Architecture du projet

16 juin 2015

Résumé

Nous commençons par définir les concepts utilisés dans ce projet de simulation d'ifttt. En quelque sorte, une fois que les mots auront un sens, nous proposons une ébauche de formalisation mathématique puis nous détaillons notre proposition technique. Cette dernière s'appuie sur la philosophie de développement KISS : *Keep It (the actor) Simple Stupid* et sur l'*imitation de la nature* (la bionique). La combinaison de ces deux idées nous semble particulièrement féconde. Nous abordons enfin de manière plus spécifique quelques questions, dont le sujet de thèse de Robin (prévenir les situations aberrantes). L'ébauche de formalisation peut être ignorée en première lecture.

Composition de ce document Ce document est originellement composée avec LaTeX (au format `tex`). Deux formats de sortie sont proposés : `pdf` et `md` (Markdown). Le fichier `*.tex` est toujours à jour ; il est suivi de près par le `*.pdf` qui est le plus agréable à l'œil. Le fichier `*.md` est susceptible d'accuser parfois un petit retard mais est adapté pour une lecture directement sur GitHub ; il est écrit par la commande `pandoc -s docs/Architecture.tex -o docs/Architecture.md`.

Table des matières

1 Introduction

(to be expended)

Panorama général présentation de la programmation orientée acteur, de play, akka et ifttt.

Points saillants Nous confessons que l'architecture de cette branche n'est pas aussi simple que KISS. Il nous semble que la relative difficulté à passer outre la complexité de cette architecture procure quelques avantages :

— L'utilisation des dernières nouveautés de Java 8 nous apporte une grande finesse de simulation et améliore la compréhension intuitive du code ;
— Les étiquettes sont capables de rendre compte d'une grande variété de types de groupe ;
— La génération à la volée des messages d'action autorise de riches interactions entre le destinateur et le destinataire tout en respectant le modèle acteur ;

— L'accès au système au travers d'un proxy rendra probablement presque indolore le passage à une simulation distribuée sur plusieurs machines virtuelle Java ;
— L'envoi aléatoire de messages paramétrés permet de confronter un ensemble d'acteurs à des configurations exotiques qui n'auraient peut-être pas été prises en compte par une programmation déterministe.

2 Présentation des concepts

Nous voulons faire une simulation de ifttt. Pour cela, nous utilisons deux niveaux d'abstraction en fonction desquels nous présentations les concepts utilisés par la suite.

2.1 Le niveau des modèles

Ce niveau est le plus abstrait et définit des catégories dont on donnera en quelque sorte des réalisations [1] dans le niveau inférieur.

Canal Un canal (`models.Channel` dans le code) est une catégorie d'éléments de l'internet des objets [2] : ce peut être une classe d'objets physiques ou immatériels comme un capteur, une lampe, le compte d'un service en ligne, un site de news. Comme dans ifttt, des canaux ont des signaux (triggers en anglais et `models.Trigger` dans le code) et des actions.

Signal Les signaux sont levés par le canal pour signaler un changement : ce peut-être un changement d'environnement (l'ouverture d'une porte, la détection d'un mouvement) ou un changement interne (un délai expire, une date échoit). Sémantiquement, un signal mentionne généralement un objet et le nouvel état de ce dernier ou ce qui a provoqué ce changement d'état [3]. Il peut mentionner des modalités (dans le cas d'une porte, on peut mentionner son degré d'ouverture).

Recette Toujours de la même manière que ifttt, des recettes (`models.Recipe`) peuvent être créées pour matérialiser des réactions automatiques à un changement. Le slogan *if this then that* montre bien que si un événement arrive, alors un canal va agir d'une certaine manière : une recette matérialise une relation de causalité entre deux acteurs. Outre son nom, son propriétaire et d'autres attributs, elle contient principalement quatre membres : un canal émetteur, un signal émis, un canal récepteur et une action ; le canal émetteur lève un de ses signaux et à cause de cela, le canal

1. Au sens philosophique du terme : rendre réel ; synonyme de l'anglais « to implement ».
2. Plus précisément : une catégorie d'objets de l'internet
3. Par exemple : *la porte est ouverte* pour un capteur de porte ou bien *quelque chose a bougé* pour un détecteur de mouvement

récepteur accomplit l'action précisée dans la recette. Contrairement à certains de ses concurrents, une recette de ifttt ne peut lier plus d'un canal émetteur, un signal émis, un canal récepteur et une action : c'est atomique.

Sémantique Sémantiquement, les actions contiennent un verbe, un objet et si besoin des modalités complémentaires. Le sujet de l'action est toujours le canal récepteur. Par exemple, une action simple est : *allume la lumière* et une action plus élaborée *allume la lampe en jaune, en mode économie d'énergie et à moyenne intensité*. Une recette lie de manière statique un unique élément pour chacun de ses quatre membres principaux : la sémantique est donc figée. En revanche, les modalités de l'action peuvent être définies en fonction de celles du signal émis. Si l'on considère la recette actuelle comme une bijection, on propose une évolution possible de ce modèle pour avoir une surjection, une injection ou encore autre chose.

2.2 Le niveau des acteurs

Le niveau des acteurs est en quelque sorte une représentation concrète du monde. Il contient des acteurs qui évoluent indépendamment et communiquent par message. La communication entre ces acteurs suit le formalisme indiqué plus haut.

Relation avec un canal Le mode d'action d'un acteur est définit par un canal. Un acteur peut donc être vu comme une instance d'un canal.

Message Les messages échangés par des acteurs ont la sémantique soit d'un signal, soit d'une action. De la même manière qu'une lettre à la Poste, ils peuvent être envoyés anonymement mais ont forcément un destinataire. Les acteurs ne communiquent que par message. Les messages peuvent contenir des modalités qui précisent le changement décrit par le signal ou l'action à effectuer.

Relation de causalité A ce niveau concret, une recette est définie comme relation de causalité entre deux acteurs et « if this then that » devient : quand un acteur envoit un message qui contient un signal, alors un autre acteur reçoit un message qui lui dit d'accomplir telle action. Dans le monde physique, un capteur ne sait faire qu'une chose : lever des signaux. Pour respecter la simplicité des objets physiques, un pseudo-acteur est définit (`actors.Commutator` dans le code) pour permettre aux acteurs de lever des signaux simplement en envoyant un message à icelui. C'est formellement un objet, pas un acteur. Les recettes sont « diluées » dans ce pseudo-acteur : à réception d'un message signal d'un acteur, le pseudo-acteur regarde si une relation de causalité existe. Le cas échéant, il envoit anonymement un message d'action à l'acteur définit.

Pseudo-acteur Ce pseudo-acteur est utile pour envoyer des messages signaux et simuler des changements. La période caractérisant l'envoi de ces messages est définit par une loi de probabilité qui peut être fixe (toutes les trente secondes) ou plus évoluée : lois de Bernouilli, Gauss, Poisson...

Alternative à cette architecture Pour plus de détail sur les alternatives à cette architecture, voir plus bas le paragraphe 5.1.

2.3 Relation de génération entre les deux niveaux

Nous définissons plus haut les acteurs par rapport aux canaux. Cette démarche est cependant erronnée à cause de akka qui nous force en quelque sorte à définir programmatiquement les acteurs : leur définition n'est donc pas chargée en base mais codée « en dur » et iceux ne sont donc pas déduits des canaux comme le présentaient les explications précédentes.

Loin de poser problème, il suffit de considérer que les acteurs ont tout de même une intelligence qui échappe aux canaux et correspondent à des objets physiques qui ne sont pas facilement modifiables. Le paradigme d'ifttt est d'en abstraire des catégories en créant un canal qui regroupe des acteurs tandis que celui de l'utilisateur est de redescendre du canal vers son objet physique. La dichotomie de ces deux mouvements de pensée (induction d'ifttt et déduction de l'utilisateur) renvoie à un phénomène déjà existant dans ifttt : il est donc naturel que nous le rencontrions également.

En conclusion, les éléments des deux niveaux doivent être définis en étroite accointance.

3 Présentation technique

3.1 Conception générale

Pour générer des objets à la volée, ce code utilise le patron de conception de la fabrique. D'abord explicite, il est maintenant sous-jacents des λ-expressions qui ont l'avantage d'être bien plus lisibles. On peut définir à la volée un comportement à adopter, un message, une période : tout cela peut donc changer à chaque fois.

Java 8 permet une manipulation intuitive des collections de données grâce au λ-calcul (`filter`, `map`, `flatMap`)

3.2 Manipuler facilement le système d'acteurs avec `SystemProxy`

A décrire : à quel problème ça répond, qu'est-ce que ça fait, comment ça le fait ?

3.3 Construire à la volée un message d'action en fonction d'un message émis avec `MessageMap`

Du Java 8, de la réflexion, de la généricité...bon appétit. Savoir comment ça se passe à l'intérieur n'est pas important, on veut juste savoir à quoi ça sert.

3.4 Etablir à la volée une ligne téléphonique entre deux acteurs avec `Commutator`

A décrire : à quel problème ça répond, qu'est-ce que ça fait, comment ça le fait ?

Tordons le cou à une fausse idée : `Commutator` utilise une réalisation de l'interface `LookUpEventBus`. Cette interface fonctionne avec un patron éditeur / lecteur (« publisher / subscriber » en anglais). En revanche, le commutateur porte bien son nom puisse qu'il a pour effet d'établir comme une ligne téléphonique entre deux acteurs. De la même manière que dans la boucle locale, il n'y a pas n^2 lignes pour relier n utilisateurs du téléléphone mais que la commutation par circuit donne cette illusion, l'objet `Commutator` émule cette situation : dans le cas d'une relation de causalité strictement bijective, il ne peut y avoir qu'un destinateur et un destinataire. En revanche, le patron éditeur / lecteur est effectif si plusieurs relations de causalité prennent pour source la même classe de message et le même acteur.

(to be expanded) Les attributs des messages doivent être objets et non des types primitifs pour pouvoir être nul. Si nul, l'attribut correspondant de l'acteur n'est pas changé.

3.5 Définir un envoi de message automatique avec `RandomScheduler`

En faisant un parallèle avec la vie réelle, il y a principalement trois manières d'initier un message automatique :

— La manière la plus simple est de commander une action, donc d'envoyer un message d'action à un acteur ;
— On peut aussi simuler l'émission d'un signal par un acteur ;
— Enfin, si un objet est suffisament évolué, il peut envoyer des messages tout seul et on peut le faire agir en ce sens.

La classe `RandomScheduler` propose les deux premières façons de faire. La troisième est laissée au soin du lecteur.

Cet envoi automatique peut être interrompu par l'utilisateur qui peut également donner un nombre ou un temps limite d'envoi.

4 Formalisation des recettes : vers une généralisation ?

Nous commençons par faire quelques rappels mathématiques puis nous définissons ce qu'est une recette et une relation de causalité. Une fois cela fait, nous voyons

comment les deux notions peuvent être liées.

4.1 Rappels mathématiques

On rappelle qu'une application est une relation mathématique entre deux ensembles pour laquelle chaque élément du premier est relié à un unique élément du second. On rappelle qu'une relation dans un ensemble E est caractérisée par un sous-ensemble du produit cartésien $E \times E$, soit une collection de doublet d'éléments de E.

4.2 Définitions

Soient S et C respectivement les ensembles des signaux et des canaux. L'ensemble S se décompose en deux parties exclusives S_E et S_R car un signal est soit émis (le premier) soit reçu (le second). Un canal peut avoir plusieurs signaux. Un canal sans signal est comme une soupe sans sel : c'est moins bon puisqu'il ne peut pas communiquer.

Une recette de rang (m, n) est une relation de $(S \times C)^m$ dans $(S \times C)^m$ qui lie m doublets à n autre doublets. Une recette dont le rang n'est pas récisé est une recette de rang $(1, 1)$. Une recette est dite réalisable si et seulement si :

— Pour tout doublet (s, c) de $(S \times C)^m$, s est élément de E (c'est un signal émis) ;

— Pour tout doublet (s, c) de $(S \times C)^n$, s est élément de R (c'est un signal reçu) ;

Pour définir clairement les choses, une recette (de rang $(1, 1)$) est une application dans $S \times C$ qui lie un doublet $d_1 = (e, c_1)$ à un autre doublet $d_2 = (r, c_2)$. Une telle recette est dite réalisable si et seulement si e appartient à E et r à R. On notera incidemment que les canaux impliqués dans une recette ne sont, suivant cette définition, pas forcément, tous différents.

D'autre part, soit M et A respectivement l'ensemble des espaces de messages et l'ensemble des espaces acteurs. Tout élément de A est un acteur, qui peut envoyer et recevoir des messages. Tout élément de l'ensemble M des espaces des message appartient exclusivement à l'un des deux sous-ensembles M_E et M_A car un message contient une sémantique particulière : il est envoyé par un acteur après un évènement ou reçu par un acteur pour accomplir une action.

Précisons tout de suite cette formulation d'« ensemble d'espaces » qui peut sembler lourde et inutile. Par la suite nous parlerons pour alléger les phrases d'ensemble de classes de messages. Si nous détaillons pour les messages, une explication du même acabit vaut aussi pour les acteurs. Peut-être pourrions-nous pour les messages parler d'ensemble de classes. Tout message a une sémantique particulière : il est d'une

classe donnée [4] mais chaque message possède ses propres modalités [5]. Chaque espace regroupe les messages qui ont la même sémantique. Chacun de ces espaces est de dimension le nombre de modalités des message de cette sémantique.

Une relation de causalité de rang (m, n) est définie sur $(M \times A)^m$ dans $(M \times A)^m$ et lie m doublets « d'émission » à n doublets « de réception ». Une relation de causalité dont le rang n'est pas précisé est de rang $(1, 1)$. Par un abus de langage bien pratique, on définit les types de rang suivants :

— Une relation de causalité de rang $(m, n), m > n$ est une injection, dite stricte quand $n = 1$;

— Une relation de causalité de rang $(m, n), n > m$ est une surjection, dite stricte quand $m = 1$;

— Une relation de causalité de rang $(m, n), m = n$ est une bijection, dite stricte quand $m = n = 1$.

Puisque les acteurs envoient et recoivent des messages et non des classes de message, il faut également munir la relation de causalité d'une fonction reçoit les messages émis et envoient des messages d'action. Je ne sais pas comment définir proprement les ensembles de départ et d'arrivée de cette fonction mais ce qui est sûr, c'est que ces arguments sont m messages de classes éléments de M et produit comme résultat au plus n messages sur l'ensemble d'arrivée. Cela veut dire que la fonction reçoit un message de chaque acteur de départ liés par la relation de causalité mais, en fonction de ses entrées, n'envoit pas forcément un message à tous les acteurs d'arrivée de la relation.

Ceci peut sembler attirant sur le papier mais une facette importante des relations de causalité reste à traîter : notre mimétisme du réel nous impose d'ajouter à une relation de causalité un délai de traitement : ce délai court à partir du premier message sur m émis par un acteur et les $m - 1$ messages suivants doivent être reçus au plus tard strictement avant l'expiration de ce délai pour déclencher une causalité.

Notre code se borne pour l'instant à proposer des relations de causalité de rang $(1, 1)$ qui sont dont des bijections strictes. Le délai de traitement n'est donc pas utile et n'est pas considéré.

4. Par exemple : la classe des messages qui disent que quelqu'un est entré dans la pièce, ou qui ordonnent à une lampe de s'allumer.

5. Par exemple : la quantité de mouvement détectée, la couleur dont allumer la lampe, la valeur du potentiomètre

5 Quelques explications techniques et fonctionnelles

5.1 Vers un système d'acteurs auto-organisé ?

Le pseudo-acteur défini plus haut n'est pas un acteur, d'où sa dénomination : c'est un objet. Quelles sont les possibilités de se passer d'un tel objet pour un système d'acteurs qui s'organiserait de lui-même ?

Des acteurs plus intelligents Il est possible rendre chaque acteur conscient des relations de causalité qui le lient aux autres. Cette architecture présente des avantages indéniables, puisqu'elle rend le système réellement distribué. Le projet est actuellement réparti en deux branches pour étudier la facilité d'implémentation de chacune. Conceptuellement attirante, elle poserait cependant des questions techniques hardues puisqu'il faudrait définir des « tranches » de pseudo-acteur. Elle s'éloigne en outre un peu plus du monde réel puisque les acteurs recouverts de la tranche de pseudo-acteur ne représente plus véritablement un objet du monde réel.

Toujours plus d'acteurs D'aucun pourrait souhaiter que les classes d'objet définies soient des acteurs (c'est-à-dire réalisent l'interface `UntypedActor`) : cela serait tout à fait possible, aurait l'élégance de montrer qu'ifttt accepte une décomposition KISS et serait enfin plus proche du monde réel. Nous objections que cela ne serait cependant pas sans poser de vrais problèmes conceptuels :

- On attend tout d'abord dans ce projet qu'un acteur puisse être lié par des relations de causalité : serait-il acceptable qu'une relation de causalité lie les acteurs `MessageMap` ou `Commutator`, qui sont utilisés pour caractériser justement ces relations ?
- Si `Commutator` était un acteur alors il perdrait son rôle de commutateur (donc de medium de communication) pour devenir un routeur. En effet, un objet commutateur n'envoie pas de message en son nom propre mais ne fait qu'ouvrir une ligne téléphonique directe entre deux acteurs qui se parlent par l'intermédiaire d'une fonction de traduction. Un acteur devrait plutôt parler en son nom plutôt qu'utiliser la méthode forward(Object message, ActorContext context).
- La classe `RandomScheduler` pourrait effectivement devenir un acteur. Au lieu d'un objet condamné à un certain immobilisme, on pourrait alors considérer l'acteur `RandomScheduler` résultant comme une espèce de Zorro masqué, qui reste discret mais se place derrière un acteur pour lui souffler quoi faire.

Mise en abyme : l'exemple type de la fausse bonne idée Les acteurs tels que définit par akka peuvent contenir d'autres acteurs. On peut pousser la proposition précédente encore plus loin en intégrant les acteurs qui réalisent des canaux dans des meta-acteurs. On s'éloigne alors du monde réel tout en générant des problèmes techniques. Il n'y a donc à cela aucun intérêt.

Nous avons choisi dans cette branche de nous inspirer de la « philosophie de développement » KISS : *keep it simple stupid*. Le nom de la branche vient de là.

5.2 Un défaut conceptuel : le serpent qui se mord la queue

Dans toute discussion sur ce sujet, nous commençons toujours par parler des canaux pour en venir ensuite aux acteurs, définis par rapport aux canaux. Or en réalité nous définissons les acteurs dans le code et les canaux dans la base : pourrions-nous définir et charger dynamiquement les classes des acteurs rendues nécessaires par les canaux ?

Un rapide état de l'art mené dans ce domaine fait état d'un niveau de technicité extrême et d'une complexité faramineuse :

- `http://twit88.com/blog/2007/10/21/compile-and-reload-java-class-dynamically-usi`
- `http://javahowto.blogspot.de/2006/07/javaagent-option.html`
- `http://www.nurkiewicz.com/2009/09/injecting-methods-at-runtime-to-java.html`
- En fait, il semble même que des projets de recherche soient menés en ce sens : `https://github.com/Sable/soot/wiki/Adding-attributes-to-class-files-%28Advanced%29`

Bien que techniquement passionnant, l'analyse que nous faisons de la relation de génération entre les deux niveaux d'abstraction tend à montrer que ce ne serait qu'une inutile fioriture dans l'état d'avancement actuel de ce projet.

5.3 Groupe d'objets et de propriétaire : la notion d'étiquette

Premier jet On peut clairement utiliser plus d'un objet `actors.Commutator`. Il y aura donc des groupes de relations de causalité. On peut facilement définir pour chaque groupe un propriétaire. Attention cependant : créer des groupes de relations de causalité est une chose, rendre l'objet conscient de cette proriété en est une autre. Il ne faut pas définir un propriétaire pour un objet s'il est trop simple pour comprendre ce que cela signifie : « tu es trop jeune mon fils, tu n'es encore qu'un petit no0b ».

Critique Démultiplier le commutateur revient à considérer qu'il y a plusieurs media de communication différents. Lorsqu'un objet émet un message, il doit aller chercher tous les commutateurs qui ont une relation de causalité pour lui. Dans le monde réel,

cela correspondrait à plusieurs opérateurs téléphonique. En réalité, un téléphone n'est relié au réseau que d'un seul opérateur, sauf les téléphones portables qui proposent deux emplacements de cartes SIM : ceci peut donner une image acceptable d'un acteur qui serait lié à des relations de causalité sur deux machines Java différentes.

Exception faîte des téléphones à deux cartes SIM qui correspond à une situation différente, voir tous les relations de causalité d'un même système local passer par le même commutateur est plus respectueux de la philosophie KISS pour les acteurs.

Idée mise en œuvre : les étiquettes Nous proposons donc d'utiliser des étiquettes. Une étiquette est simplement un ensemble (au sens strict) de chaînes de caractères (des mots courts en langage compréhensible) dont on dote les relations de causalité. On peut alors former s'appuyer sur le sens de ces chaînes pour grouper des objets selon un critère spatial (une habitation, une pièce, une ville), d'appartenance (le groupe qui appartient à tel ou tel) ou un critère de proximité conceptuelle (tous les outils de bricolage). Les étiquettes offrent une plus grande richesse et une plus grande finesse de définition par rapport à un attribut `Propriétaire` ou `Groupe d'objet`.

A quoi attacher ces étiquettes ? Nous considérons qu'il est plus judicieux, bien que contre-intuitif, d'attacher ces étiquettes aux relations de causalité. En effet, puisque nous voulons rester près de la réalité que nous cherchons à simuler, il faut bien considérer que beaucoup des objets les plus rudimentaires de l'internet n'ont pas la moindre idée [6] de leur emplacement (quand ils en ont seulement un) ou de leur propriétaire : ces informations sont dans l'esprit des êtres humains qui les utilisent et définissent entre eux des relations de causalité. En outre, il est très tentant de renseigner un attribut sémantiquement très fort (comme un champ référençant l'objet `Propriétaire` lié) pour tous les objets : un tel propriétaire ne serait donc défini que dans la machine virtuelle locale tandis que les acteurs sont mobiles : notre simulation serait alors moins facile à distribuer. Enfin, du point de vue du développement, il est plus facile de parcourir l'ensemble des relations de causalité d'un commutateur que l'ensemble des acteurs d'un système.

5.4 Distribuer les acteurs sur plusieurs machines virtuelles Java

(to be expended)

C'est possible avec cette architecture, à condition de comprendre comment accéder à une machine virtuelle depuis une autre. Une fois ce détail technique résolé [7], l'objet

6. Voire pas la moindre idée tout court.
7. Coquille. Contraction de *résolu* et *réglé*.

`SystemProxy` devra ne plus permettre d'accéder aux acteurs par leur nom [8] mais par leur chemin.

5.5 Prévenir les situations aberrantes

Définir une situation aberrante. C'est plus compliqué qu'il n'y parait : aberrant à partir de quel seuil, selon quel point de vue ?

Une telle situation est causée par quatre types d'interférences.

Interne au commutateur On peut les prévenir super facilement, ça se résume à une recherche de cycle sur le graphe (A, C) des acteurs et des relations de causalité. La mise en œuvre est en revanche hardue : car en réalité ce n'est pas un graphe (mais ça se dessine pareil donc la structure est pareille) et il ne faut pas oublier que cet chose pseudo-graphe joue avec des pointeurs de fonction, des pointeurs de classe et des acteurs. Je suis très currieux de voir l'algorithme de recherche de cycle sur un hypergraphe qui correspond au cas général des relations de causalité de rang quelconque.

Entre commutateurs x acteurs touchés par des relations de causalité qui appartiennent à y commutateurs différents, x et y quelconques.

Externe au commutateur Par exemple l'influence thermodynamique mutuelle de x acteurs radiateurs dont les relations de causalité respectives sont émulées par $y, y > x$ commutateurs sans qu'un commutateur émule deux recettes du même radiateur.

Le piège de l'utilisateur malicieux **Gentille** périphrase. La supputation d'intention est probablement compliquée, mais on peut imaginer une forte incitation dans l'ergonomie de l'interface à le cantonner à des trucs basiques. On peut également éviter d'ajouter des éléments trop exotiques à l'objet `MessageMap` : même si le faisceau de relations de causalité est indemne de tout problème des types précédents, l'utilisateur risque de considérer comme un aberrant que sa porte de garage s'ouvre lorsqu'il se brosse les dents ou que l'alarme de sa maison se réveille quand il se réveille.

5.6 What « Model (in MVC) is an abstraction of Actor » means and how we could implement it

Model is an abstraction of Actor. Because it takes a class reference, one could have subtypes of this class. By the way, the best abstraction would be to link a model to an interface which some actors would implements. It would allow something like

8. Qui n'est que la dernière partie de leur chemin et dont l'unicité n'est garantie qu'au sein d'une seule machine virtuelle.

multiple inheritance. As an actor would implements several interfaces, it could be sent different messages. The main issue with this idea is an actor only receive message by onReceive() and the message sending protocol doesn't imply any other method.

5.7 Lien entre une recette et une relation de causalité

Nous pouvons relier ces deux notions.

Euh en fait ça se fait à l'instinct dans le code puisque le niveau 1 n'est utilisé que par l'interface et que pour l'instant il n'y a pas d'interface. Les exemples du code n'utilisent que des événements de niveau 2. Mais c'est une vraie question qu'il faut traiter. D'ailleurs, petit détail tant qu'on y est : les actions et les signaux d'un canal sont complètement inutiles. Il prévu de ne s'en servir lorsqu'on crée ni une recette ni une relation de causalité, cela par conception. Ils ne servent qu'à être affichés dans l'interface.

5.8 Quelques utilisations de l'interface

Pour aider à finir l'interface, voici quelques exemples de ce qu'on doit pouvoir y faire.

Pour un utilisateur donné

Enregistrer un objet connecté Equivaut à créé un acteur et à le sauvegarder en base (`Model.Actor`).

Oublier un objet enregistré Inutile pour nous, la base est purgée pour chaque nouvelle simulation.

Créer une relation de causalité Cette étape est abusivement appelée création d'une recette par ifttt, mais les recettes sont déjà définies (ou crées) et on ne fait que choisir celle qu'on veut appliquer comme relation de causalité pour le quadruplet (acteur émetteur, type de message, canal récepteur, type d'action). Il est a noter que la fonction de création d'une relation de causalité demande qu'on lui passe ne paramètre la fonction qui permet de créer le message d'action en fonction du message de signal. Si cette fonction n'est pas définie alors on va demander une fonction par défaut à l'objet `MessageMap`. Si une telle fonction par défaut n'a pas encore été explicitement définie, `MessageMap` créera une fonction constante à la volée. Il est donc bon de noter que définir une recette programmatiquement est bien plus fin et offre bien plus de liberté.

Voir le log A mieux définir

Pour l'administrateur de la machine virtuelle Java locale

Définition d'une recette Attention on reste dans le côté abstrait, ne pas confondre avec créer une relation de causalité. Il ne s'agit ici que de créer un quadruplet (canal émetteur, signal, canal récepteur, action).

Simuler l'envoi aléatoire d'un message Si la période est constante alors l'envoi est périodique. Pour n'envoyer qu'un seul message, il suffit d'utiliser (`StopCriteria.Occurence`, 1).

Voir le log A mieux définir

6 Utilisation

Parler des classes et des manières « pratiques » de les utiliser avec des exemples de code and so on...

Bah euh la javadoc et les exemples du code peuvent suffire (temporairement) à ce point.

7 Dernière soutenance

Ce qui pourrait faire briller les yeux de Mme Vigne :

— Qualimétrie (pas encore fait)
— Java 8 : flux, lambda-calcul
— Interface fonctionnelle : on doit bien pouvoir en placer une en réfléchissant bien.
— Réification des génériques
— Patrons de conception
— Tentative de formalisation rigoureuse
— ...