(est_stable,pre C,pre Cible,2.0) in
       while (C <> Cible)
  }
```

Le nœud `change_consigne_pas_a_pas` est du même style. Il maintient la consigne tant qu'on n'a pas la stabilité ; se rapproche de la consigne cible d'au plus « `pas` » ; si la consigne cible n'est pas atteinte, on recommence au début.

```
node change_consigne_pas_a_pas(est_stable:bool; C,Cible,pas:real)
returns (newC:real) =
 loop {
   loop { not est_stable and newC = C } -- On attend la stabilite
   fby
     if Abs(Cible - C) < pas then newC = Cible
     else if Cible < C       then C - pas < newC and newC < C
     else                         C < newC and newC < C + pas
   fby
   loop [secondes(10)] { newC=C } -- on attend que le chgt de Consigne prenne effet
 }
```

Cette modélisation illustre la capacité de Lutin à exprimer des scénarios assez évolués permettant d'améliorer la couverture.

3.2.3 Un stimulateur du Niveau 2

Voici maintenant un stimulateur issu d'une étude de cas Niveau 2 (annexe C). L'objectif est de choisir au hasard des valeurs pour les coordonnées d'un point dans un domaine numérique en deux dimensions. Ce domaine est divisé en 4 zones, définies ainsi :

```
let zone1(x,y:real):bool = between(x,20.0, 80.0)  and between(y,0.0,100.0)
let zone2(x,y:real):bool = between(x,-10.0,100.0) and between(y,0.0,.0)
let zone3(x,y:real):bool = between(x,-10.0,100.0) and between(y,0.0,45.0)
let zone4(x,y:real):bool = between(x,0.0,  90.0)  and between(y,0.0,40.0)
```

Le choix de la zone est contrôlé par 2 entrées : `zone_elue`, qui provient d'une élection effectuée par ailleurs, et `zone_forcee` qui correspond à une zone choisie (forcée) par l'opérateur. Quand la `zone_forcee` vaut 0, c'est la `zone_elue` qui doit être utilisée ; sinon on utilise `zone_forcee`.

```
node gen_point_dans_zone(zone_elue,zone_forcee:int) returns (x,y:real) =
  exist zone_choisie:int in
  assert zone_choisie = if zone_forcee = 0 then zone_elue else zone_forcee in
  loop {
    if zone_choisie = 0 then zone1(x,y) else
    if zone_choisie = 1 then zone1(x,y) else
    if zone_choisie = 2 then zone2(x,y) else
    if zone_choisie = 3 then zone3(x,y) else
    if zone_choisie = 4 then zone4(x,y) else raise Error
  }
```

Ici, le choix à l'intérieur de chaque zone est effectuée dans toute la zone choisie, de façon similaire à ce qui est fait par le nœud `gen_x_v1` de la section 2.6. Nous pourrions également utiliser l'une des trois autres façons proposées dans cette section 2.6. La `zone_elue`, quant à elle, est choisie au hasard toutes les 5 minutes.

```
node election_de_la_zone() returns (Zone:int) =
  loop {
    between(Zone, 1, 4) fby loop ~minutes(5) { maintient(Zone) }
  }
```

A Étude de cas Scade (RRCN)

expe_rrcn.pdf

B Étude de cas Alices (CORYS)

expe_objet_procede.pdf

C Étude de cas Scada

expe_scada.pdf

D Étude de cas COMON

expe_comon.pdf