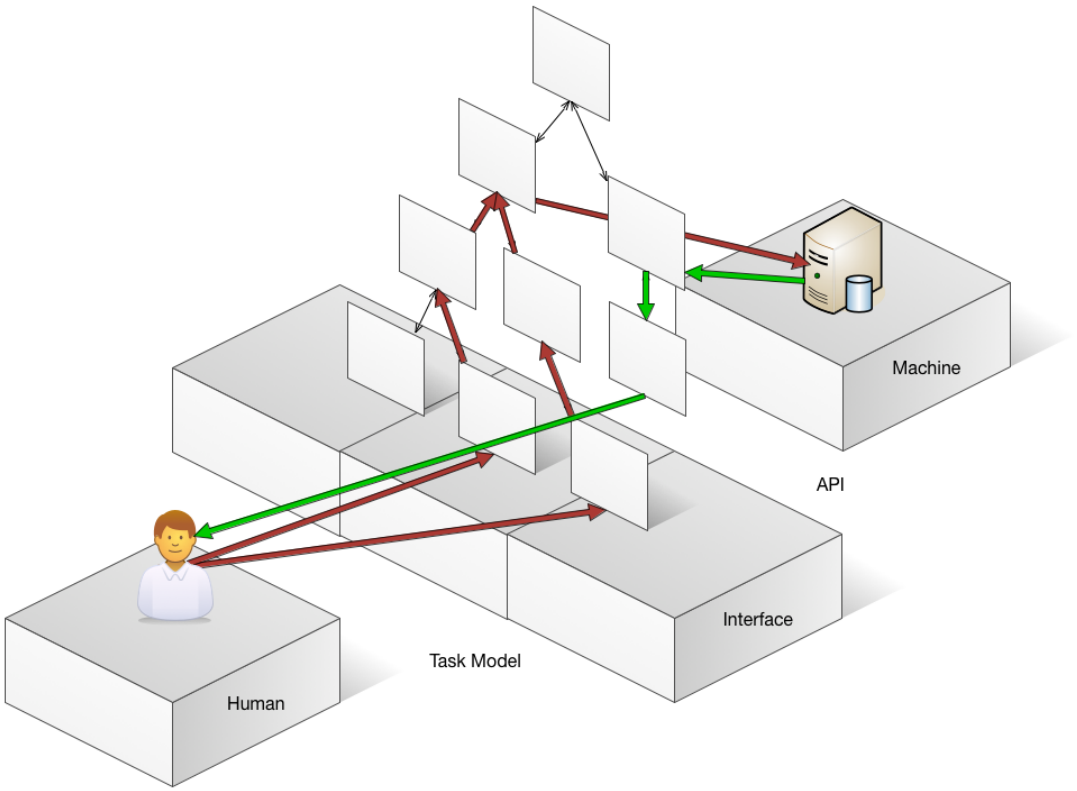
Langage L : référence

Un interacteur est une entité qui dispose des caractéristiques suivantes, qui seront vues en détail dans les pages suivantes de ce document :

- Un état interne défini par des variables d'état
- Une association avec d'autres interacteurs par le biais d'une relation de composition
- Des moyens de communication avec les autres interacteurs
- Un ensemble de comportements causés soit par l'état interne de l'interacteur, soit par des signaux venus d'autres interacteurs, et agissant soit sur l'état interne de l'interacteur, soit sur des signaux envoyés vers les autres interacteurs



Etat interne d'un interacteur

- A une date donnée, l'état interne courant d'un interacteur est un, et un seul, des éléments de l'ensemble des états possibles pour cet interacteur
- L'ensemble des états possibles d'un interacteur est le produit cartésien des ensembles de définition des variables d'état de cet interacteur
    - Dans le cas ou certaines variables d'état sont de type composé, on rappelle que l'ensemble de définition d'un type de donnée composé est évidemment le produit cartésien des ensembles de définition de ses champs
    - Dans le cas ou le cardinal de l'ensemble de définition d'une variable élémentaire (c'est à dire présente une fois les types composé « dépliés ») est infini, (par exemple dans le cas d'une variable entière non bornée, ou d'une variable réelle), on le réduira lors de la génération de code pour le model-checking, en utilisant des techniques d'interprétation abstraite sur les comportements mettant en scène cette variable. (Est ce bien possible dans tous les cas? J'imagine que ce n'est pas forcément facile...)
- Les types de base pour les variables d'état d'un interacteur sont au nombre de 6:
    - Nombres entiers avec le mot clef **integer**
    - Nombres réels avec le mot clef **real**
    - Booléens avec le mot clef **boolean**, utiles pour les expressions logiques
    - Chaines de caractère avec le mot clef **text**
    - Date absolue avec le mot clef **time**, utile pour la spécification de propriétés temporelles
    - Référence vers un interacteur avec le mot clef **interactor**, ce qui permet un certain niveau d'introspection des interacteurs (Pas sûr que cela ne soit ni utile ni raisonnable...)
- La déclaration d'une variable d'un type de base peut s'accompagner de la spécification d'une restriction de son domaine de validité à l'aide du mot clef **in**
    - L'ensemble dans lequel les variables sont restreintes est spécifié à l'aide de notations ensemblistes de base :
        - Ensembles discrets à l'aide d'accolades : {1, 2, 3, 4}
        - Pour les réels et les entiers, intervalles à l'aide de crochets : [1, 4]
    - Un exemple utile de restriction : l'équivalent des « enum » du langage C sont les variables de type **text**, avec un ensemble de définition restreint aux différentes valeurs de l'énumération
    - Autre exemple : un angle peut être défini comme un réel restreint sur [0, 2*pi[
- Les types composés existent, et sont définis avec le mot clef **structure**, séparément des interacteurs afin d'être réutilisables par plusieurs « classes » d'interacteurs
- Les tables existent, et sont définies en utilisant la notation classique du langage C : **integer**[100] représentera une table de 100 entiers. (Operateurs de base sur les tables ?)

Composition des interacteurs

- Les interacteurs sont composés de manière hiérarchique. Cette hiérarchie forme un arbre d'interacteurs qui représente la structure de l'IHM
- Les interacteurs feuilles de l'arbre sont
    - Soit d'un type de base
    - Soit d'un type abstrait dont seule l'interface avec les autres interacteurs est connue. Ce type abstrait sera implémenté manuellement par les développeurs, et sera exclu de toute interprétation du code, en particulier du model checking, de la génération de code… (Ceci risque de poser tout un tas de problème, dans un premier temps, cette possibilité sera donc exclue)
- Les types d'interacteurs de base reflètent les types de base des variables d'état, en ajoutant deux notions : la direction de l'interaction, et son caractère discret ou continu. Les types de bases considérés sont les **integer**, **real**, **boolean**, **text**. On aura donc en théorie 4 * 2 * 2 = 16 types de bases réguliers + 2 types de bases supplémentaires. En voici quelques exemples :
    - **integerInputDiscrete**, représente un interacteur permettant à l'utilisateur de spécifier un entier, de manière discrète dans le temps. Exemple : Une boite de dialogue qui apparaît pour demander à l'utilisateur d'entrer un entier.
    - **realOutputContinuous**, représente un interacteur affichant un réel, de manière continuelle dans le temps. Exemple : Une jauge de carburant
    - **booleanInputContinuous**, représente un interacteur permettant à l'utilisateur de spécifier un booléen, susceptible de varier en permanence. Exemple : Un interrupteur
    - **textOutputDiscrete**, représente un interacteur affichant du texte à des instants donnés. Dans une interface graphique, cela a peu de chances d'exister, mais les outils de synthèse vocale entrent dans cette catégorie d'interacteurs
    - **triggerInput**, **triggerOutput** sont deux types supplémentaires un peu spéciaux. Il s'agit d'interacteurs qui transmettent des signaux, mais sans données. Ces interacteurs sont inspirés des messages « bang » en Max/MSP ou PureData. Bien qu'ils ne transmettent aucune donnée, leur intérêt est qu'ils transmettent des évènements discrets, et donc une date. Evidemment, le suffixe discret/continu est superflu, ces interacteurs étant discrets par nature.

Communication entre interacteurs

- Il est impossible d'atteindre des interacteurs autres que les voisins directs dans l'arbre. Un interacteur n'a donc accès qu'à son unique parent et ses enfants directs. Ceci est une vraie restriction par rapport aux autres méthodes de développement d'IHM, mais il y a une vraie justification à cela : améliorer la réutilisabilité des artefacts de développement, et clarifier le code en évitant le « code spaghetti ». Si une donnée a besoin d'être présente à deux nœuds de l'arbre séparés par plus de deux arcs, c'est qu'elle a forcément :
    - Soit un sens au niveau du cœur applicatif, c'est à dire un vrai sens fonctionnel. Les deux interacteurs communiqueront donc indirectement en agissant sur cette donnée dans le cœur applicatif.
    - Soit un sens purement IHM (mais pas de sens au niveau fonctionnel) pour l'ancêtre commun de ces deux nœuds, c'est à dire un sens global, une vraie raison d'être dans le sous arbre descendant de cet ancêtre commun. Cette donnée a donc forcément du sens pour chaque nœud sur le trajet (unique puisque l'on est dans un arbre) entre les deux nœuds qui l'ont en commun. Cette donnée transitera donc par ce trajet unique.
- On peut désigner un élément de l'entourage dans l'arbre de plusieurs manières :
    - De manière générique avec des mots clefs
        - **parent** : l'interacteur parent, dont l'interacteur courant est un composant
        - **child** : n'importe lequel des interacteurs enfants
        - **self** : l'interacteur courant
        - **other** : un interacteur voisin, c'est à dire le parent ou un enfant
        - **any** : n'importe lequel des interacteurs accessibles à l'interacteur courant, aggrégation de **self** et **other**
    - En le nommant simplement si c'est un enfant en particulier
- Techniquement, la communication entre deux interacteurs se fait à l'aide d'un emplacement mémoire, que l'on appellera un slot, partagé par ces interacteurs. Seul l'interacteur propriétaire du slot peut écrire dans le slot. Le contenu de ce slot sera appelé signal.
- En plus de stocker une donnée en permanence, un slot peut aussi servir à envoyer une notification qui déclenchera un évènement.

Comportements des interacteurs

La syntaxe de spécification des comportements s'appuie sur la construction de phrase souvent utilisée pour spécifier empiriquement les comportements d'IHM :
« Si telles conditions sont vérifiées à tel instant, alors je veux que ceci se produise »
En langage L, cela donne :

```
<déclencheur> if <garde> : <effet>
```

Types de déclencheurs :

- **on** : Déclenchement à chaque réception d'un signal. Le signal peut être n'importe lequel des signaux envoyés par des interacteurs accessibles depuis l'interacteur courant, y compris lui-même.
- **when** : Déclenchement sur chaque front montant de la valeur de vérité d'une condition. La condition peut être basée sur la valeur des signaux envoyés par n'importe lequel des interacteurs accessibles depuis l'interacteur courant, y compris lui-même, ainsi que par les valeurs des variables d'état de l'interacteur courant.

Types d'effets :

- **always** : déclaration d'un invariant qui doit rester vrai tant qu'il n'est pas explicitement renié, à rapprocher de l'opérateur Globally en LTL
- **set** : affectation simple, à rapprocher de l'opérateur Next en LTL
- **send** : identique à set, mais en ajoutant la notion d'envoi d'un signal, de déclenchement d'évenement, la valeur affectée sert de paramètres de l'évenement

Signaux speciaux :

- **init** : a l'initialisation, juste avant de démarrer l'execution, quand tout est chargé
- **exit** : a la toute fin de l'execution, avant de commencer a tout desallouer

Remarque :

- Les effets **always** et **set** sont identiques si le terme de droite de l'affectation est une constante, la différence n'existe que si le terme de droite est variable. Dans ce cas, **always** stipule que le terme de gauche doit varier comme celui de droite, alors que **set** stipule que le terme de gauche doit garder la valeur qu'il a prise au moment de l'affectation.
- Les effets **always** et **set** auront tendance à déclencher des comportements en **when** chez les recepteurs, alors que les effet **send** auront tendance à déclencher des comportements en **on**
- Quand il y a un « conflit » sur les comportements, c'est à dire lorsque deux comportements affectent des valeurs différentes à une variable, alors le dernier comportement à être déclenché l'emporte.

Exemples :

Cause	Effect	Example
When	Always	`when child.value > 0 if condition :` `always parent.valid = true`
When	Send	`when child.value > 0 if condition :` `send parent.valid = child2.value`
When	Set	`when child.value > 0 if condition :` `send parent.valid = true`
On	Always	`on init if condition :` `always child1.text = « a »`
On	Send	`on child.event if condition :` `send parent.event`
On	Set	`on child.event if condition :` `set parent.event = 5`

Exemples de pratiques en IHM :

- UI Data binding : on associe les deux parties prenantes du binding à l'aide de deux comportements, un pour chaque direction
- Observer pattern : Simple association à l'aide de comportements en always
- Pattern MVC : La view est le CDS, le model est le système, le controlleur est l'application en langage L
- Déclenchement d'évenement avec paramètres. Techniquement, on place les valeurs des paramètres dans le slot, puis on notifie le receveur. Ceci se fait en une ligne.

Model checking

1. Specification en langage L

Generation de code

2. Code Promela

Spin model checker

3. Verifications

1. Specification en langage L

Generation de code par Vincent

2. Prototype en Java

Promela translator de Bandera

3. Code Promela

Spin model checker

4. Verifications