**Compilation HipHop**

Pour comprendre quelque chose à la compilation HH il faut d’abord lire « Esterel Constructive Book ». C’est le document de référence de la sémantique d’Estérel utilisée par HH. Il y a plusieurs sémantiques pour décrire Estérel :

Logical Behavorial, State Behavorial, Constructive Behavorial, Constructive Operational, Constructive circuit.

Le chapitre III de « Compiling Estérel » est très proche du « Esterel Constructive Book ».

**hiphop.js** est un point d’entrée qui ne fait rien de compliqué.

On crée à partir de **machine.js** (ReactiveMachine) l’AST (ast.js, lang.js) en utilisant le fichier source (dans *setSignals* d’orchestrationHH.js avec le code ).

1. **orchestrationHH.js** -> setSignals -> machine = new hh.*ReactiveMachine*( orchestration, {sweep:true}); La référence à **hh** n’est qu’un lien vers l’export *ReactiveMachine* de **machine.js**.

*ReactiveMachine* est un objet (class). Le constructeur de *ReactiveMachine* a comme paramètre une variable *ast\_node* qui est le programme généré par Blockly. Ce programme doit être de type Module. Cet objet comprend toute une série de paramètre dont des map de signaux in et out, et une copie de l’AST. La compilation est lancée par le constructeur en ligne 123.

Ceci crée la machine qui va gérer l’exécution avec **machine.js** en compilant avec **compile.js** qui s’appuie sur **net.js** pour la création des portes. L’exécution consiste à appeler la **machine** avec **react()** (Il peut y avoir une recompilation si l’AST a été modifié). La machine possède un clone de l’AST. Elle balaie les nets pour faire avancer la machine. Les nets constituent la machine.

**Machine.js** gère essentiellement le fonctionnement de la machine. **Machine.js** fait appel à **compile.js** pour créer la machine. **Machine.js** fait appel à **net.js** pour *react()*. Net.js sert donc à la fois pour la création des circuits et pour leur fonctionnement.

Node => nœud de l’AST, va stocker le circuit et se connecter aux autres circuits après compilation. Comprend des nœuds action, expression, count expression et les éléments du langage, emit, sustain…

Le compilateur construit les circuits à partir de l’AST et les connecte, ça donne une machine.

Net => circuits composant une machine, il y des fan out et in?

Circuit => mise en œuvre en circuits logiques de HH.

Accessor ?

# Net.js (500 lignes)

Construction et propagation des nets HH. Ici on traite des portes et pas des circuits. Du coup on ne sait plus trop où sont les circuits.

Il y a des nets logiques, des nets registres et des nets action.

Class Net ----> LogicalNet

------> RegisterNet

------> ActionNet ----> TestExpressionNet

------> SignalExpressionNet

Hormis les classes il n’y a que deux fonctions : makeOr et makeAnd.

FAN IN et OUT => entrée sortie d’un net, composé de : net, polarity, dependency et antagonist.

Une classe Net dans un nœud AST. On push un net dans un nœud (node) AST.

connectTo(netExterieur, type) : connecte le net de la classe à un net extérieur. Les fanout.antagonist sont les fanin de la classe. Les fanin.antagonist sont les fanouts du net extérieur.

Un LogicalNet neutre est un AND si true, un OR si false.

Le *receive* de net.js ne correspond pas exactement à la thèse de Colin. On applique *receive* à chaque entrée d’une porte. Voici celui qui tourne :

**Receive(value, fromDep)**

***Value****: true ou false*

***fromDep****: flag qui indique si le receive vient d’une dépendance de donnée*

***Retourne*** *faux si la porte n’est pas connue (pas évaluée) vrai si elle est évaluée.*

(Je ne sais pas exactement ce qu’est **trueFaninCount**, est-ce seulement un compteur des broches ? fan = connexion entre circuits logique. Fanout = connexion vers plusieurs circuits logiques. Il y a une limite en électronique. Dans notre cas c’est comme une « broche ». Pourquoi trueFaninCount et pas seulement faninCount ?).

Si déjà connu, on ne fait rien et on retourne false. L’appelant n’aura rien à faire.

Si **fromDep**, on résoud une dépendance la valeur **value** n’a pas de sens. On décroit le compteur de dépendances de net. **fromDep** signifie donc que l’on est dans le cas d’une connexion avec un autre « net » soit de contrôle, soit de données. (p.144 de la thèse de Colin).

Si le compteur de dépendance est à 0.

Si la valeur du net est définie on retourne true. C’est fini, tout est OK.

Sinon on test si le compteur trueFaninCount du net est à 0 (on ne dépend plus de rien donc )

Si oui on met value du net à la valeur de l’élément neutre du net et on retourne true (il faut bien avoir une valeur. Il n’y a pas de dépendance et pas de fanin. Ici on prend le neutre).

Sinon **fromDep** on **n’est pas** en train de traiter une valeur selon une dépendance. On traite donc un fanin\*\*

On décrémente le compteur trueFaninCount du net (donc on traite un fanin ?).

Si value reçue est différent de l’élément neutre (on agit sur la porte)

On met la value du net à value (la valeur du net peut être à undefined). On fait donc une opération logique avec value et this.value. En effet, l’élément neutre par défaut ne faisant rien, il n’y a que dans le cas où il est différent qu’il y a une évaluation ? Pas certain car net.value peut être undefined.

On retourne true, c’est OK.

Sinon c’est qu’on a reçu une **valeur neutre**. On ne fait rien sauf si trueFaninCount est à 0. trueFaninCount vaut 0.

*Pourquoi force-t-on la value de net dans ce cas ? En debuggant je note que dans ce cas net.value est* ***undefined****. Donc si on reçoit une valeur neutre et que trueFaninCount est à 0 on a net.value undefined que l’on met à neutre. Ce n’est pas exactement formulé comme Colin. Ici ça veut dire que l’on ne s’occupe pas de value du tout dans ce cas. Mais dans le cas d’une dépendance c’est OK. Donc ça pourrait signifier que quand on est sur une* ***dernière******broche******sans******dépendance*** *et que* ***value et neutre*** *on met* ***net.value*** *à neutre. (à vérifier).*

(This.value devient neutre s’il n’y a plus (ou pas) de faninTrue, mais la fonction renvoie true seulement s’il n’y a plus de dépendance.)

Si le compteur de trueFaninCount est à 0 (on est sur la dernière broche)

Value du net est mis à neutre

On retournera le fait ou non que le compteur de dépendance est à 0

Sinon on retourne false

**Questions** :

C’est quoi un « true fanin » par rapport à un fanin ? (sans doute la même chose)

C’est quoi les « control gates » ?

Dans net.js, que signifient STD, NEG, DEP (dependency) dans les FAN ? p.115 de la thèse de Colin.

C’est quoi la « polarity » d’un fanout ?

« Accessor » a quelque chose à voir avec les signaux ?

« La fonction getAccessorList analyse le contenu textuel de func et détermine quels sont les signaux lus en recherchant les expressions générées par la transformation des accesseurs. Elle est définie dans src/lang.js, et elle est appelée par le constructeur de ActionNode. Les informations fournies par accessorList sont nécessaires à la génération du circuit. Le mécanisme est détaillé à la section 6.4.3. » p.107

# Reaction

Dans la fonction react() on construit une liste known\_list de nets que l’on va utiliser pour propager la réaction avec propagate() qui est une méthode de l’objet Net. On retrouve grosso modo l’explication de Colin en p.112.

# Compilation

La compilation se fait une première fois à la création de l’objet **ReactiveMachine**. Mais la question de recompiler se pose à chaque réaction ou input de signaux.

Compiler.js : c’est là que se joue le document de Gérard « Esterel Constructive Book ». On y retrouve les circuits sous forme logicielle.

Suspend est un exemple assez lisible de mise en œuvre des circuits logique. (ligne 975) pour le schéma p.119 d’« Esterel Constructive Book ».

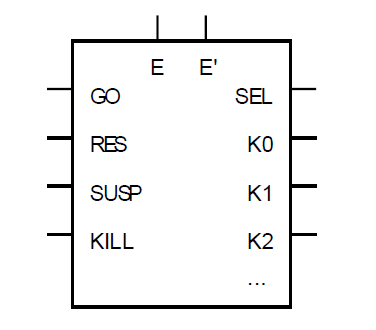
# Explication des circuits

Voir « Esterel Constructive Book » **chapitre 10**. C’est assez clair et complet au début. Attention au terme *sémantique constructive*, c’est spécifique à Estérel. On part vite dans un formalisme mathématique pas très bien expliqué pour un profane. Notamment les notions de fonction *R(w), I(w) et C(w).*

I input, R état, e expression, b résultat de l’évaluation de l’expression.



**Chapitre 11** est nécessaire pour comprendre quelque chose à net.js. On y donne les définitions des circuits.



- La broche d'entrée **GO** est utilisée pour **relancer l'instruction**, c'est-à-dire pour exécuter une règle s dans la sémantique d'état. Une s-rule dans la sémantique d'état. Cela se produit lorsque GO est activé.

- La broche d'entrée **RES** est utilisée **pour reprendre l'exécution** d'une instruction sélectionnée, c'est-à-dire pour exécuter une règle r dans la sémantique d'état. Cela se produit lorsque RES est activé.

- La broche d'entrée **SUSP** est utilisée pour **suspendre l'exécution** de l'instruction, conformément à la règle de la sémantique d'état (r-suspend+). La suspension se produit lorsque **SUSP** est activé. Ensuite, les registres conservent leur valeur actuelle, sauf s'ils sont tués par l'entrée **KILL** ci-dessous.

- La broche d'entrée **KILL** est utilisée pour **annuler les registres de l'instruction** en cas de sortie de trappe. Cela se produit lorsque KILL est activé par la traduction d'une instruction trap {p}, comme spécifié par la règle (sr-term-trap). Le signal KILL est propagé par toutes les instructions vers les registres de pause.

- La broche de sortie de sélection **SEL** indique **que l'état est actuellement sélectionné** pour la reprise, c'est-à-dire qu'un ***registre interne de pause*** est activé. Le signal SEL est simplement la disjonction des registres internes.

- Les broches de sortie **K0, K1**, etc. correspondent aux codes **d'achèvement (completion code)**. *Il y a n + 2 broches de ce type si n est le nombre de trappes dans lesquelles l'instruction est enfermée*. Lorsque l'instruction est soit lancée, soit sélectionnée et reprise, la broche qui correspond au code d'achèvement renvoyé par l'instruction est activée. Si l'instruction n'est pas exécutée, c'est-à-dire si elle n'est pas lancée, pas sélectionnée ou pas reprise si elle est sélectionnée, alors le circuit de l'instruction désactive explicitement toutes les broches K. Notez que les codes d'achèvement sont codés de manière unaire (one-hot dans la terminologie matérielle).

- Les broches **E et E’** correspondent à **l'interface de signal**. Il ne s'agit pas de simples broches mais des *broches composés ou bus*, contenant une broche élémentaire par signal visible dans la portée de l'instruction courante. Le bus d'entrée E est destiné à l'entrée de l'instruction, correspondant à l'événement d'entrée E dans la sémantique. La broche du bus de sortie E’ est destinée à la sortie de l'instruction, correspondant à l'événement de sortie E’. Nous allons librement extraire des signaux spécifiques s ou s’ de E ou E'. Comme pour les broches K, les broches E’ sont explicitement désactivées lorsque l'instruction n'est pas exécutée, c'est-à-dire lorsque ¬( GO ∨ ( RES ∧ SEL)).

# Le schéma d'exécution (Esterel Constructive Book p.112)

Le schéma d'exécution de base consiste à activer d'abord **GO** pour démarrer l'exécution, puis **RES** à chaque cycle d'horloge. A chaque cycle, le contrôle se propage de manière combinée dans le circuit d'instruction ; la broche d'achèvement correspondant au code d'achèvement retourné est activée, et les registres correspondant aux **instructions de pause** atteintes sont définis pour reprendre l'exécution à partir du bon état dans le cycle suivant. Les signaux visibles d'Esterel sont reçus et émis à travers E et E’.

Pour suspendre l'instruction pendant le cycle, nous activons **SUSP** au lieu de **RES** (RES et SUSP ne sont jamais activés en même temps). Si l'instruction est préemptée par une sortie de trappe interne ou concurrente, nous activons **KILL** pour annuler la pause registres.

Le fil de sélection **SEL** est propagé vers le haut dans les instructions composées. Il reste activé tant que certains *registres de pause* sont activés, c'est-à-dire tant que l'instruction est sélectionnée. Il est désactivé au cycle qui suit la terminaison ou la sortie de trappe, à moins que l'instruction n'ait été immédiatement redémarrée par une boucle. Le fil **SEL** est nécessaire car **RES** peut aussi être envoyé à des instructions non sélectionnées. Lorsque **RES** est activé, les instructions non sélectionnées doivent rester silencieuses, c'est-à-dire qu'elles doivent annuler tous les signaux de sortie et les fils d'achèvement. Cela se fait en connectant **RES** et **SEL** en interne.

Puisqu'il est possible de revenir instantanément en boucle sur une instruction, plusieurs comportements élémentaires d'un sous-circuit peuvent être réalisés au même instant. Nous négligerons cet aspect pour le moment et l'analyserons en détail au chapitre 12.

Il faut voir la partie II pour comprendre le chapitre 10.

**Question** :

**C’est quoi une porte pour Colin ?** C’est une **porte logique** (ET, OU, …). L’évaluation d’une expression de type *signal.val* se fait avec une porte ( On parle alors de porte ou de circuit?). Colin utilise surtout « porte » pour les portes logiques. (voir 6.4 Modèle d’exécution).

**C’est quoi un registre pour Colin ?** C’est un **registre dans un « net » (circuit)**. Un circuit est composé de portes logique (gates) et de registres (cf. Esterel Constructive Book p.111).

Une instruction HH génère donc un circuit composé de portes (gates) et de registres. La machine est une combinaison de circuits interconnectés par les broches GO, RES, SUSP, KILL, SEL, E, E’, K0, K1… Le document Esterel Constructive Book explique la constitution des circuits et leurs combinaisons.

**C’est quoi un net pour Colin ?** On pourrait croire que c’est un **circuit**. Avec des E/S GO, RES, SUSP, KILL, SEL, E, E’, K0, K1… Mais dans net.js la classe Net est une porte logique.

**Nœud d’AST** ? A un nœud de l’AST correspond un circuit. On a des circuits Action (ActionNode), des circuits Registres (RegisterNode) et des circuits Logiques (LogicalNode). Un nœud de l’AST n’est pas un circuit mais une classe permettant de produire un circuit. Un nœud contient une accessorList qui est la liste des signaux à lire, une liste de signaux émis par l’instruction (signalNameList). C’est à partir de l’AST que l’on construit la machine. Ce que fait Blockly n’est pas produire un AST mais un code JS qui permet la production de l’AST en appelant des fonctions « hh. ».

Code JS produit par **Blockly** produit l’AST par **appel aux fonctions hh** (*hiphop.js.js -> lang.js*). *lang.js*  appelle les fonctions de *ast.js*. -> AST.

La machine est produite par **ReactiveMachine** de *machine.js.* qui fait des vérifications et des manipulations puis appelle compile de *compiler.js* (6.3 de la thèse de Colin) en appliquant le modèle d’exécution selon la sémantique constructive d’Estérel..

C’est *compile.js* qui traduit donc l’AST en circuits en faisant appel à *net.js*. On obtient ainsi un grand circuit logique composé de portes et de registres correspondant à la machine. La thèse de Colin est laconique sur la production des circuits selon la sémantique constructive.

Notons que la machine permet d’associer les portes à des circuits pour déboguer (p.111). On se préoccupe plus des circuits (nets) dans l’exécution mais uniquement des portes.

# Logical Behavorial Semantics

Le chapitre 6 d’« Estérel Constructive book » est assez clair. J’ai lu jusqu’en 6.4. C’est la définition d’Estérel selon un formalisme mathématique particulier. Il y a en fait assez peu à comprendre, ce sont des définitions.

Des définitions. Elles me semblent plus claires chez G. Berry que dans le livre Compiling Estérel :

Un **sort** S (tri, classe) est un **ensemble de signaux**. (Pourquoi appeler ça comme ça ?)

Un **événement** E est un **ensemble d’états** pour chaque signal b = {+, -}

Un **sort** (tri) d’un **évènement** (pour un événement), c’est-à-dire un ensemble de signaux pour un événement est noté **S(E)**. Est-ce que ça veut dire que S(E) est l’ensemble des signaux utilisés par l’événement, en entrée, en sortie ?

Le **statut** d’un **événement** s pour un état E est E(s) inclus dans **{anti-truc, 0, 1}**

**S\s = S – {s}**, donc le sort S sans le signal s ( ?)

**E\s** correspond à un événement de S(E)-{s}=S(E)\s qui coïncide avec E sur tous les signaux sauf s. E\s correspond donc à un **ensemble d’états E qui ne prend pas s en compte**. On dirait une façon de prendre en compte E sans s’occuper de s. (Rappel, le \ correspond à l’exclusion d’un ensemble).

Soit un événement E avec un sort S, un signal s potentiellement pas dans S, un état b dans B.

**E\*sb** = **E’** (un ensemble d’états)de sort **S U {s}** (au cas où s ne serait pas dans S), avec **E’(s) = b** et **E’(s’) = E(s’)** si **s’ != s**. Il s’agit donc d’un ensemble d’états ou le signal s a la valeur **b** sans savoir ce qui se passe pour les autres signaux.

Notons que l’état de **s** dans **E** est perdu dans un nouvel état E’ = **E\*sb** si s est inclus dans S(E) c.a.d. fait partie des signaux de l’événement E. Autrement dit, si s fait partie des signaux ayant une certaine valeur b dans E ( = S(E) ). En clair :

E est un état avec s inclus avec une valeur **X**. E’ est un nouvel état mais où s vaut **b**. On ne sait donc rien de **X** qui est n’est pas visible. Si s n’était pas dans E son état ne peut-être que b, X n’existe pas. Ceci justifie les signaux locaux.

## Can et must (Constructive Behavorial Semantics 4.2)

Un programme est constructif si la propagation peut-être exprimée avec les prédicats must et can (ou cannot) pour établir la présence ou l’absence d’un signal (p38 Estérel Constructive Book).

L’opposé de « doit » est « ne peut pas ». On parle de cohérence logique en appliquant les règles **must** et **cannot**. Si on ne peut pas appliquer **must** il faut pouvoir appliquer **cannot** sinon on rejette. On dirait donc qu’un programme est cohérent si il se déroule en appliquant **must** ou **cannot** sur les signaux.

On ne peut pas appliquer un **must** sur un indétermination, mais seulement un **cannot**.