Science virtuelle

Gianni Mocellin

Straco
www.straco.ch
21.03.2021, 05h00

Introduction ... ***6***

Le simulateur ... ***6***

Les entités ... **7**

Les modèles ... **8**

Les unités ... **11**

Les types dérivés ... **11**

Les équations ... **11**

La variabilité temporelle ... **13**

La variabilité continue ... *13*

La variabilité discontinue ... *13*

Les états durables ... 14

Les interactions ... **16**

Les connecteurs ... **20**

Les connexions ... **20**

Le préfixe flow ... *21*

Les équations de connection ... *21*

Les connexions causales ... *22*

Les connexions composites ... *22*

Les coordonnées internes ... **22**

Le sens mécanique ... *23*

Le sens électrique ... *26*

Partial ... 26

Les idées discontinues ... **28**

Les prévisions ... **28**

Les réalisations ... **28**

Le langage artificiel ... **29**

Les spécimens ... **33**

Création d'un spécimen ... *33*

Initialisation d'un spécimen ... *33*

Les types prédéfinis ... **34**

Héritage entre entités ... *35*

Les notes ... *35*

Les préfixes ... **37**

Les préfixes d'accessibilité ... *37*

- publique ... 37

- protégée ... 37

Les préfixes de variabilité temporelle des idées ... *37*

Générale ... 37

Constantes ... 38

Paramètres ... 38

Discontinue ... 39

Les préfixes de causalité ... *39*

entrée ... 40

sortie ... 40

Les préfixes de modification ... *40*

remplaçable ... 40

redéclarer ... 40

finale ... 40

Les types ... **41**

Le type "proportionnée" ("Real") ... *42*

Attribut "nom" ("name") ... 42

Attribut "valeur" ("value") ... 42

Attribut "substance" ("quantity") ... 43

Attribut "unité" ("unit") ... 43

Attribut "valeur minimale" ("min") ... 43

Attribut "valeur maximale" ("max") ... 43

Attribut "valeur initiale" ("start") ... 43

Attribut "valeur fixée" ("fixed") ... 44

Attribut "valeur nominale" ("nominal") 44
Attribut "réalité obligée" ("stateSelect") 45
Le type "Echelonné" ("Integer") 45
Le type "Douteux" ("Boolean") 46
Les type "Textuel" ("String") 46
Le type "Enumérable" ("enumeration") 46
Le type "Cadenceur" ("Clock") 47
Les entités probables 48
Les équations conditionnelles (si-équations) 48
Les équations instantanées (quand-équations) 48
Les cadenceurs synchrones (Clock) 50
Les machines à état 50
Comportements continus dans des partitions temporelles 50
Les équations 52
Les équations initiales 52
Les équations conditionnelles 52
Les critères 59
Les entités instantanées 60
Les entités continues 60
Les entités cadencées régulières 60
Les entités incertaines 60
Les entités conditionnées 61
Les entités continues 61
Les entités constantes (constant) 62
Les entités paramètres (parameter) 62
Les entités discontinues (discrete) 63
Les entités instantanées 63
Les entités durables 63
La genèse des signaux 66
Les signaux prévisionnels 66
Les entités cadencés 67
Les signaux comportementaux 68
Les signaux prédéfinis 68
initial() 69
terminal() 69
terminate(message) 70
sample() 70
Clock() 71
sample(...) 71
sample(entité, cadence) 71
hold(...) 72
subSample(), superSample(), shiftSample() 72
interval() 72
noEvent() 72
smooth(...) 72
pre(...) 72
previous(...) 72
edge() 72
delay() 72
Le traitement des signaux 72
Réflexion discontinue aléatoire versus réflexion cadencée 73
Définition de quand-équations incertaines par des si-équations 74
Changements discontinus à l'apparition de signaux 75
Usage de la priorité des signaux pour éviter les définitions multiples 75
Synchronisation des signaux et propagation 75
Signaux multiples au même instant et itération de signaux 76
Le modèle séquencé 76
Les paquets 77
Organisation 77

Référencement 78
Importation 79
Caveat 81
Résumé 82
Définition des paquets 82
Enregistrement 83
Ordre des fichiers 84
Versioning 85
Resources 86
Règles de consultation 86

Architectures 87

Bibliothèque standard 88

Dictionnaire 88
- A - 88
Analyse 88
Analysis 88
Analyse fondamentale 88
Analyse rationnelle 88
Analyse sectorielle 88
Analyse technique 88
Assignation 88
Assignation 88
Attributs 89
Attributes 89
- B - 89
- C - 89
Cadence 89
Cadence 89
Considération 89
Consideration 89
Contact 89
Contact 89
- D- 89
Déclaration 89
Declaration 89
- E - 89
Equation 90
Equation 90
Expression 90
Expression 90
- F - 90
Fonction 90
Function 90
- G - 90
- H - 91
- I - 91
Idée 91
Idea 91
Initialiser, initialisation 91
Initialize, initialization 91
Instancier, instanciation, instance 91
Instanciate, instanciation, intstance 91
- J - 91
- K - 91
- L - 91
- M - 91

Modifier, modification **91**
Modify, modification *91*
- N - **92**
- O - **92**
- P - **92**
- Q - **92**
- R - **92**
- S - **92**
- T - **92**
- U - **92**
- V - **92**
- W - **92**
- X - **92**
- Y - **92**
- Z - **92**

Introduction

Le simulateur

GIANNI

Après avoir fait un petit tour des investisseurs, on pourrait faire un petit tour de ce qu'il y a dans un simulateur.

JEAN

D'accord.

JEAN et GIANNI face à un simulateur.

GIANNI

Un simulateur est un outil qui permet de représenter la réalité sous forme de système.

Il doit donc permettre de représenter les deux idées fondamentales que sont:

- les entités, d'une part, et,

- leurs connexions, d'autre part.

Le modèle est composé d'entités et de connexions acausales entre elles, c'est-à-dire encore que la causalité est initialement non précisée dans les connexions entre les entités.

Les entités ont des connecteurs bien définis avec le monde extérieur et doivent être définies de manière totalement indépendante de leur monde extérieur.

Cela signifie que la définition d'une entité, y compris ses comportements, ne peut utiliser que de l'information interne ou des informations provenant de ses connecteurs.

Aucun moyen de communication entre une entité et le reste du système, avec les autres entités et l'environnement, n'est admise à part à travers ses connecteurs.

L'intérieur d'une entité peut lui-même être composé d'autres entités selon une structure hiérarchique.

Les entités

Pour m'y retrouver quand je construis un simulateur, je commence en général par classer les idées en trois grands groupes de base, en utilisant comme critère

"*leur variabilité dans le temps du simulateur*"

Pour ce faire, j'utilise des préfixes que je place devant le nom de l'idée, comme "constante" ou "paramètre", par exemple.

J'obtiens ainsi les trois grands groupes d'idées suivants:

- les idées dont la variabilité doit être considérée comme générale par rapport au temps du simulateur, des idées que je qualifie de

"*générales*",

c'est-à-dire des idées qui peuvent varier n'importe quand dans le temps d'une simulation.

- les idées que je considère comme

"*constantes*",

c'est-à-dire des idées ni ne peuvent pas varier selon le temps du simulateur ni être variées par l'utilisateur d'une simulation à l'autre.

Seul le modélisateur a accès à ces idées dites "*constantes*", pas l'utilisateur.

Seul le modélisateur peut modifier le nombre de ces entités, en les créant ou les éliminant.

L'utilisateur n'y a pas accès pour des raisons de sécurité.

Le simulateur doit en effet répondre à un cahier des charges précis, une espèce de contrat résultant d'une négociation entre l'utilisateur et le modélisateur, et seul le modélisateur peut modifier le modèle qu'il construit en fonction des besoins de l'utilisateur.

Ces idées constantes ne peuvent donc pas être modifiées

"*d'une simulation à l'autre*"

par l'utilisateur.

Elles sont fixes dans le modèle.

- les idées que je considère comme

"*paramètres*".

Ce sont des idées dont la valeur ne peut varier selon le temps du simulateur mais qui peuvent être variées par l'utilisateur

"*entre deux simulations*"

pour qu'il puisse tester les effets dans le temps de variations éventuelles de telles idées.

L'utilisateur doit avoir un accès très facile à ces "*paramètres*", qu'il doit pouvoir modifier à sa guise entre deux simulations.

La grande différence entre les idées générales et les idées paramètres est que ces dernières varient beaucoup plus lentement que les premières.

Si on fait un parallèle avec la biologie, on pourrait dire que la longueur des ailes d'un oiseau est un paramètre, qui varie très lentement dans le temps, alors le battement des ailes est une variable générale qui peut varier très rapidement dans le temps.

La théorie de l'évolution, par exemple, essaye de comprendre les variations des paramètres d'un organisme, comme la longueur des ailes, par exemple, alors que la théorie du comportement essaie de comprendre des variations beaucoup plus rapides, comme les battements d'ailes, par exemple.

Les modèles

Pour déclarer un modèle au simulateur, je dois utiliser le langage du simulateur.

Pour bien distinguer les mots de la langue artificielle du simulateur des mots de la langue naturelle que nous utilisons entre nous, je les mets dans une autre police de caractères et en gras.

Pour déclarer un modèle au simulateur je dois donc:

- utiliser le mot **`model`** pour lui dire que ce qui suit est un modèle,

- donner un nom au modèle,

- utiliser le mot **`end`** suivi du nom du modèle pour lui dire que c'est la fin du modèle, et,

- terminer le tout par un point-virgule.

Je peux en outre mettre des commentaires entre apostrophes.

Ces commentaires sont uniquement destinés au modélisateur et ne font pas partie du modèle lui-même.

Le modèle le plus simple concevable est celui d'une entité.

```
model Entity1 "Ce modèle est celui d'une entité appelée Entity1"
end Entity1;
```

Ensuite, je peux dire au simulateur que l'entité en question est caractérisée par un état:

```
Real state1 "state1 est un état de type proportionné de l'entité Entity1";
```

Ce qui donne comme modèle:

```
model Entity1 "Ce modèle est celui d'une entité appelée Entity1"
Real state1 "state1 est un état de type proportionné de l'entité Entity1";
end Entity1;
```

Ensuite, je peux dire au simulateur que cette entité a un comportement représenté par une équation en utilisant le mot **`equation`** et lui dire que cette équation représente une variation temporelle en utilisant le mot **`der`**:

```
equation
der(state1) = 1-state1 "équation représentant la variation de state1";
```

Ce qui donne comme modèle:

```
model Entity1 "Ce modèle est celui d'une entité appelée Entity1"
Real state1 "state1 est un état de type proportionné de l'entité Entity1";
equation
der(state1) = 1-0.05*state1 "équation représentant la variation de
state1";
end Entity1;
```

Enfin, je peux dire au simulateur que l'entité avait une valeur initiale:

```
initial equation
state1 = 2 "Valeur à utiliser pour state1 au début de la simulation";
```

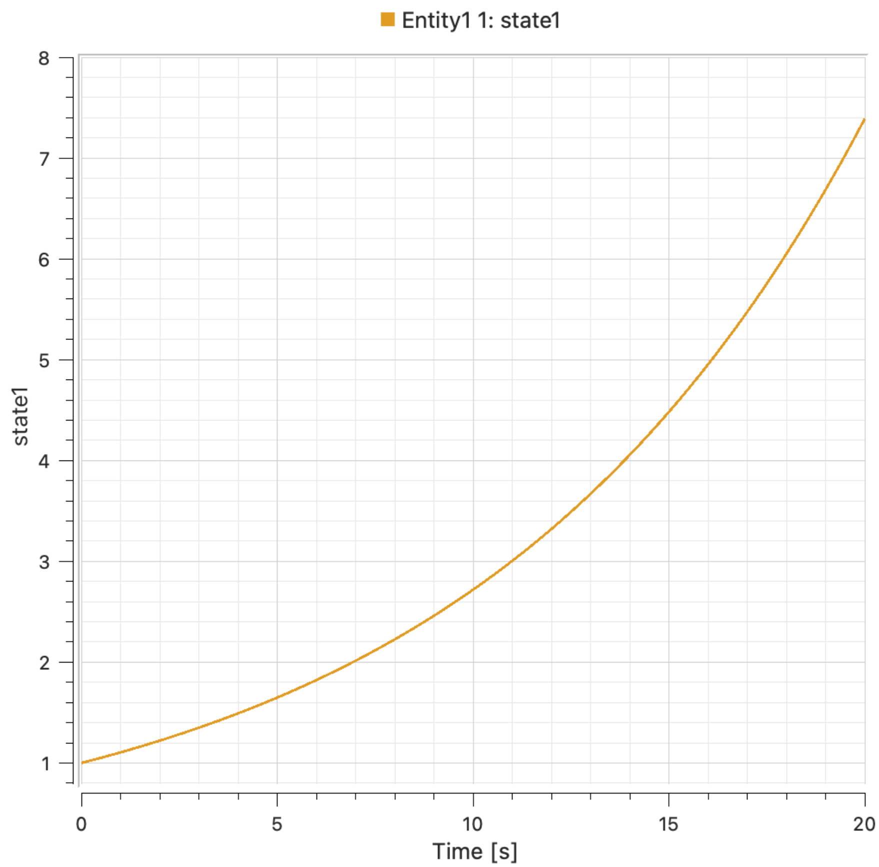
Ce qui donne comme modèle:

```
model Entity1 "Ce modèle est celui d'une entité appelée Entity1"
Real state1 "state1 est un état de type proportionné de l'entité Entity1";
equation
der(state1) = 1+0.10*state1 "intérêt composé à 10% de state1";
initial equation
state1 = 1 "Valeur à utiliser pour state1 au début de la simulation";
end Entity1;
```

Terminé.

Le modèle est complet.

Je peux immédiatement simuler le modèle de cette entité qui est en fait un investissement de 1 USD placé à un intérêt composé de 10% pendant 20 ans:



Simulation du modèle

Les unités

Les entités du modèle doivent correspondre à des entités du monde réel.

Il faut donc pouvoir associer une unité réelle à un état:

```
parameter Real state1(unit = "USD")= 1'000.000 "Capital";
```

L'instruction ci-dessus spécifie une valeur pour l'attribut unité de l'état yyy.

La fixation de l'attribut yyy est important pour deux raisons:

- la première est le simulateur définit des relations pour toutes les unités SI, y compris certaines unités composées;

- la seconde est que le simulateur possède des règles pour calculer les unités d'expressions complexes. Ainsi, le simulateur peut faire des vérifications d'unités et identifier des inconsistances dans les unités.

Les types dérivés

Pour simplifier le travail, on peut créer des "*types dérivés*" puisque le type `Real` peut être n'importe quoi.

```
type Capital = Real(unit="USD", min=0);
```

Ce type est une spécialisation du type prédéfini **Real**.

On peut ensuite utiliser ce type dans n'importe quelle déclaration:

```
parameter Capital state1 = 1'000.000 "Capital initial";
```

Ce type peut être utilisé n'importe où sans avoir à préciser son unité ni sa valeur minimale:

```
parameter Capital capitalToto = 1'000.000 "Capital de Toto";

parameter Capital capitalTiti = 2'000.000 "Capital de Titi";
```

Les équations

JEAN

En fait tu es en train de mettre mes idées sur les marchés en équations.

GIANNI

Je sais que ce mot de "*équation*" fait peur à beaucoup de gens mais le mot "*équation*" est un bien grand mot pour parler d'une idée très simple.

Une "*équation*" est simplement un moyen de relier des idées entre elles.

Par "*équation*" il faut simplement comprendre que c'est une ligne de texte qui contient un signe égal au milieu, comme dans la ligne ci-dessous:

=

La gauche de ce signe égal correspond à la main gauche.

La droite ce signe égal correspond à la main droite, et,

le signe égal correspond au nez, ou plutôt à la pensée,

qui met les deux en relation, puisqu'elle se situe forcément entre les deux mains,

qui met en relation ce qui est à gauche du signe égal avec ce qui est à droite du dit signe,

qui construit une nouvelle idée à partir des deux idées de gauche et de droite,

qui fait une nouvelle unité de pensée à partir des deux pensées de gauche et de droite.

On peut distinguer deux grands types d'équations:

- "*les équations d'affectations*"

qui permettent d'affecter une valeur à un état, c'est-à-dire d'affecter à gauche une valeur qu'on a dans la main droite;

- "*les équations de variation*"

qui permettent de représenter comment un état évolue dans le temps du simulateur, autrement dit de simuler un comportement. Ce sont les équations qui commencent par le préfixe `der`.

JEAN

Le temps du simulateur à l'air important dans toute cette affaire.

GIANNI

Il est fondamental.

Pour bien m'en sortir avec le temps, tu as vu que j'ai déjà utilisé plus haut l'idée de

"*variation*".

La variabilité temporelle

Les valeurs des états des entités peuvent varier dans le temps.

La "*variabilité*" est la liberté pour un état de

"*changer de valeur durant une simulation*".

JEAN

Pourquoi insistes-tu tellement sur "*la variabilité*"?

GIANNI

Par ce que, contrairement aux apparences, c'est une idée très complexe et fondamentale.

Si l'utilisateur et le modélisateur n'ont pas exactement la même idée de "*la variabilité*", des incohérences apparaitront immanquablement dans le modèle et donc dans les simulations.

En première approche, on peut considérer deux grands types de variabilité des états:

La variabilité continue

Les états dits

"*continus*"

peuvent changer de valeur à n'importe quel instant au cours du temps d'une simulation.

La variabilité discontinue

Les états dits

"*discontinus*"

ne peuvent changer de valeur qu'à un instant précis du temps de la simulation.

Par exemple, ces états dits "*discontinus*" peuvent prendre une valeur de manière instantanée s'ils représentent un fait se produisant à un instant précis prédéterminé ou encore faire une variation instantanée à l'instant du battement d'une cadence.

Ou encore varier instantanément à la suite du déclenchement d'un signal, d'une alerte ou d'une alarme, par exemple.

On peut faire une classification encore un peu plus fine des états discontinus, toujours grâce au même critère de la variabilité, en disant que certains états représentent des états instantanés et d'autres des états durables:

- les états instantanés

Ces états n'ont une valeur qu'à un certain instant précis du temps de la simulation.

Ils sont donc "*définis*" à un instant précis, et "*indéfinis*" ailleurs.

Hors de cet instant, ils sont donc inutilisables par le simulateur, c'est-à-dire tant "*avant*" que "*après*" le dit instant.

Ils ont

"*une variabilité instantanée*".

Les états durables

Ces états changent eux-aussi à un instant précis, mais, contrairement aux états instantanés, ils gardent la valeur qu'ils ont pris après la variation, jusqu'à leur prochaine variation éventuelle.

Ils ont

"*une variabilité durable*".

Je te rappelle que mon travail consiste à représenter la pensée des investisseurs dans un simulateur.

En fait le mot

"*modèle*"

n'est rien d'autre qu'un autre mot pour parler de

"*pensée*".

Et la pensée en question est dans la tête de l'utilisateur qui essaye de comprendre les investisseurs et non dans celle du modélisateur.

JEAN

Tout le secret d'un bon simulateur consiste à faire passer la pensée des investisseurs dans le simulateur en utilisant les connaissances de l'utilisateur.

GIANNI

Exactement.

Et dans cette affaire, certaines idées de base sont importantes.

Le mot

"*déclaratif*"

par exemple, est important.

JEAN

Pourquoi ce mot est-il si important?

GIANNI

Parce que mon travail consiste simplement à transformer

"*tes idées*"

que tu me présente sous forme de fiches, de tableaux et de graphiques dans le langage du simulateur.

Certaines de tes "*idées*" sont pour moi des "*déclarations*".

Ainsi, quand tu me dis:

"*Les actions ont une élasticité*"

c'est pour moi

"*une déclaration*"

car je dois "*déclarer*" au simulateur l'existence de l'entité "*actions*" et lui expliquer qu'elles ont "*une élasticité*".

D'autres réflexions dont tu me fait part sont pour moi des "*affectations*".

Ainsi quand tu me dis:

"*Le cours de l'or est de 1'600 USD*"

je dois "*affecter*" la valeur "*1'600*" à l'état "*cours*" de l'entité "*or*", tout en précisant que l'unité de mesure est le "USD".

représenter l'idée que tu te fais de l'or dans ta pensée dans le langage du simulateur.

Enfin, d'autres idées que tu m'exprimes sont pour moi des "*variations*".

Ainsi, quand tu me dis:

"*Le cours des actions varient en fonction de leur élasticité*"

je dois pouvoir représenter cette idée de "*variation*" du cours des actions dans le temps en fonction de leur élasticité.

Pour ce faire, j'entre dans le simulateur une équation puisque c'est de comportement dont il s'agit maintenant.

Je mets à gauche d'un signe égal la variation du cours de l'action dans le temps et à droite du signe égal ce qui explique ce changement.

En première approche, ce qu'on appelle

"*la modélisation*"

consiste donc à extraire de tes réflexions "*des déclarations*", "*des affectations*" et "*des variations*".

Les interactions

JEAN

Qu'en est-il des interactions entre les entités.

GIANNI

Pour représenter les-dites interactions, j'utilise une méthode que j'ai mis des années à mettre au point.

Elle a une caractéristiques fondamentale: quand je représente l'idée de

"*une interaction entre deux entités*"

je ne spécifie jamais le comportement de quelle entité est la cause du comportement de l'autre, autrement dit je ne précise jamais quel comportement est une cause ni quel comportement un effet.

Mes représentations des interactions entre les réalités sont

"*acausales*".

Cette manière de représenter la réalité est totalement différente des méthodes classiques de l'informatique, exigeant des égalités ayant toujours:

- des effets à gauche du signe égal, et,

- des causes à droite du dit signe égal,

reflétant un sens d'écriture des égalités de droite à gauche adopté par une grande majorité de l'humanité, le sens contraire de l'écriture normale, qui est de gauche à droite, ce qui pose beaucoup de problèmes quand il s'agit de comprendre les mathématiques.

Ces égalités causales ont donc une forme comme celle ci-dessous:

Effets à gauche = Causes à droite

Avec mes représentations "acausales", c'est le simulateur lui-même qui va trier les causes et les effets, ainsi que construire toutes les chaînes de causalité entre les causes et les effets.

Mon principe de base de modélisation consiste donc à transformer ce que tu me dis en

- des "*déclarations*", qui permettent de déclarer l'existence de certaines idées au simulateur;

- des "*affectations*" qui permettent d'affecter une certaine valeur aux idées, et en

- des "*variations*" qui permettent de représenter le comportement temporel des idées.

Pour moi les deux équations ci-dessous sont identiques:

Effets = Causes

Causes = Effets

C'est le simulateur qui va choisir la bonne en fonction des autres informations dont il dispose.

JEAN

Complexe.

Ce qui m'intéresse c'est de savoir comment tu peux représenter un marché financier dans un simulateur.

GIANNI

La causalité est fondamentale dans cette affaire. En effet, nous avons vu que la loi fondamentale de la finance, de la bourse en particulier, est

"*La loi de l'offre et de la demande*"

JEAN

"*Supply and demand law*"

en anglais.

GIANNI

Exactement.

Ce qui importe, c'est que je sois capable de traduire les raisonnements des investisseurs dans des représentations que le simulateur comprenne.

D'ailleurs, à propos de causalité, la loi de l'offre et de la demande est un excellent exemple dont on peut se faire une idée avec l'image suivante:

La loi de l'offre et de la demande

Les informaticiens classiques sont incapables de représenter la causalité d'un tel événement:

Est-ce "*plus de gâteaux*" qui cause "plus de fleurs", ou,

"*plus de fleurs*" qui cause "*plus de gâteaux*"?

Les modèles des informaticiens classiques sont "*algorithmiques*" et non pas "*déclaratifs*" comme les miens: ils représentent l'événement par une suite d'instructions données à un ordinateur et non par une série d'équations, ce qui impose une causalité dans leur modèles, la leur, et non pas celle de l'événement lui-même.

Mes modèles étant "*déclaratifs*", c'est le simulateur qui trouvera lui-même la causalité, puisqu'elle est implicitement contenue dans mon modèle de l'événement.

Ainsi, la représentation acausale de la pensée des investisseurs dans le simulateur lui permet déterminer lui-même les causes et les effets, contrairement aux représentations

"*statistiques*"

qui abondent sur internet et ne font qu'identifier des

"*corrélations*",

c'est-à-dire constater que certaines entités varient en même temps que d'autres sans pouvoir dire ni les variations desquelles sont des causes, ni les variations desquelles sont des effets.

Les connecteurs

Pour que les entités puissent interagir, elles doivent être connectées.

Les connecteurs du simulateur permettent de déclarer les transactions possibles à partir ou vers les entités qui composent le modèle.

Ils permettent de spécifier toutes les interactions possibles entre une entité, une entité opposée et l'environnement.

Seules des idées "*publiques*" peuvent être déclarées dans des connecteurs, afin que toutes les entités introduites dans le modèle puissent se connecter entre elles si elles ont le connecteur adéquat.

Les équations, fonctions ou algorithmes ne sont pas admises dans les connecteurs, puisqu'ils n'ont pas de comportements propres, leur rôle consistant simplement à matérialiser l'interface possible d'une interaction.

En bref, les connecteurs permettent aux entités d'interagir, de communiquer.

Et les connecteurs peuvent être très variés puisque pour le monde technique par exemple on a autour de 500 entités prédéfinies par les normes ISO, qui doivent pouvoir communiquer entre elles via des connecteurs

Les connexions

Deux types de connexions sont possibles dans le simulateur:

- des connexions "*explicites*" entre des paires de connecteurs, chacun des deux étant situé sur une entité différente, représentées par des lignes sur les graphiques du modèle et par des équations d'interaction dans le simulateur;

- des connexions "*implicites*" possibles entre une entité déclarée comme "*globale*" pour tout le modèle, avec laquelle toutes les autres entités présentes dans le modèle peuvent interagir.

Ce type de connexions implicites est utile quand trop d'interactions explicites seraient nécessaires, par exemple pour représenter un champ qui agit sur toutes les entités d'un modèle ou permettre l'accès à toutes les entités à un attribut propre à une entité particulière.

Le préfixe flow

Nous avons vu que deux idées simultanées sont implicitement contenues dans un connecteur acausal:

- l'idée de potentiel, une espèce de niveau énergétique ou vital, si tu préfères, qui n'a pas de préfixe particulier, et,

- l'idée de flot, préfixée par le mot clef `flow`, qui représentent le flot d'une certaine "*transité*" entre les deux entités: la force en mécanique, le courant en électricité, le flot de masse en hydraulique, par exemple.

Deux couplages sont simultanément établis par une connection.

- un couplage d'égalité pour les potentiels;

- un couplage de somme à zéro pour les flots, c'est à dire une idée de conservation de la transité qui transite entre les deux entités.

Par exemple:

. pour un système mécanique, la somme des forces qui transite en une position est nulle (conservation de l'impulsion),

. pour un système hydraulique, la somme des flots de masse qui transite en une position est nulle (conservation de la masse).

Pour assurer qu'un couplage à zéro soit toujours effectif pour les flots, il faut une règle consistante pour assigner la direction des flots, pour donner un signe aux flots qui passent par un connecteur.

Dans le simulateur, la valeur d'un flot est considérée comme positive quand le flot entre dans l'entité à laquelle le connecteur appartient.

Les équations de connection

Les connections entre les connecteurs sont représentées par des équations dans le simulateur.

Ces équations sont spécifiées par le mot clef **`connect`** et sont mises dans la section "*équations*" d'une entité, donc dans la section d'une entité qui traite de ses comportements.

Les **`connect-équations`** sont à considérer comme des équations malgré le fait que leur syntaxe soit différente des équations normales, puisque les `connect-équations` sont transformées en équations normales par le simulateur.

La forme générale d'une connect-équation est la suivante:

```
connect(connector1, connector2)
```

Les connexions causales

Dans les "*connections causales*", le sens d'information est supposé connu et spécifié par le modélisateur en utilisant les déclaration `input` ou `output` sur au moins l'un des connecteurs de l'entité.

Les connexions composites

Connexions composites sont des connexions structurées, composées de connexions plus simples.

Les coordonnées internes

De nombreuses entités ont une espèce de système de coordonnées internes auquel les connecteurs propres des entités doivent se référer.

Sur les graphiques de modèles, le sens des coordonnées est défini par convention entre deux connecteurs pour les entités simples à deux connecteurs, comme les résistances ou les ressorts, par exemple.

Un flot positif est propagé d'un connecteur à l'autre dans l'entité.

Autrement dit le flot entre dans un connecteur et sort par l'autre connecteur.

Ceci est valable en particulier pour les composants électriques, par exemple, car ils ne peuvent pas emmagasiner une charge nette.

Même une capacitance a des charges positives et négatives sur ses plateaux, c'est-à-dire pas d'accumulation de charge nette.

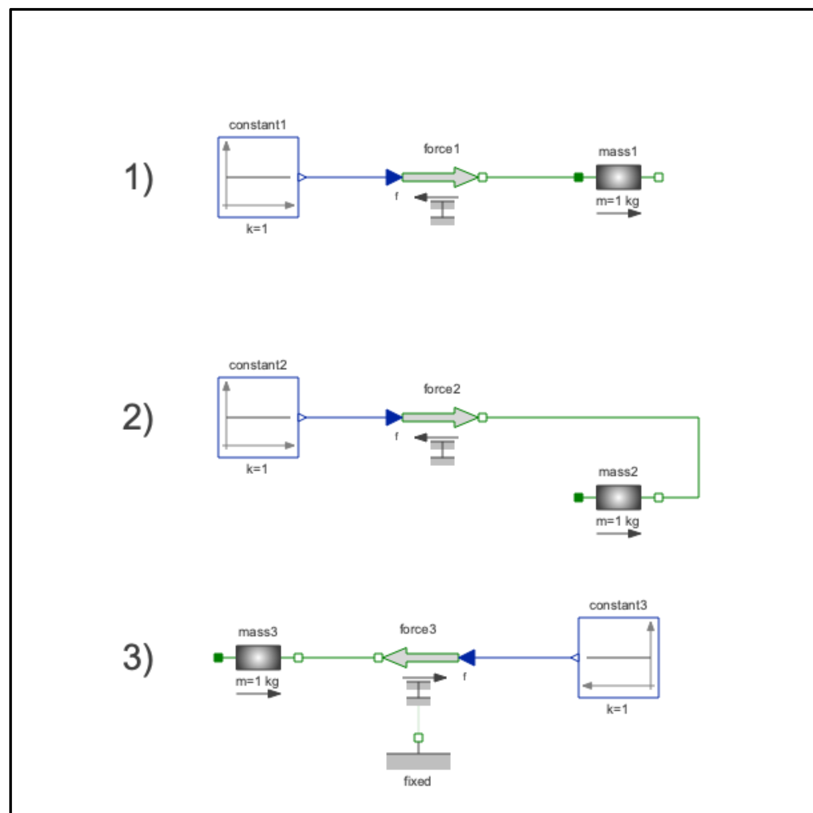
donc le flot entrant par le connecteur positif est égal au flot sortant par le connecteur négatif, et leur somme est nulle

$$p.i + n.i = 0$$

Le sens mécanique

Cependant cette convention n'est pas valable pour les entités mécaniques ayant une inertie puisque le flot entrant dans le connecteur a n'est pas égal au flot sortant par le connecteur b, la différence étant l'impulsion emmagasinée dans la masse:

$$a.f + b.f = m * der(v)$$



Connexions mécaniques

Si toutes les flèches pointent dans le même sens, une force positive provoque une accélération positive, une vitesse positive et un changement de position positif.

Les systèmes 1 et 2 sont équivalents. Cela n'a pas d'importance si la force pousse le connecteur a dans le système 1 ou tire sur le connecteur b dans le système 2.

Evidemment, il est possible d'ignorer les flèches et de connecter les entités de manière arbitraire. Mais il est ensuite difficile de voir dans quelle direction la force agit.

Dans le troisième système, les deux flèches sont opposées

Dans ce cas, le flot entrant dans le connecteur a est en général différent du flot sortant du connecteur b, de l'autre côté de l'entité, à moins que l'entité soit au repos ou n'ait pas d'inertie, comme un ressort, par exemple.

Un ressort sans inertie comprimé exerce une force de valeur absolue |f| depuis son connecteur sur une entité connectée à ce connecteur.

Les étiquettes "p" et "n" pour "positif" et "négatif" pour les connecteurs électriques et "a" et "b" pour les connecteurs mécaniques sont arbitraires.

On peut donc tourner une entité électrique et elle se comportera de la même manière: elle aura les mêmes valeurs de flot à travers elle.

Pour les entités mécaniques, la situation est un peu différente.

Pour une entité à une dimension, toutes les positions et forces sont relatives à un référentiel à une dimension global, à une dimension de référence.

Donc, si on tourne le ressort, la force f, qui est une quantité vectorielle dans un tel système de référence, change de signe.

Cependant la règle des valeurs des flots positifs pour les flots entrant dans l'entité est toujours valable, donc on utilise la valeur absolue |f| du vecteur représentant le flot.

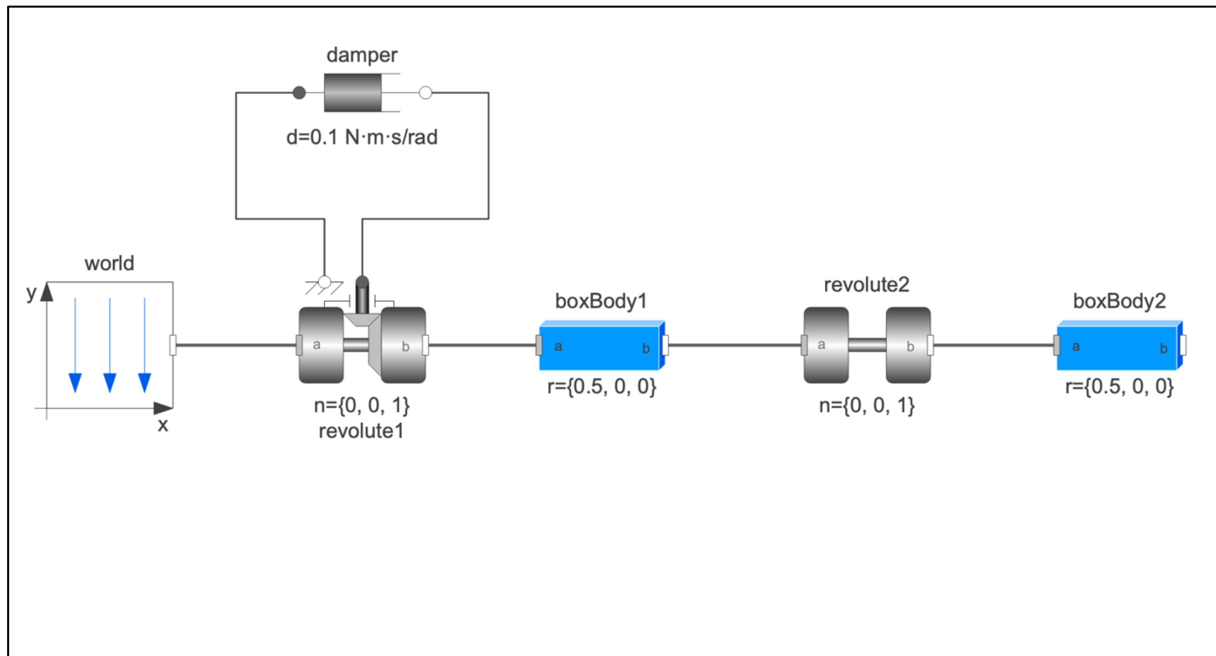
Les forces passant dans les connecteurs mécaniques peuvent être définies comme des vecteurs dans ce système de coordonnés global.

Le sens positif de la force dans une entité mécanique est indiqué par une flèche allant du connecteur a au connecteur b.

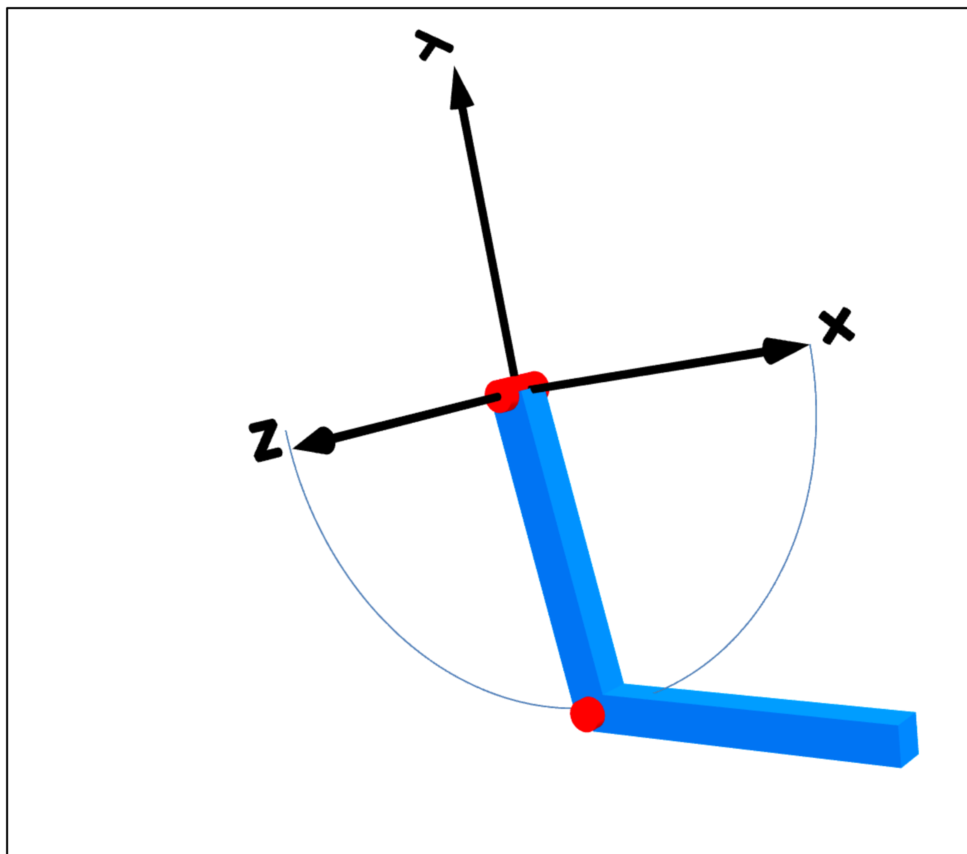
En ce qui concerne les connecteurs mécaniques, la convention est que le sens positif des flots, des forces donc, est toujours dans l'entité à laquelle le connecteur appartiennent, ce qui est indiqué par une flèche dans les connecteurs mécaniques.

Comme le sens positif pour le connecteur a est dans le sens opposé de celui du connecteur b, la valeur de la force du connecteur a est négative, -f c’est-à-dire moins la valeur absolue de la force, à savoir -|f|.

Les conventions de sens de flot dans les entités mécaniques est un peu différente de la convention de flot dans les entités électriques.



Modèle mécanique



Simulation mécanique

Le sens électrique

Partial

Une propriété commune des entités électriques est d'avoir deux connecteurs.

On peut donc concevoir un classe de bi-connecteurs

`BiConnector`

qui capture cette propriété.

C'est une entité est une entité "*partielle*" puisqu'elle ne contient pas assez d'équations pour spécifier un comportement. Elle sera donc préfixée par le préfixe `partial`.

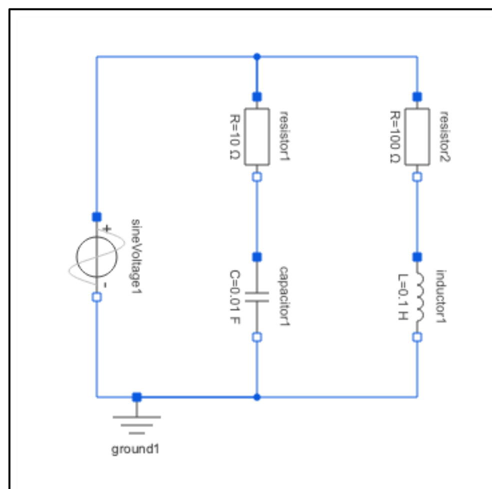
```
partial class BiConnector
    Connectors p, n;
    Voltage v;
    Courant i;
    equation
    v = p.v - n.v;
    0 = p.i + n.i;
    i = p.i;
end BiConnector
```

Le domaine électrique dispose de plusieurs entités qui étendent la classe d'entité `BiConnector`, disposant d'un connecteur positif et d'un connecteur négatif.

Le système de coordonnée ou le sens positif de la coordonnée est défini comme un flot allant de p vers n à travers l'entité.

Le sens positif du flot comme entrant par p et sortant par n est donc le sens positif du flot dans l'entité.

Les connecteurs `positive-connector` et `négative-connector` ont la même forme que la classe d'entité `Connector` sur les graphiques, si ce n'est leur couleur qui distingue les connecteurs positifs et négatifs.

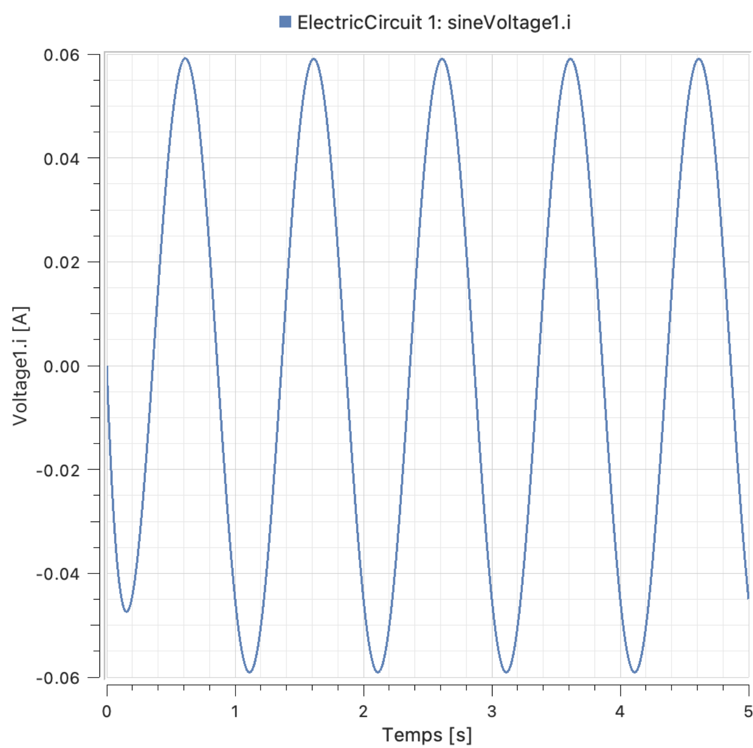


Modèle de circuit électrique

connector contact-positif = connecteur (flot p.i pénétrant dans contact-positif)

connector contact-négatif = connecteur (flot n.i pénétrant dans contact-négatif)

L'entité partielle partial de DeuxContacts.



Comportement du modèle du circuit

Les idées discontinues

JEAN

Qu'est-ce que tu entends par les idées discontinues?

GIANNI

Par discontinues je veux dire qu'il y a un saut dans la valeur de l'idée.

Lors d'une simulation, des faits se produisent sans cesse dans le simulateur, dont certains peuvent déclencher une discontinuité, provoquer un saut.

Les prévisions

Les faits les plus simples pouvant provoquer un saut sont ceux qui se produisent à un instant particulier connu.

On peut les appeler des prévisions, car ils sont liés au temps et le modélisateur sait quand ils vont se produire, par exemple des faits déclenchés par une cadence activée à une certaine fréquence.

Les prévisions sont très utiles en modélisation: le modélisateur peut, par exemple, vouloir modéliser la réponse du modèle à une certaine perturbation, comme une baisse de température par exemple.

Une autre manière simple d'utiliser les prévisions est d'initialiser la simulation dans une condition d'équilibre, ce qui permet de se concentrer sur la perturbation qui intéresse l'utilisateur.

Les réalisations

L'autre type de faits sont les réalisations, beaucoup plus complexes à modéliser.

La raison en est que le modélisateur ne sait pas à priori quand ces faits vont se réaliser.

Contrairement aux prévisions, le simulateur doit attendre qu'une entité traverse un certain état, un seuil, un critère, en d'autres mots qu'une condition soit satisfaite.

La détermination de l'instant précis auquel ces faits se réalisent est une tâche délicate pour le simulateur.

Pour déterminer cet instant on utilise une when-affirmation.

Ce qui fait que ces when-affirmations des faits de réalisation est que l'expression conditionnelle référence des variables autres que le temps.

```
when h<0 then
reinit(v, -e*pre(v));
end when;
```

Une when affirmation est composée de deux parties.

- la première partie est une expression conditionnelle qui indique l'instant de réalisation, le fait apparaitra quand la condition est satisfaite.

- la seconde partie est ce qui se produit quand le fait se réalise, souvent une réinitialisation, une espèce d'équation d'initialisation insérée au milieu d'un comportement, ne pouvant changer qu'une seule valeur contrairement aux initialisations qui peuvent en changer plusieurs.

Quand la valeur de la condition change durant un pas de calcul, le simulateur détermine l'instant exact du saut quand la valeur devient vraie et calcule l'état du système à cet instant, traite les instructions du corps de la when-affirmation, c'est-à-dire le reinit ici, qui affecte l'état du système, puis recommence à calculer après cet instant.

Le langage artificiel

GIANNI

On peut détailler un peu le langage artificiel pour voir de quoi il retourne.

JEAN

D'accord

GIANNI

Tu as vu que j'ai utilisé trois mots bien différents pour parler du modèle:

- les "*déclarations*";

- les "*affectations*", et,

- les "*variations*".

Ainsi, quand tu "*penses*" qu'il existe une certaine entité que tu appelles

"*Elasticité des actions*",

je dois la "*déclarer*" dans le simulateur.

Pour faire cette "*déclaration*", je commence par découper ta "*pensée*", ta "*réflexion*", ton "*idée*" en trois grandes parties:

- zéro ou plusieurs "***préfixes précisant certaines caractéristiques de l'entité déclarée***";

- suivis par un "***type de valeurs possibles que peut prendre l'entité déclarée***";

- suivi par un "***nom de l'entité déclarée***".

La structure d'une "*déclaration d'existence*" de l'une des entités constituant le marché, comme

"*L'élasticité*",

par exemple,

est donc la suivante dans le langage artificiel:

préfixes de déclaration de l'entité - type de valeurs possibles de l'entité - nom de l'entité

Pour l'élasticité, que je considère comme un paramètre du modèle, ayant une variabilité proportionnée, cela donne:

paramètre *-* ***proportionnée*** *- Elasticité*

Avec le préfixe "*paramètre*", je dis au simulateur de considérer

"*l'élasticité d'une action*"

comme

"*un paramètre du modèle*",

c'est-à-dire une entité dont

"*l'utilisateur*"

va pouvoir changer la valeur

"*entre deux simulations*",

pour voir l'effet d'une

"*variation d'élasticité*"

sur

"*le comportement d'une action*",

Je déclare ensuite au simulateur que

"*la valeur*"

de ce paramètre peut varier de façon

"*proportionnée*",

c'est à dire qu'elle peut varier par de toutes petites variations, par opposition à un type de valeur

"*échelonnée*",

qui ne peut varier que par sauts entre des valeurs entières, c'est-à-dire comme des sauts entre des nombres entiers ou entre des grades à l'armée, ou encore comme des évaluations d'obligations souveraines par des organismes d'évaluation, par exemple;

- je dis au simulateur que

"*le nom*"

de cette entité de type "*paramètre*", qui peut varier de manière "*proportionnée*" est

"*Elasticité*"

Quand je "*déclare*" ainsi au simulateur que l'entité "*Elasticité*" doit y exister, le simulateur ajoute immédiatement de lui-même, de manière automatique, cinq informations derrière le nom de la dite idée, ce qui donne la structure suivante pour la déclaration d'une entité proportionnée dès que j'en déclare l'existence, dans laquelle les cinq informations sont entre parenthèses:

préfixes - type de valeurs possibles de l'entité - nom de l'entité

(1 structure - 2 modifieurs - 3 équation de déclaration - 4 initialisation - 5 si-condition)

Les cinq informations, que je peux préciser lors de la déclaration d'existence d'une entité sont les suivantes:

1/5: l'information "*structure*" qui permet de déclarer si cette entité a "*une valeur simple*" ou une valeur elle-même constituée de "*une liste de valeurs*", à toujours considérer ensemble dans une simulation;

2/5: les informations "*modifieurs*" me permettent de dire que cette entité est une spécialisation d'une idée déjà déclarée autre part dans le simulateur.

Si tel est le cas, une telle entité héritera directement de toutes les caractéristiques de l'entité parente dont elle est une enfant.

En utilisant le modifieur "*étend*", par exemple, un "*modifieur*" particulier, je peux ajouter de nouveaux attributs et équations à une entité déjà existante.

Par exemple, je peux déclarer une entité

"*Banque*"

qui contient elle-même deux entités,

"*Actions*"

et

"*Obligations*",

toutes deux de type "*proportionnées*", c'est à dire que leur valeur peut varier de moins l'infini à plus l'infini, ce qui donne la déclaration suivante:

entité Banque
publique
proportionnée Actions;
proportionnée Obligations;
fin de Banque;

Une déclaration comme celle ci-dessus constitue un modèle qui décrit comment une entité "*Banque*" est structuré.

Le mot

"**publique**"

sert à spécifier le niveau de visibilité de cette idée depuis le reste du simulateur.

J'aurais pu me passer du préfixe "**publique**" puisque c'est la visibilité par défaut dans le simulateur.

L'autre possibilité aurait été d'utiliser le mot

"**protégée**"

qui impliquerait que seul l'intérieur de l'entité déclarée elle-même ainsi que toutes les entités qui en descendent peuvent accéder à ces informations.

Les spécimens

Création d'un spécimen

Disposant d'un modèle général de cette entité qu'est "*une banque*", le simulateur permet de créer directement des entités concrètes en "*déclarant*" simplement des spécimens de l'entité

"Banque"

comme existants concrètement dans le modèle.

Par exemple, pour créer un spécimen

"Banque1"

de type

"Banque", il me suffit de déclarer au simulateur

entité maBanque1
Banque maBanque1
fin de maBanque1;

Initialisation d'un spécimen

Par défaut, le simulateur met une valeur zéro dans l'attribut "*valeur*" du spécimen créé.

Si je veux, je peux mettre directement à la création du spécimen une valeur initiale d'actions et d'obligations, correspondant au temps zéro de la simulation:

entité maBanque1
Banque maBanque1(**initiale** 100'000.00, 200'000.00);
fin de maBanque1;

entité maBanque2
Banque maBanque2(**initiale** 500'000.00, 100'000.00);
fin de maBanque2;

Il m'est dès lors possible de créer très rapidement un spécimen d'une entité de type

"*Fortune*",

qui regroupe toutes les spécimens de banque constituant le spécimen de fortune:

entité maFortune
Banque maBanque1;
Banque maBanque2;
fin de maFortune;

Donc, pour déclarer un spécimen d'une idée générale

"*Fortune*",

je peux à choix:

- soit déclarer une idée "*Fortune*" comme une idée générale;

- soit déclarer directement un spécimen de "*Fortune*".

Il faut bien noter que chaque spécimen créé dans le modèle a une identité particulière.

Cela signifie qu'un attribut d'un spécimen d'une certaine entité est distinct d'un attribut ayant le même nom dans les autres spécimens de ce type d'entités créés.

Le mot "modèle" est en fait totalement similaire au mot "pensée" et ces mots sont complètement interchangeables.

Le concept de

"*pensée*"

est donc un concept clef de la modélisation.

Les types prédéfinis

Pour faciliter la vie du modélisateur, de moi-même donc, le simulateur propose un certain nombre de

"*types d'entités prédéfinies*"

ayant un certain nombre de

"*attributs prédéfinis*".

Seul l'attribut "*valeur*" de ces entités prédéfinies peut être modifié au cours d'une simulation et le simulateur peut accéder directement à cette valeur en utilisant simplement le nom de l'idée.

Ces attributs prédéfinis peuvent être changés lors de la déclaration d'une nouvelle entité.

Par exemple, je peux déclarer une entité

"*Capitalisation*"

Capitalisation = **proportionnée**(**unité**"="$", **min**=0, **max**=100'000'000'000);

Les attributs prédéfinis d'une entité peuvent être interrogés très simplement par le simulateur en posant une question dite "*ponctuée*".

Capitalisation**.unité**

Le mot

"*ponctuée*"

signifie simplement qu'il faut mettre "*un point*" entre le nom de l'entité et l'attribut que le simulateur veut connaître, entre le nom de l'entité questionnée et celui de l'attribut recherché.

Héritage entre entités

Le simulateur permet d'étendre les attributs et les comportements des entités en les spécialisant, c'est-à-dire en ajoutant de l'information dans l'entité "*enfant*" par rapport à l'entité "*parent*".

L'entité au-dessus dans la hiérarchie des entités est en effet qualifiée de

"*parent*",

et l'entité en dessous de

"*enfant*".

On peut ainsi "*étendre*" les entités en les spécialisant tout en maintenant "*une hiérarchie*" entre elles.

Les notes

Parmi les entités prédéfinies dans le simulateur existe aussi le type

"*note*"

qui a les deux spécificités de:

- ne pouvoir contenir que des informations publiques pour toute les autre entités existant dans le simulateur;

- ne pas pouvoir contenir des "*variations*".

Je peux, par exemple, créer

"*une note publique*",

accessible par toutes les autres entités contenues dans le simulateur:

note CouleursDeBase
proportionnée Rouge;
proportionnée Jaune;
proportionnée Vert;
fin de CouleursDeBase;

Grâce à la l'héritage des entité parentes au entités enfant, je peux facilement créer une entité comme:

"*Risques en couleurs*"

entité RisquesEnCouleurs
étend CouleursDeBase;
équation Rouge + Jaune + Vert = 1
fin RisquesEnCouleurs;

L'équation que j'ajoute après avoir "*étendu*" l'entité CouleurDeBase "*déclare au simulateur*" que dans l'entité RisquesEnCouleurs

la somme des trois couleurs symbolisant les trois niveaux de risque, doit valoir 1,

la somme des trois parts d'investissement du plus au moins risqué doit valoir 1,

c'est-à-dire la totalité de l'investissement dans les trois risques doit valoir 1.

Je peux maintenant étendre l'entité générale "*Banque*" avec l'entité "*Risques en couleurs*" pour en faire une nouvelle entité plus particularisée

"*Banque avec risques en couleurs*":

entité BanqueAvecRisquesEnCouleurs
étend Banque;
étend RisquesEnCouleurs;
fin de BanqueAvecRisquesEnCouleurs;

C'est ce qu'on appelle "*l'héritage multiple*" des entités puisque la nouvelle entité BanqueAvecCouleurDesRisques hérite à la fois des entités Banque et RisquesEnCouleurs.

Les préfixes

On peut repasser ensemble les préfixes des déclarations pour bien voir à quoi ils servent dans le simulateur.

Les préfixes d'accessibilité

On a deux préfixes d'accessibilité à une entité déclarée

- publique

Toutes les autres entité présentes dans le simulateur ont accès à cette entité.

C'est l'accessibilité par défaut du simulateur.

- protégée

Cette accessibilité implique que

- les autres entités présentes dans le simulateur ne peuvent pas accéder à l'information que cette entité contient, sauf,

- les entités descendantes de cette entité, qui ont également accès à cette information.

Les préfixes de variabilité temporelle des idées

Si aucun préfixe de variabilité n'est utilisé, l'entité est considérée par défaut par le simulateur comme

Générale

Par défaut, le simulateur considère donc toutes les idées comme

"*proportionnées*"

et comme

"*continues*"

durant toute la durée d'une simulation.

La valeur d'une telle entité peut donc changer en tout instant de la simulation.

Les entités continues

- qui reçoivent explicitement une valeur fixe, ou,

- qui reçoivent une valeur instantanée déclenchée par un signal à un certain instant de la simulation,

deviennent

"*des entités discontinues durables*".

Constantes

Une entité constante ne change jamais de valeur une fois définie lors de la modélisation, donc l'utilisateur ne peut pas la faire varier d'une simulation à l'autre.

La variation temporelle d'une entité constante est donc nulle et l'opérateur de "*variabilité temporelle*" ne peut lui être appliqué.

Une entité constante nommée peut être définie dans un dossier, importée d'un dossier et peut être accédée depuis l'intérieur d'une sous-entité qui descend de l'entité où la constance est définie.

Paramètres

Une entité dite "*paramètre*" est définie durant la modélisation et reste constante durant toute la durée d'une simulation.

Mais l'utilisateur peut la modifier entre deux simulations pour voir l'effet de cette modification.

La variation temporelle d'une idée paramètre est donc nulle pendant la durée d'une simulation et on ne peut lui appliquer

"*l'opérateur de variation temporelle*"

lors de la modélisation.

Des valeurs peuvent être assignées à ces paramètres par

"*une initialisation par le modélisateur*",

au moment de la modélisation, ou par

"*une intervention de l'utilisateur*"

avant le début d'une simulation.

Une entité paramètre

- ne peut être définie dans un dossier,

- ne peut être importée depuis un dossier, et,

- ne peut être accédé implicitement depuis l'intérieur du modèle dans lequel il est défini.

Discontinue

Ces entités ne changent de valeur qu'à des instants particulier de la simulation et ont une variation temporelle nulle par définition dans le reste du temps de la simulation, puisqu'elles ne changent de valeur qu'à des instants précis.

On ne peut donc pas leur appliquer

"*l'opérateur de variation temporelle*"

durant la modélisation.

Les préfixes de causalité

Les équations dynamiques du simulateur, qui représentent la dynamique du marché, qui représentent des "*variations temporelle*" de certaines entités, ne prescrivent pas un sens de circulation de l'information entre les entités puisque ces équations sont acausales.

Certaines entités peuvent néanmoins avoir une causalité fixée volontairement par le modélisateur lors de la modélisation, pour faciliter le travail subséquent du simulateur.

C'est le cas des entités qui contiennent et exécutent des

"*fonctions*",

par exemple.

Les entités dites

"*bloc*",

en particulier, ont une causalité fixée.

Lors de l'appel à une entité qui contient "une fonction", comme "*un bloc*", par exemple, des informations sont fournies en entrée de l'entité en question et elle fournit des résultats en sortie.

Ces "*fonctions*" et ces "*blocs*" ont donc une causalité prescrite dès la modélisation.

Les préfixes "*entrée*" et "*sortie*" permettent de spécifier, et donc de restreindre, le sens de circulation de l'information dans certaines entités, ce qui peut simplifier le processus de solution du problème par le simulateur, puisque le sens de l'information est connu à priori.

A noter que le simulateur étant un simulateur à équations, il peut inverser le flot de l'information entre les entités, y compris dans les blocs, si cela lui permet de trouver malgré tout une solution.

Ainsi, toute idées ayant une connexion avec une idée à causalité fixe doit être déclarée avec un préfixe de causalité déterminé, soit "*entrée*" soit "*sortie*".

entrée

Spécifie que cette information est une information d'entrée dans l'entité.

sortie

Spécifie que cette information est une information de sortie de l'entité.

Les préfixes de modification

remplaçable

Ce préfixe autorise une modification en forme de "*remplacement*" de l'entité concernée par une autre entité.

redéclarer

Quand un modificateur contient une déclaration complète contenant préfixes et type, le préfixe "*redeclare*" est indispensable devant le modificateur qui va remplacer la déclaration originelle.

En général il faut préfixer la déclaration d'une entité qui peut être remplacée par "*remplaçable*" pour éviter tout problème au simulateur.

finale

Ce préfixe en face d'une déclaration d'entité permet d'éviter toute modification d'une entité déclarée par ailleurs.

Il est surtout utilisé pour éviter toute modification intempestive par le modélisateur et ainsi introduire des erreurs inconscientes, en sécurisant des entités importantes.

On peut revoir aussi les types d'entités?

GIANNI

On y vient.

Les types

GIANNI

La caractéristique fondamentale des représentations de tes entités est leur

"*type*".

En fait, les représentations du simulateur sont très "*typées*".

JEAN

Qu'est-ce que tu entends par représentations "*typées*"?

GIANNI

La partie

type

de la déclaration d'une entités décrit des caractéristiques fondamentales de la dite entité,

le type de

"*valeur*"

qui la caractérise, en particulier.

"*Typées*" revient à dire que je dois classer tes réflexions sur la réalité, tes idées, selon quelques types très précis, prédéfinis dans le simulateur.

Un "*type*" est une manière pratique de classer les entités selon un certain nombre de propriétés essentielles, communes à toutes les entités faisant partie de ce type.

Ces propriétés peuvent être vues comme une facilitation de communication entre les entités représentées dans le simulateur.

Elles peuvent être vues aussi comme une description d'interface de communication entre entités.

En première approche, je dois classer tes réflexions selon cinq types principaux de représentations prédéfinies.

Le type "proportionnée" ("Real")

Quand tu déclares qu'une certaine entité peut prendre n'importe quelle valeur entre deux infinis, l'infini négatif et l'infini positif, je dois la représenter par le type "***proportionnée***" dans le simulateur.

Ce type de représentation dit "***proportionnée***" de certaines de tes réflexions possède automatiquement les attributs prédéfinis suivants, ce qui est très pratique:

Attribut "*nom*" ("***name***")

un nom connu partout dans le modèle, rendant cette entité accessible partout et n'importe quand par n'importe quelle autre entité présente dans le simulateur.

Par exemple:

"***Hermes***",

ou encore

"**GE**".

Attribut "*valeur*" ("***value***")

Cette attribut "*valeur*" est une valeur numérique de type "***proportionnée***".

Cette valeur est accessible directement et simplement par le "***nom***" de l'entité concernée, sans aucune précision supplémentaire, ce qui est très pratique.

Cette valeur est un ***nombre*** dit "**proportionné**" ("***Real***"),

"***10.25***",

ou encore

"***13.15***",

ou

"***1'234.67356534***"

par opposition à un nombre dit "***échelonné***" ("***Integer***"), comme

"***1***", "***2***", "**9**", ou

"***10***", ou encore

"***13031***"

par exemple,

un type d'entité "***échelonnée***", différent du type d'entité "***proportionnée",*** donc.

Attribut "*substance*" ***("quantity")***

Un nom décrivant de quelle substance l'entité que tu déclares exister est constituée, comme

"***liquide***",

ou

"***immobilisations***"

par exemple.

Attribut "*unité*" ***("unit")***

Un nom décrivant les unités de substance que le simulateur doit utiliser quand il utilise la valeur de cette idée,

"***Dollar***",

par exemple.

Attribut "*valeur minimale*" ***("min")***

La valeur minimale représentable dans le simulateur.

Attribut "*valeur maximale*" ***("max")***

La valeur maximale représentable dans le simulateur.

Attribut "*valeur initiale*" ***("start")***

La valeur initiale que l'entité doit avoir au début d'une simulation, à un instant qui est fixé à zéro par défaut dans le simulateur.

Attribut "*valeur fixée*" ***("fixed")***

Spécifie si la valeur initiale d'une entité de type continu ***proportionnée*** est

- fixée définitivement par le modélisateur lui-même ou bien

- peut être ajustée par le simulateur lui-même avant de commencer une simulation.

La valeur par défaut de cet attribut "***valeur fixée***" est

- "***Vrai***" ("***True***")

pour les entités "*constantes*" et les entités "*paramètres*", et,

- "***Faux***" ("***False***")

pour les autres entités générales dont je déclare l'existence,

ce qui signifie que si cet attribut est mis à "*Faux*", à un instant ou à un autre de la simulation, l'entité peut malgré tout varier dans le temps au cours de la simulation,

ce qui signifie aussi que tant que ***valeur fixée*** n'est pas mise à "*Vrai*" (depuis "*Faux*") par le modélisateur,

la valeur initiale de l'entité est traitée par le simulateur comme

"*une supposition initiale*"

que le modélisateur fait sur la capacité de variation de la dite entité,

et qui peut être changée par le simulateur pour trouver une valeur initiale raisonnable avant de commencer une simulation.

Attribut "*valeur nominale*" ***("nominal")***

Une valeur nominale typique qui peut être fournie par le modélisateur et utilisée par le simulateur pour se faire sa propre idée des valeurs possibles, ainsi que des tolérances possibles pour la valeur de cette entité,

en trouvant lui-même un ordre de grandeur pour des valeurs typiques de cette entité.

Le modélisateur n'est pas obligé de fixer cette valeur nominale: même si une valeur par défaut n'est pas spécifiée par le modélisateur, le simulateur sait trouver et propager lui-même automatiquement une valeur nominale.

Attribut "*réalité obligée*" *(***"stateSelect"***)*

Un attribut, utilisé pour un contrôle manuel par le modélisateur, lui permet de spécifier que cette entité doit impérativement figurer dans les simulations, c'est-à-dire figurer impérativement dans les équations que le simulateur va construire lui-même pour faire une simulation.

Cet attribut est en fait une énumération et sa valeur par défaut est

StateSelect.default,

qui autorise le simulateur à faire une sélection automatique de cette entité s'il estime qu'elle doit figurer dans la simulation.

Cette valeur par défaut fonctionne normalement bien dans la plupart des applications et le modélisateur n'a normalement pas besoin de fixer cet attribut dans le modèle.

A part "***default***", StateSelect permet au modélisateur de donner quatre instructions au simulateur pour le choix des entités à représenter:

"***jamais***" ("never")

Ne jamais utiliser cette entité dans les simulations.

"***toujours***" ("always")

Toujours utiliser cette entité dans les simulations.

"***préférer***" ("prefer")

Préférer cette entité sur celles ayant la valeur "*default*" par défaut, ce qui permet aussi au simulateur de considérer des entités auxquelles l'opérateur de variation temporelle n'est jamais appliqué dans le modèle

"***éviter***" ("avoid")

Utiliser cette entité si elle ne peut pas être évitée, mais seulement si elle apparait dans une opération de variation temporelle et qu'aucune autres des entités potentielles ayant les attributs "*default*", "*prefer*" ou "*always*" ne peuvent être sélectionnées.

Le type ***"Echelonné"*** *(***"Integer"***)*

Quand tu penses qu'une entité varie par paliers, selon des échelons, comme les classifications des fonds souverains, par exemple, je dois la représenter comme un nombre échelonné, comme un nombre entier.

Un cas très classique de telles entités est celle de

"compteur".

Le type "Douteux" ("Boolean")

Quand tu penses qu'une certaine entité ne peut être que "*vraie*" ou "*fausse*", c'est qu'elle est "*binaire*", qu'elle ne peut prendre que l'une de ces deux valeurs de "*vrai*" ou "*faux*":

Les type "Textuel" ("String")

Quand une de tes réflexions est une information textuelle, un commentaire sur une entité, par exemple, je dois la représenter comme une chaine de caractères, comme un texte en quelque sorte.

Par exemple:

"*Hermes va bien*".

A noter que les entités représentées par des types

Echelonnée, Douteux et Texte

ont une variabilité durable, c'est-à-dire que

- lorsqu'elles prennent une valeur, elles la gardent au cours du temps de la simulation jusqu'à leur prochain changement,

- contrairement aux idées instantanées, dont tu as vu qu'elles n'ont une valeur qu'à un instant précis, en particulier celui de l'apparition d'un fait ou d'un battement de cadence, c'est-à-dire ni avant, ni après cet instant précis.

Le type "Enumérable" ("enumeration")

Conceptuellement, une entité énumérée ressemble énormément à une entité échelonnée.

Elle a beaucoup de caractéristiques communes avec ces dernières mais sa valeur, au lieu d'être un nombre entier, échelonné, peut être explicitement nommée.

En outre, une entité énumérée possède en général plusieurs valeurs de type verbalisable en langue naturelle, constituant une liste.

Par exemple, une entité énumérée de trois valeurs permet de représenter une entité comme la

"*Taille*",

par exemple,

qui peut être la caractéristique d'une autre entité comme une

"*Entreprise*",

par exemple,

ce qui donne la déclaration suivante au simulateur

Taille = **enumeration** (petite, moyenne, grande)

Le type "Cadenceur" ("Clock")

Le type "*Cadenceur*" ("*Clock*") permet de créer des cadences particulières, des cadences ayant certains attributs, en particulier celui de pouvoir émettre des battements (beat) déclenchant des comportements.

Tant des cadenceurs "*périodiques*", qui émettent un battement régulier, que des cadenceurs "*apériodiques*", qui émettent un battement à certains instants particuliers irréguliers, sont possibles.

Les entités et les équations qui sont définies au battement d'une cadence particulière appartiennent à

"*une partie cadencée*"

du modèle.

Tu as vu qu'il y a trois types d'entités permettant de représenter tes réflexions de type discontinues dans le modèle:

- un type instantané, et,

- deux types durables, que l'on peut caractériser par leur variabilité.

Les entités discontinues ont une variabilité discontinue:

- elles ne peuvent changer de valeur que lorsqu'un critère devient vrai lors d'une simulation,

ou,

- à l'instant du battement d'une cadence, régulière ou irrégulière.

Elle ont donc une variation temporelle nulle en dehors de l'instant précis de apparition de ce seuil ou de ce battement.

Il est donc impossible d'appliquer l'opérateur variabilité temporelle à une réalité discontinue puisqu'elle ne varie pas dans le temps, hormis à un instant précis.

Les expressions discontinues permettent de représenter:

- des entités "*constantes discontinues*" ou des entités "*paramètres discontinus*" ainsi que,

- des entités exécutant des fonctions exigeant des entrées discontinues.

Si l'entité n'est pas cadencée par un cadenceur, elle est seulement probable.

Les entités probables

Le simulateur possède quatre possibilités de représentation des modèles hybrides continus-discontinus:

Les équations conditionnelles (si-équations)

Utilisées pour représenter des équations qui ne sont valides qu'à des instant précis, c'est-à-dire quand certaines conditions deviennent vraies, quand certains critères deviennent vrais.

Ces "*si-équations*" sont très utiles pour représenter des comportements ayant différentes expressions dans différentes régions de comportement.

valeur de l'entité **= si** valeur de l'entité **>** critère **alors** critère **sinon** valeur de l'entité;

Les équations instantanées (quand-équations)

Les équations instantanées existent dans le simulateur pour exprimer des entités instantanées, des équations qui ne deviennent valides qu'à un certain instant, pour représenter des entités qui arrivent à des discontinuités quand des conditions spécifiques que nous appelons "*critères*" deviennent vraies.

Les équations apparaissant dans une quand-équation sont activées quand au moins l'un des critères, l'une des conditions, devient vrai et n'est vrai seulement à un instant, de durée nulle dans le temps du simulateur, par définition.

quand critère
alors équations;

fin du quand;

Plusieurs critères sont aussi possibles dans une quand-équation.

quand {critère1, critère2, ...}
alors équations;
fin du quand;

Une

"*Action rebondissante*"

est un bon exemple d'idée hybride pour laquelle une quand-équation est appropriée dans un modèle.

Dans le modèle, le comportement de l'action en bourse, son

"*Cours*"

peut être considéré comme variant au-dessus d'un certain

"*Cours plancher*".

L'idée de base est que quand l'action touche son cours plancher, elle

"*rebondit*".

Un changement discontinu se produit donc à l'instant du rebond sur le cours plancher.

L'idée du rebond sur le cours plancher peut être représentée comme une inversion de la vitesse de variation de l'action.

Une action idéale aurait un coefficient d'élasticité de 1 ce qui signifie qu'elle ne perdrait aucune énergie à l'instant du rebond.

Une action plus réaliste aurait une élasticité de 0.9, lui permettant de garder 90% de sa vitesse à l'instant du rebond après le rebond mais dans le sens contraire.

Le modèle de

"*une action rebondissante*"

doit contenir deux équations de comportement liant "*son cours*" à "*sa vitesse de variation*".

A l'instant du rebond, la vitesse est soudainement inversée et légèrement diminuée selon l'élasticité de l'action:

vitesse après le rebond = -Elasticité de l'action * vitesse de variation avant le rebond.

Pour représenter ce comportement, le simulateur dispose d'une forme prédéfinie d'équation instantanée dite "**réinitialise**", qui permet d'effectuer une réinitialisation de la vitesse au moment de l'impact avec le cours plancher:

rénitialise(Vitesse, -Elasticité * **pre**(Vitesse));

Le modèle d'une action rebondissante est donc le suivant:

entité
ActionRebondissante
constante proportionnée Tendance = 9.81;
paramètre Elasticité = 0.9;
paramètre EcartParRapportAuCoursPlancher = 0.1;
proportionnée Cours(start = 1);
proportionnée Vitesse(start = 0);
équations
der(Cours) = Vitesse;
der(Vitesse) = - Tendence;
quand
Cours <= EcartParRapportAuCoursPlancher
alors
reinitialise(Vitesse, -E * **pre**(Vitesse));
fin du quand;
fin de
ActionRebondissante;

A noter que les équations dans la quand-équation ne sont actives qu'à l'instant où la condition devient vraie, ou le critère "*écart par rapport au cours plancher*" devient vrai, alors que les conditions dans une si-équation restent actives tant que la condition de la si-équation est vraie, tant que le critère est vrai.

Les cadenceurs synchrones (Clock)

Les machines à état

Comportements continus dans des partitions temporelles

Le but de pouvoir représenter des comportements continus dans des partitions temporelles est de permettre des systèmes de gouvernance discontinus sur des entités continues.

Ce problème est résolu en intégrant dans le temps les équations continues avec une méthode d'intégration définie entre des signaux qui peuvent être des critères ou des battements de cadence.

Avec cette méthode, il est possible d'inverser le modèle d'une entité et de l'utiliser dans une anticipation, une prévision, associée à une cadence générale, dans la gouvernance de la dite réalité par un gouverneur.

Les caractéristiques des équations continues dans des partitions temporelle cadencées permet également la définition de systèmes multi-cadencés.

Cela signifie que différentes parties de la réalité continue sont associées à différentes cadences et utilisent des méthodes d'intégration différentes entre les battements si cela est nécessaire.

Du point de vue d'une partition continue, des battements cadencés ne sont pas interprétés comme des battements mais comme des "*tailles de pas temporels*" que l'intégrateur temporel du simulateur doit absolument satisfaire.

C'est exactement la même hypothèse que les intégrateurs cadencés manuellement, comme la transformation en z par exemple.

Du point de vue d'une partition discontinue, la partition continue est découpée en morceaux et les entités discontinues n'ont de valeur qu'à l'instant d'un signal.

Le simulateur gère donc une telle partition exactement comme toute partition discontinue.

Les opérateurs

"*prend*", "*maintien*", et "*reprend*"

sont utilisés pour rendre opérationnels les battements de la partition continue instantanée dans les autres partitions temporelles continues.

Une partition continue découpée par une cadence particulière est donc considérée comme une partition cadencée.

Il y très peu de concepts à comprendre pour les partitions continues.

Il y en a en revanche beaucoup pour les partitions discontinues.

Le simulateur doit en effet propager explicitement les conditions de cadencement et cette nécessité exige que les équations soient groupées dans des partitions de même cadence où seules des équations ayant la même cadence peuvent être résolues, ce qui impose certaines contraintes au simulateur.

A cause du principe

"*prélever et maintenir*" ("*prend et maintien*"),

toutes les entités dont on affirme l'existence ou faisant partie d'une affirmation temporisée doivent avoir une valeur initiale, car elles peuvent être utilisées avant qu'une valeur ne leur soit assignées pour la première fois.

Les équations

JEAN

On peut reparler un peu des équations?

GIANNI

Il n'y a rien de bien compliqué à comprendre quand on parle d'équations.

"*une équation*",

souvent appelée aussi

"*une égalité*".

n'est rien d'autre qu'une représentation graphique des deux mains, de la main gauche et de la main droite, séparées par un signe égal, que l'on peut concevoir comme une représentation de son nez ou plutôt de sa pensée.

A noter que la main gauche ne doit pas forcément contenir une entité variable dans nos équations.

Les équations initiales

Ce type d'équation sert uniquement à définir des conditions initiales pour une simulation.

Elles sont donc utilisées par le modélisateur ou l'utilisateur uniquement pour spécifier des conditions initiales dans lesquelles se trouvent les entités.

Les équations conditionnelles

Les équations conditionnelles sont un peu plus compliquées que les équations initiales, mais pas beaucoup.

Les équations conditionnelles servent à représenter des réflexions où ce qui est à gauche du signe égal est égal à ce qui est à droite seulement si une condition est vraie ou devient vraie.

JEAN

Mes réflexions commencent souvent par un

"si"

GIANNI

S'il y a un "**si**", c'est qu'il est suivi par

"*une condition*"

qui, si elle est remplie, si elle est vraie, ou si elle le devient au cours d'une simulation, entraine elles-même

"*des comportement*".

JEAN

Exactement.

GIANNI

Est-ce que je peux représenter ça comme la construction suivante:

si
condition vraie ou devient vraie;
alors
comportements
fin du si

JEAN

Oui.

GIANNI

Si je veux considérer plusieurs conditions et non une seule, est-ce que je peux exprimer ça sous la forme suivante:

si
condition1 vraie ou devient vraie;
alors
comportement1;
ou si
condition2 vraie ou devient vraie;
alors
comportement2;
ou si

condition3 vraie ou devient vraie;
alors
comportement3;
sinon
comportement4;
fin du si

où les parties

ou si

sont optionnelles.

JEAN

Oui.

GIANNI

Est-ce que je peux aussi écrire une équation du type suivant:

Réflexion = **si** condition vraie ou devient vraie **alors** comportement1 **sinon** comportement2

JEAN

Oui.

GIANNI

Bon. Alors on peut appeler ce genre de réflexions des

"*équations conditionnelles*",

ou même des

"*si-équations*"

puisque le formalisme de base utilisé pour représenter des comportements est

"*des équations*".

Les si-équations ne sont que des équations particulières.

En résumé, les équations conditionnelles deviennent actives dans deux cas:

- quand la condition de la si-équation change de la valeur *fausse* à la valeur *vraie*, ou,

- quand un cadenceur contrôlant l'équation donne un coup de cadence.

En outre, ces équations conditionnelles peuvent contenir tant des expressions continues que des expressions discontinues.

Enfin, les équations conditionnelles ne représentent que des idées discontinues puisqu'elles ne deviennent actives qu'à l'instant précis d'un signal indiquant soit qu'une condition passe de *faux* à *vrai*, devient vraie, soit un coup de cadence.

JEAN

Si tu veux.

A part des réflexions commençant par un

"***si***"

je fais souvent des réflexions qui commencent par un

"***quand***"

pour comprendre ma réalité:

- "*quand*" un critère devient vrai, ou,

- "*quand*" un certain signal arrive,

se déclenchent instantanément certains comportements.

GIANNI

Dans ce cas je peux qualifier ces critères ou ces signaux de

"*imprévisibles*"

puisqu'ils peuvent se produire ou arriver n'importe quand.

JEAN

Si tu veux.

GIANNI

On peut donc aussi parler de

"*quand-affirmations*",

et de

"*quand-équations*"

JEAN

Si tu veux.

GIANNI

Tu utilises bien ces quand-réflexions pour comprendre des entités instantanées, qui ne sont actives qu'à un instant précis:

quand
arrivée d'un signal
alors
comportement
fin du quand

JEAN

Exactement.

Je fais souvent également aussi des réflexions sur des cloches qui sonnent à des intervalles réguliers.

GIANNI

Qu'est-ce que tu entends par sonnerie de cloche?

JEAN

Par exemple, on sonne une cloche pour signaler la fin d'une séance à la bourse.

Ou encore, certaines entreprises publient des comptes tous les trois mois.

Ou encore, certaines entreprises publient des comptes tous les six ou douze mois.

GIANNI

Si tu acceptes, je peux appeler de telles entités des

"entités cadencées"

Ce sont des entités dont les comportements sont déclenchés par un signal particulier, par

"*un coup de cadence*"

sachant que les cadences peuvent être irrégulières, comme

"*de temps-en-temps*"

ou régulières comme

"*tous les trois mois*"

par exemple.

JEAN

Oui.

GIANNI

Alors, je peux représenter ces réflexions sous la forme suivante:

quand
coup de cadence
alors
comportement
fin du quand

Si plusieurs cadences existent simultanément, le simulateur doit pouvoir déterminer quelles équations sont activées par quelle cadence en faisant des raisonnements sur les différentes cadences existantes dans le modèle.

JEAN

Logique.

GIANNI

Pour résumer, on peut dire que

les si-équations

sont plus fondamentales que

les quand-équations

puisque les quand-équations imprévisibles peuvent être exprimées en terme de si-équations.

JEAN

Comment ça?

GIANNI

Les quand-équations imprévisibles sont des équations conditionnelles ayant la forme suivante:

quand
critère passe de faux à vrai
alors
équations

Elle peuvent en outre avoir des parties optionnelles:

quand
critère1
alors
équations1
ou quand
critère2
alors
équations2
fin du quand

JEAN

D'accord.

GIANNI

Je me demande quand tu utilises plus particulièrement des quand-réflexions plutôt que les si-réflexions.

En effet, les quand-équations peuvent être utilisées dans

"*des sections algorithmiques traditionnelles*"

mais pas dans

"*des fonctions*"

qui par définition reçoivent une information en entrée et en fournissent une en sortie, sans se préoccuper du temps de la simulation, car le temps de la simulation ne s'écoule pas pendant le calcul de la fonction par le simulateur.

Les critères

GIANNI

Quand tu fais des réflexions sur des entités discontinues, comme en pensant qu'un certain comportement est déclenché si un critère devient vrai, je dois représenter ces entités comme existant à certains instants précis

Ceci par opposition aux représentations continues qui sont en général conçues selon des principes de variations temporelles et de conservations de certaines entités.

Un bon modèle doit donc être hybride, c'est-à-dire pouvoir représenter à la fois des idées discontinues et des idées continues.

En outre, un bon modèle doit être interactif en ce sens qu'il doit représenter tant

- les contraintes temporelles fixées par l'environnement sur l'entité sur laquelle tu es entrain de réfléchir, comme

. l'apparition soudaine d'un signal externe, ou

. l'existence d'une cadence provenant d'un signal cadencé externe.

- les contraintes temporelles de l'entité étudiée qui interagit avec cet environnement, comme la limitation de ses capacités de calcul, par exemple, impliquant des délais de réaction.

Enfin les deux modèles, continu et discontinu, doivent être synchronisés.

Les faits de la réalité se suivent, se succèdent, s'enchaînent: ils sont ordonnés dans le temps de la simulation.

Néanmoins cet ordre est partiel car deux faits peuvent apparaître au même instant, ce qui implique que pour obtenir un ordre il faille:

- utiliser des relations causales entre les faits,

- prioriser certains faits ou même, si cela n'est pas suffisant,

- déterminer un ordre fondé sur d'autres propriétés des dits faits.

Pour résumer, on doit représenter une entité instantanée dans un modèle hybride par une idée qui:

- n'a pas de durée, a une durée nulle dans le simulateur, et,

- peut dépendre d'une condition binaire, qui peut changer de valeur de *faux* à *vrai*, ou inversement de *vrai* à *faux*, pour qu'un comportement se produise, pour qu'un comportement soit *activé* ou *désactivé*.

Les comportements instantanés sont donc un cas spécial de comportements, qui ne sont actifs que

- si certaines conditions sont remplies, représentés par des si-équations;

- si certains instant sont déterminés dans le temps, représentés par des quand-conditions.

Les entités instantanées

Une entité instantanée est définie par sa ponctualité temporelle, définie ni avant ni après le dit instant.

En outre, les entités instantanées sont à la base de la synchronisation.

Dans le temps de la simulation, les entités instantanées ne prennent pas de temps pour intervenir: elles n'ont pas de durée d'existence dans le temps de la simulation.

Les entités continues

Une entité continue est définie par sa non segmentation par des instants, par une existence sans distinction d'instants entre "*moins l'infini*" et "*plus l'infini*".

Les entités cadencées régulières

Une entité cadencée régulière est définie par des durées constantes entre instants qui se suivent.

Les entités cadencées emboitées, multi-durées, sont possibles.

Les entités incertaines

Une entité incertaine est définie par des durées variables entre les instants qui caractérisent son passage de fausse à vraie.

Les entités conditionnées

Une entité conditionnée l'est par une condition.

On peut faire ici un petit rappel sur les quatre classes de variabilité des entités.

Les entités continues

La variabilité continue est la variabilité par défaut du simulateur.

Sans autre précision de la part du modélisateur, les entités sont considérées comme continues dans un modèle de l'entité à laquelle tu penses.

Les entités continues ont une variabilité continue: elles peuvent changer de valeur continuellement durant une simulation.

La partie continue du modèle peut contenir des entités constantes, des entités paramètre, des entités instantanées durables ainsi que des appels à des algorithmes ou à des fonctions qui ne sont rien d'autre que des entités un peu spéciales.

Une idée contenant des entités continues est toujours continue, même combinée avec des entités constantes ou paramètres.

Par exemple si:

"*Elasticité*"

est une entité continue, alors, combinée avec une entité constante, comme le nombre 2, par exemple,

2*Elasticité

elle donnera une nouvelle entité qui est toujours une entité continue.

Le fait qu'une entité ait une variabilité continue ne signifie pas qu'elle va nécessairement varier.

Cela signifie seulement qu'elle a la possibilité de varier au cours d'une simulation.

Par exemple, j'affirme que

Elasticité= 2

dans un modèle,

alors Elasticité aura une valeur constante mais conservera sa propriété de variabilité continue.

Les entités constantes (constant)

L'usage d'entités constantes dans le temps permet de s'assurer que l'utilisateur du simulateur ne peut pas les modifier au cours de son travail de simulation.

Ce sont des entités considérées comme générales, donc non modifiables par qui que ce soit, sauf au moment de la modélisation, évidemment.

Les entités paramètres (parameter)

Une entité paramètre est un paramètre du modèle, définie par le modélisateur, ne variant pas dans le temps d'une simulation.

Elle est:

- une entité définie à priori, avant toute simulation, résultant généralement d'une intuition de l'utilisateur ou d'un besoin du modélisateur.

- une entité que le modélisateur doit définir ou modifier à l'avance, en plus des idées constantes,

- une entité considérée comme constante durant une simulation seulement, donc modifiable entre deux simulations par l'utilisateur,

- une entité considérée comme une entrée dans le modèle, en plus des entités connues considérées comme constantes.

Une entité "*supposée*", autrement dit "*un paramètre*", ne peut être changée que d'une modélisation à l'autre, et non d'une simulation à l'autre, alors que les entités "*connues*", autrement dit "*les constante*" ne le peuvent jamais, même au cours de l'amélioration du modèle.

Des valeurs variables peuvent être assignées à de telles entités que sont les paramètres durant une phase d'initialisation du modèle par le modélisateur ou interactivement par l'utilisateur du simulateur immédiatement avant une simulation.

Le calcul d'une variation temporelle ne peut pas s'appliquer à de telles entités puisque la valeur d'une telle variation temporelle durant une simulation est toujours nulle pour elles.

Les entités supposées ne peuvent être définies que dans un modèle.

Elles ne peuvent être ni importées d'autres modèles ni être accédées implicitement par un balayage dans le modèle où elles sont définies.

Il est donc, répétons-le, possible à l'utilisateur d'assigner une valeur particulière à une entité paramètre juste avant qu'une simulation ne démarre.

Les entités discontinues (discrete)

Les entités discontinues ne changent de valeur qu'à des instants précis de la simulation.

Elles ont également une variabilité temporelle nulle par défaut dans la modélisation. Sans autre précision elles sont considérées comme durables entre deux instants particuliers.

Le simulateur fait la distinction entre deux types d'entités discontinues.

Les entités instantanées

Elles sont des entités qui ne sont définies et ne changent de valeur qu'à certains instants précis, en particulier à réception de signaux d'une cadence associée, qui peut être irrégulière ou régulière.

Les entités durables

Ces entités maintiennent leur valeur dans la durée séparant deux instants d'arrivée de signaux susceptibles de les faire changer de valeur.

Les entités binaires sont quant à elles discontinues durables par défaut dans la modélisation.

Elles ne peuvent pas être continues par définition.

Si l'utilisateur fait des réflexions sur des entités en pensant qu'elles sont discontinues, elles seront représentées par des équations discontinues, qui auront ont une variabilité discontinue et durables par défaut, sans autres précisions de sa part.

C'est la conséquence naturelle du fait que les équations et les expressions instantanées ne sont exécutées par le simulateur qu'à des instants précis, causant le changement de telles expressions contenues dans des constructions instantanées changeant de valeur seulement à ces instants précis.

Si elle apparaissent:

- comme entrée de la fonction

noEvent()

prédéfinie dans le simulateur, ou

- dans des expressions de comparaison comme:

valeur entité1 < valeur entité2

ou

valeur entité1 > valeur entité2

ou

valeur entité1 <= valeur entité2

ou

valeur entité1 >= valeur entité2

ou

- dans deux fonctions prédéfinies dans le simulateur:

ceil

et

floor

ou encore

- comme argument des opérateurs

div*, *mod et ***rem***,

également prédéfinies dans le simulateur,

on obtient des expressions à variabilité discontinues, même si leurs entrées ont une variabilité continue.

Cela provient du fait que les résultats de ces "*opérateurs relationnels*" et de ces "*fonctions prédéfinies*" sont instantanés durables et causent la génération d'une entité nouvelle uniquement quand ils interviennent dans la simulation

C'est-à-dire aussi que les valeurs des entités sur lesquelles ils interviennent en sortie ne changent qu'à certains instants précis du temps du simulateur.

Néanmoins, une réflexion relationnelle et une fonction insérée dans un *noEvent()* sont des affirmations continues plutôt que discontinues.

Par exemple,

valeur de l'entité < 2

est une réflexion discontinue alors que

noEvent(valeur de l'entité < 2)

est une réflexion continue puisqu'elle ne produit rien en sortie.

Les réflexions concrétisées par des fonctions dans le simulateur se comportent comme si elles étaient discontinues puisqu'une fonction est toujours appelée et évaluée à un certain instant précis de la simulation et que le simulateur attend le résultat de l'évaluation avant de continuer.
C'est-à-dire encore que le temps de la simulation ne s'écoule pas durant l'évaluation d'un appel de fonction, tout comme il ne s'écoule pas lors d'une équation instantanée.

Pour reprendre l'exemple précédent,

valeur de l'entité < 2

est une réflexion discontinue, puisqu'elle génère un signal alors que

noEvent(valeur de l'entité < 2)

est une réflexion continue, puisqu'elle ne génère pas de signal.

En outre les appels de fonctions peuvent se produire à n'importe quel instant d'une simulation.

Une entité continue que je représenterais sans préfixe

"*discontinu*"

et qui serait incluse dans une équation instantanée devient discontinue par défaut dans le simulateur.

Pour résumer, les entités durables peuvent être:

- des idées paramètres.

Ceci peut sembler une contradiction de dire que des entités paramètres peuvent appartenir à la classe des idées discontinues même si elles ne changent pas durant la simulation.

Mais, si une entité discontinue ne "doit" pas forcément changer, elle "peut" changer à certains instants précis comme:

- des entités continues déclarées par des équations temporelles contenant un

"*quand*"

- des appels de fonctions où toutes les informations d'entrée de la fonction sont des entités durables.

- des réflexions dont toutes les sous-réflexions sont des réflexions durables.

La genèse des signaux

JEAN

Qu'est-ce que tu entends par genèse des signaux?

GIANNI

Nous avons vu que les signaux peuvent être classés en deux grands groupes selon qu'ils sont générés par l'entité sur laquelle tu te concentre ou par son environnement.

Les signaux peuvent aussi être classés selon la manière dont ils sont générés dans l'événement représenté dans le simulateur.

Les signaux prévisionnels

Les signaux prévisionnels sont directement liés au temps du simulateur de telle sorte qu'il est possible pour le simulateur de prévoir l'instant d'apparition d'un signal avant qu'il n'apparaisse.

De tels signaux sont facilement gérés par le simulateur puisqu'ils peuvent être prévus à l'avance.

Un signal temporel ne peut résulter que d'une réflexion temporelle discontinue impliquant le temps du simulateur.

Il peut apparaître:

- dans une équation temporelle cadencée irrégulière;

- dans une équation temporelle cadencée régulière;

- des réflexions de la forme:

temps du simulateur >= entité temporelle discontinue;

temps du simulateur > entité temporelle discontinue.

Toute l'entité doit avoir une variabilité discontinue durable, c'est-à-dire qu'elle ne doit contenir que des entités constantes, des entités paramètres ou des entités discontinues durables.

Par exemple, la condition

temps du simulateur >= 12.0

dans l'équation temporelle

quand
temps du simulateur >= 12.0
alors
comportement
fin du quand

va générer un signal déclenchant un comportement à

temps du simulateur = 12.0

puisque l'équation devient *vraie* autour de l'instant 12.0.

Le mot "autour" que j'ai utilisé provient du fait que les valeurs des entités de type proportionnelles, telles que 12.0, par exemple, ne sont pas forcément des valeurs dites entières telles que 12, par exemple, puisqu'elle ont une virgule et quelque-chose après la virgule, ce qui fait que le simulateur déclenche le signal "autour de la valeur 12 exacte" et non pas à la "à la valeur 12 exacte".

Une autre manière de générer un signal est de passer une affirmation temporelle à l'une des fonctions internes du simulateur ayant par défaut une discontinuité, comme:

abs, sign, div, mod, rem, ceil, floor et integer.

En effet, ces fonctions ont une variabilité discontinue.

De tels signaux sont déclenchés au point de discontinuité, quand une entrée de la fonction en question contient une sous-affirmation continue et quand l'appel de la fonction est fait hors d'une équation ou d'une affirmation temporelle.

Quand l'appel à la fonction est à l'intérieur de la construction temporelle un nouveau signal n'est pas produit puisqu'un signal est déjà en traitement.

Rappelons que les fonctions prédéfinies du simulateur ne peuvent pas générer des signaux même si elles font appel à des fonctions signalétiques, car le corps d'une fonction du simulateur est traité comme si noEvent() s'appliquait à elle par défaut.

Les entités cadencés

Les idées de signaux cadencés sont directement liées aux battements d'une cadence produite par un cadenceur.

Ils sont donc similaires aux signaux prévisionnels dans le sens qu'il est possible de prévoir le prochain battement avant qu'il ne se produise, au moins pour temps cadences régulières.

Cela permet au simulateur de gérer les signaux instantanés cadencés réguliers puisqu'ils peuvent être anticipés.

Par exemple, une réflexion qu'on peut mettre sous forme d'équation instantanée, qui s'active à un instant précis du simulateur:

quand
instant précis
alors
comportement
fin du quand

Les signaux comportementaux

Les signaux comportementaux ne peuvent être prévus et sont liés aux changements de valeurs des entités présentes dans le modèle au sens large, ce qui inclut les entrées et les sorties des blocs et des fonctions, qui ne sont que des entités un peu particulières, ayant une entrée et une sortie bien définies au moment de la simulation, donc une causalité prédéfinie lors de la modélisation, et donc de la simulation.

Un signal comportemental peut être induit par une équation conditionnelle impliquant au moins une entité comme dans l'équation temporelle ci-dessous, où un signal est généré et géré chaque fois que la fonction d'une entité devient vraie:

quand
fonction(entité) > 0.5
alors
comportement
fin du quand

Un signal comportemental peut également être produit par le simulateur quand il connaît une affirmation contenant une référence à une entité contenue dans l'une des fonctions discontinues du simulateur comme mod(entité-5,2), par exemple.

Le simulateur surveille en permanence les affirmations produisant des signaux durant la solution numérique de la partie continue du modèle.

Les signaux prédéfinis

Le simulateur dispose d'un certain nombre de fonctions prédéfinies liées aux signaux, que l'on peut utiliser lors de la modélisation et donc pendant la simulation.

Certaines fonctions génèrent des signaux instantanés, d'autres génèrent des durées, d'autres convertissent des entités instantanées en entités durables, d'autres peuvent être utilisées pour exprimer des conditions de genèse de signaux et d'autres encore permettent l'accès à des entités avant que le signal ne se produise.

initial()

La fonction initial() retrourne *vrai* à un instant juste avant le début d'une simulation, quand temps de la simulation est toujours égal à

temps.start,

généralement fixé à zéro.

En utilisant initial() quelque part dans le modèle, le modélisateur peut générer un signal fictif immédiatement avant le début de la simulation.

L'utilisation typique de initial() est dans des conditions faisant partie d'équations temporelles durables.

Celà peut être utile si le modélisateur a besoin d'initialisations additionnelles d'entités discontinues en plus des valeurs données par l'attribut *start* de ces entités.

A noter que fixer une valeur initiale en la qualifiant de *fixée*, par une équation initiale pour cette entité, provoque un système d'équations surdéterminé ce qui peut provoquer des problèmes dans le simulateur:

quand
initial()
alors
quelques initialisations ou autres actions
fin du quand

En fait, si initial() fait partie des conditions d'une équation instantanée ou d'une expression instantanée, les équations figurant dans ces affirmations deviennent partie du système d'équations d'initialisation traité par le simulateur avant le début de la simulation.

Une autre manière d'affirmer un comportement à l'instant initial est d'utiliser l'expression:

initial equation.

terminal()

L'appel à terminal() retourne *vrai* à l'instant précis d'une simulation réussie.

S'il est utilisé quelque-part dans le modèle, il peut générer un signal.

Cette fonction est typiquement utilisée pour déclencher des actions à la fin de la simulation, par exemple en appelant une fonction externe pour enregistrer certaines données dans un fichier qui laissera une trace de la simulation pour l'utilisateur.

Si la simulation échoue, terminal() ne retournera jamais *vrai* durant la simulation en question.

terminate(message)

Cause une fin de la simulation et permet au simulateur de passer un message à l'utilisateur.

La raison peut être qu'il n'est pas nécessaire de poursuivre la simulation à partir de l'instant où une condition particulière est remplie, comme

"*toucher son cours plancher*"

pour une action, par exemple.

Si le modélisateur veut tester une condition qui, lorsqu'elle est remplie, cause la fin de la simulation, il doit utiliser assert() au lieu de terminate().

sample()

Cette fonction que l'on peut comprendre comme "***saisir()*** " permet de générer des signaux périodiques.

La fonction continue

sample(premier, intervalle)

où

"*premier*" est un nombre réel précisant le premier signal, et,

"*intervalle*" est un nombre réel spécifiant la durée entre signaux périodiques,

retourne "*vrai*" et peut être utilisée pour déclencher des signaux aux instants

temps + nombre * intervalle, nombre = 1, et ainsi de suite

Cette fonction est typiquement utilisée pour des entités cadencées.

Il existe une version discontinue de sample() qui est même meilleure pour représenter les entités cadencées.

La fonction agit comme une sorte de temps qui retourne *vrai* à des instants occurrents périodiquement, que l'on peut stocker dans une variable binaire

" *coup de cadence*",

qui peut être vraie ou fausse.

On peut utiliser ce comportement en combinaison avec une équation continue pour déclencher des signaux périodiques.

La fonction durable *sample* retourne toujours *faux* entre les signaux qu'elle génère, c'est-à-dire durant la solution de la partie continue de l'événement faisant partie de la simulation entre les signaux.

L'instant de départ *start* et l'intervalle *interval* doivent être des entités paramètres, c'est-à-dire être constantes durant toute une simulation, et doivent être des types *proportionnelles* ou *échelonnées,* ou des sous-types de ces types, par exemple *temps* qui est un sous-type de *proportionnelle*.

Clock()

Cette fonction génère des signaux instantanés cadencés.

Elle permet de représenter des cadenceurs.

sample(...)

C'est la version durable de la version instantanée.

Elle génère une information continue aux coups d'une cadence et retourne une entité instantanée durable.

La version instantanée de sample() est relativement différente de de la version durable sample(...).

sample(entité, cadence)

est une entité continue.

l'entrée "*entité*" représente une entité de type "*continu*".

l'entrée "*cadence*" est une entité de type "*cadence*".

La version instantanée de sample(entité, cadence) ou sample(entité) retourne une version instantanée de la valeur de l'entité qui a le temps associé "*temps*" et la valeur de la limite de "*cadence*" quand le temps "*temps*" est activé.

C'est la valeur de l'entité immédiatement avant que le signal "*temps*" ne soit activé.

Si l'entrée optionnelle "*temps*" n'est pas fournie, le temps est inféré par le simulateur.

hold(...)

subSample(), superSample(), shiftSample()

interval()

noEvent()

smooth(...)

pre(...)

previous(...)

edge()

delay()

Le traitement des signaux

GIANNI

Tu peux me faire un petit résumé des principales représentations possibles du fonctionnement des signaux, autrement dit pour représenter le fonctionnement d'une signalisation?

JEAN

Face à une réalité que je cherche à comprendre, je commence en général à la découper en deux entités fondamentales:

- une entité particulière, et,

- un environnement.

Ainsi je peux distinguer deux grands types de signaux:

- ceux qui sont générés dans l'entité particulière elle-même, et,

- ceux qui apparaissent naturellement dans son environnement.

Puis, je me demande comment ces signaux sont gérés, c'est-à-dire que je me demande quels comportements ils impliquent.

GIANNI

Parfait.

Comment fait tu pour choisir entre une réflexion incertaine et une réflexion cadencée pour représenter tes réflexions?

JEAN

C'est une question d'humeur.

Mais je peux te donner les indications suivantes.

Réflexion discontinue aléatoire versus réflexion cadencée

- si la réflexion contient des signaux contrôlés par une cadence, j'adopte une représentation cadencée.

- si la réflexion donne des idées principalement continues, j'adopte le style incertain.

- si la réflexion donne surtout des idées incertaines, j'adopte le style incertain.

GIANNI

Autrement dit:

- le style incertain est plus flexible en ce sens qu'il permet des réflexions sur des entités totalement aléatoires;

- le style cadencé exige de faire des groupes d'équations: une partie continue et une partie contenant un certain nombre de parties cadencées, une pour chaque cadence, les entités ayant la même cadence devant être placées dans la même partie cadencée.

Dans les autres parties, les entités ne peuvent pas être changées par une cadence non attitrée, elle ne peuvent qu'être interrogées avec des opérateurs cadencés comme sample, hold, subSample, superSample, noClock, ...

JEAN

Continue.

GIANNI

Définition de quand-équations incertaines par des si-équations

Les quand-équations peuvent être définies par une combinaison de:

- une si-équation;

- une idée binaire ayant la valeur initiale appropriée, vraie ou fausse;

- un appel à *edge* dans les conditions de la si-équation.

équation
quand
Anticipation > CoursDeAction.start
alors
comportement
fin du quand

Cette quand-équation est équivalente à la si-équation suivante:

binaire Décision(start = Anticipation.start > CoursAction.start);
équation
Décision = Anticipation > CoursAction.start;
si
edge(Décision)
alors
comportement
fin du si

L'entité binaire "*Décision*" est nécessaire car l'opérateur edge() ne peut être appliqué qu'à des entités binaires.

La condition de la quand-équation est égalée à une entité binaire, qui devient l'entrée d'un opérateur edge() dans la condition de la si-équation.

L'opétateur edge() dans la condition d'une si-condition assure que les équations dans la si-équation sont activées uniquement quand la si-condition devient vraie, c'est-à-dire seulement aux instants de signaux.

Changements discontinus à l'apparition de signaux

C'est souvent le cas que des entités changent discontinument à des signaux.

Selon le type d'entité, on peut choisir entre deux mécanismes pour infliger un changement direct dans une quand-équation ou une quand-déclaration.

- la valeur d'une entité incertaine peut être changée en plaçant l'idée à gauche de l'équation dans une quand-équation, ou à gauche d'une déclaration dans une quand-déclaration.

- la valeur d'une entité cadencée peut être changée en plaçant l'entité en question dans une équation contenue dans une quand-équation ou dans une équation qui sera résolue par le simulateur dans une partie cadencée.

- la valeur d'une entité continue qui apparait dans une équation de variation temporelle, c'est-à-dire une idée dynamique du modèle, qui apparaît dans le modèle de l'événement précédée de l'opérateur der() qui lui est appliqué, peut être instantanément changée par une réinit()-équation dans une quand-équation.

Usage de la priorité des signaux pour éviter les définitions multiples

Deux quand-équations ou quand-affectations incertaines dans différentes parties temporelles peuvent ne pas définir la même entité.

Sans une règle de priorité un conflit entre équations peut apparaître si les deux conditions deviennent vraies au même instant.

Synchronisation des signaux et propagation

Les signaux doivent souvent être synchronisés et propagés.

Le simulateur n'offre aucune garantie que deux différents signaux indépendants n'apparaissent exactement au même instant s'ils ne sont pas explicitement synchronisés, par des équations ou des cadences du type prédéfini "*Cadence*".

Pour garantir la synchronisation entre deux signaux, on utilise le principe de synchronisation disant qu'un calcul à un signal ne prend pas de temps, pas de temps du simulateur donc.

Un signal peut, via une ou plusieurs équations, changer instantanément la valeur de certaines entités discontinue utilisées pour la synchronisation, typiquement des entités binaires ou échelonnées.

Par exemple, il est possible de synchroniser des "*compteurs*" faisant des comptages "*échelonnés*" par définition, à différentes fréquences.

Le type "*Cadenceur*" synchronisée prédéfini du simulateur permet de représenter la synchronisation de signaux cadencés.

C'est la manière la plus efficace de représenter des idées impliquant des cadences devant être synchronisées.

Signaux multiples au même instant et itération de signaux

La discussion ci-dessous concerne les signaux incertains.

Il est important de maintenir un ordre bien défini entre signaux, même s'ils arrivent au même instant et qu'un ordre temporel n'est pas possible dans ce cas.

Deux solutions sont possibles:

- ordre selon les données;

- ordre selon les affirmations (623)

Le modèle séquencé

JEAN

Encore du nouveau?

GIANNI

Non.

Par modèle séquencé il faut comprendre lecture et écriture cadencées.

Je dois formaliser la manière dont tu prélève des informations et passe des ordres dans ta réalité.

En fait tu:

- prélèves de temps en temps des informations, puis,

- fais des raisonnements, puis,

- passes de temps en temps des ordres,

- qui vont à leur tour modifier la réalité que tu cherches à maîtriser,

et ainsi de suite.

On a donc une séquence temporelle entre la prise d'information et le passage à l'action.

Le simulateur doit pouvoir représenter une telle réalité séquentielle.

C'est ce que j'appelle la partie séquentielle du modèle, ou le modèle séquentiel, comme tu préfères.

En fait, le modèle séquentiel n'a qu'un seul type de signal, le signal d'interaction.

Ton action et la réalité restent constantes entre les signaux d'interaction.

Les paquets

Organisation

La pensée organise les idées en paquets.

Ces idées peuvent alors être référencées et importées en évitant la duplication.

Un paquet ne contient que des définitions ou des constantes, pas de comportements.

```
package Types
type Rabbits = Real(quantity="Rabbits", min=0);
type Wolves = Real(quantity="Wolves", min=0);
type RabbitReproduction = Real(quantity="Rabbit Reproduction", min=0);
type RabbitFatalities = Real(quantity="Rabbit Fatalities", min=0);
type WolfReproduction = Real(quantity="Wolf Reproduction", min=0);
type WolfFatalities = Real(quantity="Wolf Fatalities", min=0);
end Types;
```

Dans ce paquet, toutes les valeurs des types sont elles-mêmes de type proportionnée.

En outre, ce paquet ne contient que des définitions de types.

Un modèle peut alors se référer à ces types:

```
model LotkaVolterra "Lotka-Volterra with types"
parameter Types.RabbitReproduction alpha=0.1;
parameter Types.RabbitFatalities beta=0.02;
parameter Types.WolfReproduction gamma=0.4;
parameter Types.WolfFatalities delta=0.02;
parameter Types.Rabbits x0=10;
parameter Types.Wolves y0=10;
Types.Rabbits x(start=x0);
Types.Wolves y(start=y0);
equation
der(x) = x*(alpha-beta*y);
der(y) = -y*(gamma-delta*x);
end LotkaVolterra;
```

Tous les paramètres et les comportements ont maintenant un type spécifique, pas simplement le type prédéfini proportionné.

La pensée peut ainsi ajouter de l'information en plus du fait que ce sont des entités proportionnées.

Par exemple, le fait que les valeurs de ces entités ne doivent pas devenir négatives.

En regardant le modèle lui-même il n'est pas évident de savoir où trouver ces définitions de types.

Le simulateur utilise des règles de consultation pour **consulter** ces définitions.

Le paquet est contenu dans un autre paquet par within:

```
within ModelicaByExample.PackageExamples;
package NestedPackages
"An example of how packages can be used to organize things"
package Types
type Rabbits = Real(quantity="Rabbits", min=0);
type Wolves = Real(quantity="Wolves", min=0);
type RabbitReproduction = Real(quantity="Rabbit Reproduction", min=0);
type RabbitFatalities = Real(quantity="Rabbit Fatalities", min=0);
type WolfReproduction = Real(quantity="Wolf Reproduction", min=0);
type WolfFatalities = Real(quantity="Wolf Fatalities", min=0);
end Types;

model LotkaVolterra "Lotka-Volterra with types"
parameter Types.RabbitReproduction alpha=0.1;
parameter Types.RabbitFatalities beta=0.02;
parameter Types.WolfReproduction gamma=0.4;
parameter Types.WolfFatalities delta=0.02;
parameter Types.Rabbits x0=10;
parameter Types.Wolves y0=10;
Types.Rabbits x(start=x0);
Types.Wolves y(start=y0);
equation
der(x) = x*(alpha-beta*y);
der(y) = -y*(gamma-delta*x);
end LotkaVolterra;
end NestedPackages;
```

Tous les modèles sont contenus dans des paquets.

La clause within signale au simulateur que cela doit constituer un fichier unique indépendant et lui permet de connaître les relations entre les fichiers.

Référencement

La clause within sert simplement à spécifier le paquet parent.

```
within ModelicaByExample.PackageExamples;
model RLC "An RLC circuit referencing types from the Modelica Standard
Library"
parameter Modelica.SIunits.Voltage Vb=24 "Battery voltage";
parameter Modelica.SIunits.Inductance L = 1;
parameter Modelica.SIunits.Resistance R = 100;
```

```
parameter Modelica.SIunits.Capacitance C = 1e-3;
Modelica.SIunits.Voltage V(fixed=true, start=0);
Modelica.SIunits.Current i_L(fixed=true, start=0);
Modelica.SIunits.Current i_R;
Modelica.SIunits.Current i_C;
equation
i_R = V/R;
i_C = C*der(V);
i_L=i_R+i_C;
L*der(i_L) = (Vb-V);
end RLC;
```

La pensée n'a pas défini les types dans ce paquet. Elle se réfère à des types définis dans la bibliothèque standard qui commence par Modelica.

Les types référés dans ce modèle sont:

```
type Voltage = ElectricPotential;
type Inductance = Real(final quantity="Inductance",
final unit="H");
type Resistance = Real(final quantity="Resistance",
final unit="Ohm");
type Capacitance = Real(final quantity="Capacitance",
final unit="F", min=0);
type Current = ElectricCurrent;
```

Eux-même référant les définitions de base:

```
type ElectricPotential = Real(final quantity="ElectricPotential",
final unit="V");
type ElectricCurrent = Real(final quantity="ElectricCurrent",
final unit="A");
```

Lorsque des entités sont définies par des types, le simulateur vérifie systématiquement si la gauche d'une équation est consistante avec la droite par rapport aux unités, détecte automatiquement les erreurs de modélisation.

Importation

Outre savoir comment référer des types définis dans d'autres paquets elle peut utiliser la notion de import qui lui permet d'utiliser une définition comme si elle était définie localement.

```
within ModelicaByExample.BasicEquations.CoolingExample;
model NewtonCoolingWithTypes "Cooling example with physical types"
// Types
type Temperature=Real(unit="K", min=0);
type ConvectionCoefficient=Real(unit="W/(m2.K)", min=0);
type Area=Real(unit="m2", min=0);
type Mass=Real(unit="kg", min=0);
type SpecificHeat=Real(unit="J/(K.kg)", min=0);

// Parameters
parameter Temperature T_inf=298.15 "Ambient temperature";
parameter Temperature T0=363.15 "Initial temperature";
parameter ConvectionCoefficient h=0.7 "Convective cooling coefficient";
```

```
parameter Area A=1.0 "Surface area";
parameter Mass m=0.1 "Mass of thermal capacitance";
parameter SpecificHeat c_p=1.2 "Specific heat";

// Variables
Temperature T "Temperature";
initial equation
T = T0 "Specify initial value for T";
equation
m*c_p*der(T) = h*A*(T_inf-T) "Newton's law of cooling";
end NewtonCoolingWithTypes;
```

La pensée peut importer les définitions en une fois depuis la bibliothèque et les utiliser sans avoir à spécifier tout le chemin:

```
within ModelicaByExample.PackageExamples;
model NewtonCooling
"Cooling example importing physical types from the Modelica Standard
Library"
import Modelica.SIunits.Temperature;
import Modelica.SIunits.Mass;
import Modelica.SIunits.Area;
import ConvectionCoefficient = Modelica.SIunits.CoefficientOfHeatTransfer;
import SpecificHeat = Modelica.SIunits.SpecificHeatCapacity;

// Parameters
parameter Temperature T_inf=300.0 "Ambient temperature";
parameter Temperature T0=280.0 "Initial temperature";
parameter ConvectionCoefficient h=0.7 "Convective cooling coefficient";
parameter Area A=1.0 "Surface area";
parameter Mass m=0.1 "Mass of thermal capacitance";
parameter SpecificHeat c_p=1.2 "Specific heat";

// Variables
Temperature T "Temperature";
initial equation
T = T0 "Specify initial value for T";
equation
m*c_p*der(T) = h*A*(T_inf-T) "Newton's law of cooling";
end NewtonCooling;
```

Ici la pensée a remplacé les définitions par des import.

```
import Modelica.SIunits.Temperature;
```

importe la temperature dans le modèle courant.

Par défaut, le nom de ce type importé sera le dernier nom du chemin.

Cela signifie qu'avec le import la pensée peut simplement utiliser le nom Temperature et le simulateur remonte tout seul à sa définition dans la bibliothèque.

```
import ConvectionCoefficient = Modelica.SIunits.CoefficientOfHeatTransfer;
```

La syntaxe est un peu différente ici: au lieu de créer un type local basé sur le nom la pensée spécifie que le type local doit être ConvectionCoefficient.

Ceci permet à la pensée d'utiliser le nom qu'elle utilisait avant.

Cela lui évite de refactoriser le code qui utilisait un nom auparavant.

Une autre raison pour définir un nom alternatif, autre que celui que le simulateur assignerait normalement, est d'éviter les collisions.

Par exemple, si la pensée veut importer deux types de deux paquets.

```
import Modelica.SIunits.Temperature; // Celsius
import ImperialUnits.Temperature;    // Fahrenheit
```

cela donnerait deux types nommés Temperature.

En utilisant un alias:

```
import DegK = Modelica.SIunits.Temperature; // Kelvin
import DegR = ImperialUnits.Temperature;    // Rankine
```

la confusion disparait.

Mais c'est une mauvaise pratique que d'importer des unités différentes.

Le modélisateur devrait toujours utiliser des unités SI et ne jamais utiliser d'autre système d'unité.

S'il veut entrer des données ou afficher des résultats, il doit utiliser l'attribut displayUnit.

Il est possible d'importer toutes les unités une à la fois.

Le joker, caractère de remplacement "*" permet d'importer tous les types d'un paquet d'un coup.

```
import Modelica.SIunits.*;
```

Cette instruction importe toutes les unités dans le modèle courant.

Avec le joker, il n'est pas possible de renommer les types. Ils auront localement exactement le même nom qu'ils ont dans le paquet nommé.

Caveat

Il y a trois formes d'importation.

Dans tous les cas il y a création d'un alias dans la définition qui réfère un nom défini à un autre endroit.

La première forme importe simplement une définition par son nom entier:

```
import Modelica.SIunits.Temperature;
```

La conséquence en est que des références locales au nom Temperature sont plaquées sur le nom entier.

Avec cette forme d'importation, le nom de l'alias reproduit le dernier élément du nom entier.

Dans certains cas la pensée peut vouloir un autre alias et introduire un nom alternatif:

```
import DegK = Modelica.SIunits.Temperature; // Kelvin
```

Après une telle importation, elle peut utiliser le nom DegK pour référer la Temperature.

Cette pratique évite les collisions et facilite la lecture du modèle.

Finalement, il est possible d'importer toutes les définitions d'un paquet dans le modèle courant avec le caractère de remplacement:

```
import Modelica.SIunits.*;
```

Une telle importation crée autant d'alias qu'il y a de définitions dans le paquet supérieur et il n'est pas possible d'assigner un nom alternatif pour l'alias: chaque alias a le même nom que l'original.

Le joker est donc considéré comme dangereux car la pensée ne peut pas renommer un type d'entité.

En outre, une telle importation permet de remonter à la définition initiale et de retrouver la définition associée à un certain type.

Si elle veut importer de multiples entités du même paquet, elle peut utiliser l'importation:

```
import Modelica.SIunits.{Temperature, Length};
```

qui a été introduite dans le simulateur pour faire des importations multiples en évitant le joker.

Enfin, l'importation par un nom entier n'a pas besoin d'être tapée dans une interface graphique.

Résumé

Définition des paquets

```
package PackageName "Description of package"
// A package can contain other definitions or variables with the
// constant qualifier.
end PackageName;
```

Les paquets peuvent être emboités

```
package OuterPackage "A package that wraps a nested package"
// Anything contained in OuterPackage
package InnerPackage "A nested package"
// Things defined inside NestedPackage
end InnerPackage;
end OuterPackage;
```

L'emboîtement des paquets dans un fichier comme une série de paquets emboités, est une mauvaise pratique pour deux raisons:

- le fichier obtenu est très indenté et difficile à lire, et,

- du point de vue de l'entretien il est bien plus judicieux de découper les idées en morceaux.

Enregistrement

Le code source doit correspondre au système de fichiers.

La manière la plus simple est de tout enregistrer dans un seul fichier.

Ce fichier aura un suffixe .mo.

Un tel fichier peut contenir que la définition d'un seul modèle ou de toute une hiérarchie de modèles.

Tout enregistrer dans un seul fichier n'est pas une bonne idée.

Le mieux est de projeter les définitions de Modelica dans une structure de fichiers.

Un paquet peut être enregistré comme un classeur en utilisant le même nom que le paquet pour le classeur.

Puis, dans ce classeur il doit y avoir un fichier .mo qui enregistre la définition du paquet mais pas de définition emboitée.

Les définitions emboitées peuvent être stockées soit dans des fichiers indépendants soit dans des dossiers représentant des paquet.

```
/RootPackage                # Top-level package stored as a directory
  package.mo                # Indicates this directory is a package
    NestedPackageAsFile.mo    # Definitions stored in one file
   /NestedPackageAsDir       # Nested package stored as a directory
   package.mo              # Indicates this directory is a package
```

Le paquet package.mo aura l'air suivant:

```
within;
package RootPackage
// only annotations can be stored in a package.mo
end RootPackage;
```

La clause within doit être présente mais vide.

Elle indique que ce paquet n'est pas contenu dans aucun autre paquet.

Similairement, le fichier package.mo file associé au paquet NestedPackageAsDir ressemblerait à ça:

```
within RootPackage;
package NestedPackageAsDir
// only annotations can be stored in a package.mo
end NestedPackageAsDir;
```

De nouveau, il ne doit pas y avoir de définition dans ce paquet, que des annotations.

La clause within est un peu différente, réflectant le fait que NestedPackageAsDir appartient au paquet RootPackage.

Finalement, le fichier NestedPackageAsFile.mo ressemblerait à ça:

```
within RootPackage;
package NestedPackageAsFile
// The following can be stored here including:
//  * constants
//  * nested definitions
//  * annotations
end NestedPackageAsFile;
```


```
within RootPackage;
package NestedPackageAsFile
// The following can be stored here including:
//  * constants
//  * nested definitions
//  * annotations
end NestedPackageAsFile;
```


La clause `within` est la même que pour la definition du paquet `NestedPackageAsDir`, mais puisqu'on enregistre ce paquet dans un seul fichier, les définitions emboitées pour les constante, les modèles, les paquets etc. sont autorisées aussi bien ici.

Ordre des fichiers

When all definitions are stored within a single file, the order they appear in the file indicates the order they should appear when visualized (*e.g.*, in a package browser). But when they are stored on the file system, there is no implied ordering. For this reason, an optional `package.order` file can be included alongside the package.mo file to specify an ordering. The file is simply a list of the names for nested entities, one per line. So, for example, if we wanted to impose an ordering on this sample package structure, the file system would be populated as follows:

Quand toutes les définitions sont enregistrées dans un seul fichier, l'ordre dans lequel elles apparaissent dans le fichier indiquent l'ordre dans lequel elles doivent apparaitre dans le navigateur de paquets.

Mais quand elles sont enregistrées dans le système de fichiers, il n'y a pas d'ordre à priori.

Pour cette raison un fichier `package.order` peut être inclus avec le fichier `package.mo` pour spécifier un ordre.

Ce fichier est simplement une liste de noms pour les entités emboitées, un par ligne.

Ainsi, par exemple, si la pensée veut imposer un ordre dans une structure de paquets, le système de fichiers serait comme suit:

```
/RootPackage                # Top-level package stored as a directory
  package.mo                # Indicates this directory is a package
  package.order             # Specifies an ordering for this package
    NestedPackageAsFile.mo    # Definitions stored in one file
  /NestedPackageAsDir         # Nested package stored as a directory
    package.mo              # Indicates this directory is a package
    package.order           # Specifies an ordering for this package
```

En l'absence du fichier `package.order` le simulateur trie les paquets alfabétiquement.

Si le modélisateur veut ordonner les contenus de `RootPackage` en ordre inverse le l'ordre alphabétique le fichier `package.order` aurait la forme suivante.

```
NestedPackageAsFile
NestedPackageAsDir
```

Cela indiquerait au simulateur que le fichier `NestedPackageAsFile` devrait venir avant `NestedPackageAsDir`.

Versioning

La plupart des outils du simulateur permettent au modélisateur d'ouvrir un fichier soit en spécifiant le chemin complet en utilisant le dialogue de sélection.

Mais il peut être embêtant de trouver et charger une quantité de fichiers différents chaque fois.

Pour cette raison, le simulateur défini un environnement spécial appelé MODELICAPATH que le modélisateur peut utiliser pour spécifier la position du code source qu'il faut utiliser.

La variable d'environnement MODELICAPATH doit contenir la liste des fichiers à examiner.

Sur Mac, la séparation est ":".

Quand le simulateur arrive sur un fichier qu'il n'a pas encore chargé, il examine les fichiers de MODELICAPATH pour trouver. Par exemple:

```
/home/mtiller/Dir1:/home/mtiller/Dir2
```

et si le simulateur cherche un paquet `MyLib` il cherche d'abord dans le dossier `home/mtiller/Dir1` pour un paquet nommé `MyLib.mo` enregistré dans un seul fichier ou dans le dossier `MyLib` qui contenait un fichier `package.mo` qui définit une paquet nommé `MyLib` .

Si aucun ne peut être trouvé, il chercherait les mêmes choses dans le dossier `/home/mtiller/Dir2`.

Resources

Dans ne nombreux cas, il est utile d'ajouter des resources (données, scripts, images).

Ces resources peuvent être référées par une URL.

De telles URL permet de définir des locations de ressources relatives au simulateur dans le système de fichiers.

```
/RootPackage                # Top-level package stored as a directory
  package.mo                # Indicates this directory is a package
  package.order             # Specifies an ordering for this package
    NestedPackageAsFile.mo   # Definitions stored in one file
  /NestedPackageAsDir        # Nested package stored as a directory
   package.mo              # Indicates this directory is a package
   package.order           # Specifies an ordering for this package
     datafile.mat             # Data specific to this package
 /Resources                 # Resources are stored here by convention
            logo.jpg                # An image file
```

Si le modélisateur a besoin d'informations contenues dans le `NestedPackageAsDir`, il peut utiliser la référence suivante:

```
modelica://RootPackage/NestedPackageAsDir/datafile.mat
```

Une telle URL commence par .

Ceci indique que la ressource référencée l'est par rapport à un modèle et non, par exemple, quelque-chose à récupérer sur le web.

Autre exemple:

```
modelica://RootPackage/Resources/logo.jpg
```

La convention est d'enregistrer des resources liées à une bibliothèque dans un paquet emboité nommé `Resources`, d'où les valeurs par défaut de `IncludeDirectory` et de `LibraryDirectory`.

Règles de consultation

Architectures

Bibliothèque standard

Dictionnaire

- A -

Analyse
Analysis

Analyse fondamentale

Prédire les sous-jacents.

Analyse rationnelle

Mixte technique complet financier.

Analyse sectorielle

Techno, pharmacie, banque.

Analyse technique

Chandeliers japonnais.

Assignation
Assignation

Assigner une valeur à une réalité en faisant une assignation:

réalité **:=** expression

L'expression ne doit pas avoir une variabilité supérieure à la réalité à gauche.

Attributs
Attributes

Attribut d'une réalité

- B -

- C -

Cadence
Cadence

Donner une cadence qui peut être régulière ou irrégulière.

Considération
Consideration

Regarder ensemble, donc deux mains, c'est-à-dire une équation.

main gauche = main droite

Contact
Contact

Les contacts sont représentés par des équations ayant le préfix *connect*:

contact = **connect**(contact1, contact2)

- D-

Déclaration
Declaration

Déclarer l'existence d'une réalité.

- E -

Equation
Equation

Deux mains.

main gauche = main droite

Une équation contraint la valeur de la partie gauche à être égale à la valeur de la partie droite.

Expression
Expression

Toutes les expressions ont un type.

Ce type est obtenu à partir des types de ses parties constitutives.

La vérification automatique de types lors de la vérification du modèle empêche de mettre types incompatibles à gauche et à droite d'un signe égal:

<même type> = <que ce type>

- F -

Fonction
Function

Une section algorithmique qui contient du code procédurier constitué d'une suite d'instructions à exécuter lorsque la fonction est appelée.

Les entrées de la fonction sont précédées du préfixe *input* et ses sorties du préfixe *output*.

Les fonctions n'ont pas de mémoire et retournent toujours le même résultat si elles sont appelées avec les mêmes entrées.

Cette syntaxe ressemble à celle des blocs.

- G -

- H -

- I -

Idée
Idea

Toute idée d'existence d'une réalité a la structure suivante:

<préfixes> <type> <nom> <spécification des attributs>

Initialiser, initialisation
Initialize, initialization

Donner une valeur initiale à une instance.

Instancier, instanciation, instance
Instanciate, instanciation, intstance

Créer un spécimen d'un réalité dont il existe déjà un modèle dans le simulateur.

- J -

- K -

- L -

- M -

Modifier, modification
Modify, modification

Une équation de modification remplace l'équation de déclaration.

Normalement utilisée pour modifier les attributs d'une instance.

- N -

- O -

- P -

- Q -

- R -

- S -

- T -

- U -

- V -

- W -

- X -

- Y -

- Z -