PASSERELLE UML/SIGNAL

RAPPORT N° CS/311-1/AJ000212/RAP/02/06/chrono Version 0.1

Préparé par

CS SI
Division Opérationnelle Ingénierie Scientifique
Centre Opérationnel France Nord
16 avenue Galilée
92350 LE PLESSIS ROBINSON

Rédigé pour

	NOM(S)	DATE(S)	VISA(S)
REDACTEUR(S)	C. MODIGUY	27/06/02	
VERIFICATEUR(S)			
APPROBATEUR			

FICHE DE SUIVI DES MODIFICATIONS

Version/Révision		Références		Description des modifications	Auteur(s)
Indice	Date	Page	N° §		

LISTE DE DIFFUSION

DIFFUSION EXTERNE

DIFFUSION INTERNE

Archivage DO **SCSA**

DOCUMENTS APPLICABLES ET DE REFERENCE

DOCUMENTS APPLICABLES

DOCUMENTS DE REFERENCE

SOMMAIRE

1. INTRODUCTION

Ce document vise à présenter les spécifications d'une maquette de passerelle UML vers Signal.

2. SPECIFICATIONS DE LA PASSERELLE

2.1 MODELE UML SOURCE

La génération s'appuiera sur le paquetage Core du DAM.

Restrictions par rapport à la méthode UML/ACCORD :

- Un seul niveau hiérarchique pour les états des classes actives.
- Les signaux peuvent posséder 0 ou 1 attribut
- Les types de données utilisés sont des types qui existent en langage signal
- Un seul diagramme d'états-transitions pour la machine à états d'une classe active.
- Dans un digramme d'états-transitions, les sent event de deux transitions diférentes ne peuvent instancier le même signal.
- Dans un digramme d'états-transitions, si une transition admet un sent event, alors une opération avec paramètre de retour est associée à la transition.
- Si au moins une des méthodes, d'une des classes actives est déclenchée avec une période temporelle, il doit y avoir un signal d'entrée sans attribut nommé "Top".

2.2 REGLES DE GENERATION DU CODE SIGNAL CIBLE

2.2.1 Organisation du programme Signal cible

Un process principal contenu dans un fichier nommé "env_**MySytsemCore**.SIG".

Un module pour chaque classe contenu dans un fichier nommé "**MyClass**.LIB".

2.2.2 Génération du process principal

Un process principal est nommé env_**MySytsemCore**, où **MySytsemCore** est le nom du paquetage "Core".

2.2.2.1 Process "env"

2.2.2.1.1 Déclaration de l'interface

Entrée :

Pour chaque signal **myInputSignal** du paquetage InputSignals, on déclare :

- si **myInputSignal** n'a pas d'attribut, un signal nommé B**myInputSignal** de type *boolean*,

- si **myInputSignal** a un attribut, un premier signal, nommé B**monSignal**, de type *boolean*, et un second signal, nommé **V myInputSignal**, de même type que l'attribut.

Sortie :

Pour chaque signal **myOutputSignal** du paquetage OutputSignals, on déclare :

- si **myOutputSignal** n'a pas d'attribut, un signal, nommé B**myOutputSignal**, de type *boolean*.

- si **myOutputSignal** a un attribut, un premier signal, nommé B**myOutputSignal**, de type *boolean*, et un second signal, nommé V**myOutputSignal**, de même type que l'attribut.

2.2.2.1.2 Corps

- Une équation d'appel du process main défini plus loin, qui définit les signaux de sortie non booléens du process "env".

Supposons que le paquetage "InputSignals", contienne les signaux **myInputSignal1**, sans attribut, et **myInputSignal2**, avec un attribut, et que le paquetage "OutputSignals", contienne les signaux **myOutputSignal1**, sans attribut, et **myOutputSignal2**, avec un attribut, l'appel du process "main", s'écrira :

*(E**myOutputSignal1**, V**myOutputSignal2**) := main_**mySystemCore**(Tick, when B**myInputSignal1**, V**myInputSignal2** when B**myInputSignal2**)*

- Une équation d'horloge qui synchronise tous les signaux définis en entrée, les signaux booléens de sortie, et le signal local de type *event* nommé *Tick*, dont la déclaration est précisé plus loin :

Tick ^= B***myInputSignal1*** ^= B***myInputSignal2*** ^= V***myInputSignal2***

- Des équations de définitions des signaux booléens de sortie du type :

  B***myOutputSignal1*** := ^ E***myOutputSignal1*** *default false*

  B***myOutputSignal2*** := ^ V***myOutputSignal2*** *default false*

2.2.2.1.3 Déclaration locales

- Un signal nommé *Tick* de type *event*,

- Le process main défini ci-après.

2.2.2.2 Process "main"

2.2.2.2.1 Déclaration de l'interface

Entrée :

- Un signal nommé *Tick* de type *event*,
- Pour chaque signal **myInputSignal** du paquetage "InputSignals", on déclare :

si **myInputSignal** n'a pas d'attribut, un signal nommé **myInputSignal** de type *event*,

si **myInputSignal** a un attribut, un signal nommé **myInputSignal**, de même type que l'attribut.

Sortie :

- Pour chaque signal **myOutputSignal** du paquetage "OutputSignals", on déclare :

si **myOutputSignal** n'a pas d'attribut, un signal nommé **myOutputSignal** de type *event*,

si **myOutputSignal** a un attribut, un signal nommé **myOutputSignal**, de même type que l'attribut.

2.2.2.2.2 Corps

L' équation d'appel du process sub_main défini plus loin, qui définit les signaux de sortie du process "main".

2.2.2.2.3 Déclaration locales

- Pour chaque attribut **myAttribute** de type **myType** de valeur initiale **myInitialValue** des classes du paquetage "Core", on déclare un signal local statevar qui lui correspond :

*statevar **myType myAttribute** init **myInitialValue**;*

Si l'attribut n'a pas de valeur initiale, il n'y a pas de valeur initial en langage Signal :

*statevar **myType myAttribute**;*

- Le process sub_main défini ci-après.

2.2.2.3 Process "sub_main"

2.2.2.3.1 Déclaration de l'interface

La même que le process "main" défini plus haut.

2.2.2.3.2 Corps

Pour chaque classe active du paquetage "Core", l'équation d'appel du process chart défini plus loin, qui définit un ou plusieurs signaux de sortie du process "sub_main".

2.2.2.3.3 Déclarations locales

Pour chaque classe active **MyActiceClass** du paquetage "Core", on déclare un *use* du module associé à la classe active, qui a le nom de la classe :

*use **MyActiveClass**;*

2.2.3 Génération des modules

2.2.3.1 Organisation d'un module

Pour chaque classe du paquetage "Core", on génère un module Signal qui comprend :

- la déclaration d'un type énuméré qui représente les différentes valeurs de l'état, si la classe est active,
- la définition d'un process "chart", si la classe est active,
- pour chaque opération de la classe, la définition d'un process correspondant à l'opération,
- pour chaque autre classe que la classe utilise, la déclaration d'un "use" du module correspondant à la classe utilisée.

2.2.3.2 Déclaration du type énuméré

Si la classe **MyActiveClass** est un classe active, qui possède n états **myState1**, **myState**,… **myStaten**, on déclare un type énuméré représentant les différentes valeurs des états possibles.

*type **MyActiveClass**_states = enum(**myState1, myState2,… myStaten**);*

2.2.3.3 Définition du process "chart"

Si la classe **MyActiveClass** est active, on lui définit un process nommé **MyActiveClass**_chart, qui décrit le comportement réactif de la classe, et dont la construction s'appuie essentiellement sur les informations contenues dans la machine à états.

2.2.3.3.1 Déclaration de l'interface

En entrée :

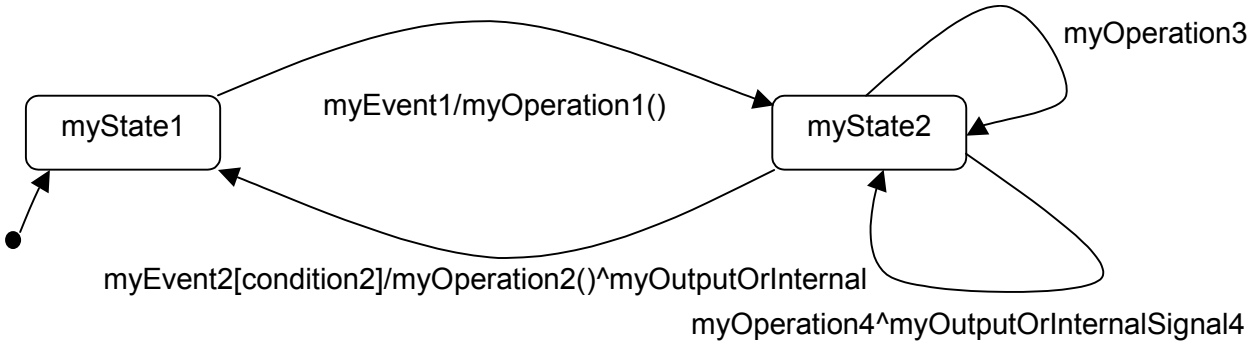
- Un signal nommé *Tick* de type *event*,
- Pour chaque DataFlow entrant instanciant un signal UML **mySignal**, un signal nommé **mySignal**. Le type sera *event* si le signal **mySignal** n'a pas d'attribut, le type de l'attribut si le signal **mySignal** a un d'attribut.

En sortie :

Pour chaque DataFlow sortant, un signal dont le nom et le type sont définis par la même règle qu'en entrée.

2.2.3.3.2 Corps

Supposons que le comportement de la classe active **MyActiveClass** soit décrit par le diagramme d'états-transitions suivant :



où **myEvent1** et **myEvent2** instancient respectivement les signaux **myInputOrInternalSignal1**, qui n'a pas d'attribut, et **myInputOrInternalSignal2**, qui a un attribut, et **condition2** est une condition sur la valeur de l'attribut du signal **myInputOrInternalSignal2**, dont la syntaxe a un sens en langage Signal.

Le corps comprend :

- Une équation de définition du signal d'état retard :

MyActiveClass**_previousState := **MyActiveClass**_currentState$ init #**myState1

- Un équation d'horloge qui synchronise le signal représentant l'état sur l'horloge maîtresse :

MyActiveClass_*currentState* ^= *Tick*

- Une équation de définition des changements d'état :

MyActiveClass_*currentState* := ***#myState2*** *when* ***^myInputOrInternalSignal1*** *when* ***MyActiveClass***_*previousState* == ***myState1***

default ***#myState1*** *when* ***^myInputOrInternalSignal2*** *when* **MyActiveClass**_*previousState* == ***myState2*** *when* ***condition2***

default ***MyActiveClass***_*previousState*

- Pour les transitions qui déclenchée par la réception d'un signal qui déclenche une opération, on définit :
  - un signal qui précise les instants d'appel de l'opération,
  - l'équation d'appel de l'opération.

Les définitions des instants d'appel s'écrivent :

*clk*_***myOperation1*** := *when* ***^myInputOrInternalSignal1*** *when* ***MyActiveClass***_*previousState* == ***myState1*** *when* ***MyActiveClass***_*currentState* == ***myState2***

et

*clk*_***myOperation2*** := *when* ***^myInputOrInternalSignal2*** *when* ***MyActiveClass***_*previousState* == ***myState2*** *when* ***MyActiveClass***_*currentState* == ***myState1*** *when* ***condition2***

Les appels d'opérations s'écrivent :

myOperation1(*clk*_***myOperation1***)

et

myInternalOrOutputSignal2 := ***myOperation2***(*clk*_***myOperation2***, ***myInputOrInternalSignal2*** *when clk*_***myOperation2***)

- Pour les états comme **myState2** dans lesquels s'effectuent des appels périodiques d'opérations, on définit :

- un signal, clk_**myState2**_state, qui précise les instants d'entrée dans l'état,

*start_**myState2**_state := when **myClass**_currentState == **myState2** when **MyActiveClass**_currentState /= **myState2***

- un signal, start_**myState2**_state, qui précise les instants où la classes est dans l'état,

*clk_**myState2**_state := when **MyActiveClass**_previousState == **myState2***

- pour chaque opération périodique de l'état, comme **myOperation3** et **myOperation4** :

- l'horloge d'un signal "compteur", qui compte les tops d'horloge,

*counter_**myOperation3** ^= (start_**myState2**_state ^+ Top) ^* (start_**myState2**_state +^ clk_**myState2**_state)*

et

*counter_**myOperation4** ^= (start_**myState2**_state ^+ Top) ^* (start_**myState2**_state +^ clk_**myState2**_state)*

- un signal qui précise les instants d'appel de l'opération,

*(counter_**myOperation3**, clk_**myOperation3**) := topmod{**period3**}(start_**myState2**_state)*

et

*(counter_**myOperation4**, clk_**myOperation4**) := topmod{**period4**}(start_**myState2**_state)*

où **period3** et **period4** sont les périodes des opérations **myOperation3** et **myOperation4**, troisièmes paramètres des tagged value RTF des opérations,

- l'équation d'appel de l'opération.

***myOperation3**(clk_**myOperation3**)*

et

***myOutputOrInternalSignal4** := **myOperation4**(clk_**myOperation4**)*

2.2.3.3.3 Déclarations locales

- Les signaux locaux de définitions de l'état de la classe,, si la classe est active :

***MyActiveClass**_states **MyActiveClass**_currentState, **MyActiveClass**_previousState;*

- Pour chaque état, qui possède un ou plusieurs traitements périodiques, un signal de définition de l'instant d'entrée dans l'état et un signal de définition des instants où la classe est dans l'état, si la classe est active :

*event start_**myState2**_state, clk_**myState2**_state;*

- Pour chaque opération invoquée au moins dans une transition, on déclare un signal de type *event* **;**

*event clk_**myOperation1**;*

*event clk _**myOperation2**;*

*event clk _**myOperation3**;*

*event clk _**myOperation4**;*

- Pour chaque opération invoquée au moins dans une transition périodique, on déclare un signal "compteur"de type *integer* **:**

*integer counter_**myOperation3**;*

*integer counter_**myOperation4**;*

2.2.3.4 Définition des process associés aux opérations de la classe

Pour chaque opération **myOperation** de la classe **MyClass**, on définit un process nommé ***MyClass_myOperation***.

2.2.3.4.1 Interface

En entrée :

- Un signal de type *event* nommé *clk*,
- Pour chaque paramètre d'entrée de l'opération, un signal de même nom et de même type.

En sortie :

Si l'opération a un paramètre de retour, on déclare un signal de sortie nommé ***output*** de même type que le paramètre de retour.

Si l'opération n'a pas de paramètre de retour, on ne déclare pas de signaux de sortie.

2.2.3.4.2 Spécifications de propriétés d'horloge

Si l'opération **MyClass_myOperation** possède les paramètre d'entrée **myInputParam1** et **myInputParam2**, et un paramètre de retour, on spécifie l'égalité des horloges des signaux correspondants avec le signal *clk*:

*spec(| clk ^= **myInputParam1** ^= **myInputParam2** ^= o_**MyClass_myOperation** |)*

Si l'opération n'a ni paramètre d'entrée ni paramètre de sortie, il n'y a pas de spécifications de propriétés d'horloges à écrire.

2.2.3.4.3 Corps

Si une note de type "Signal Language Body" est attachée à l'opération, son contenu sera le corps du process associé à l'opération.

Si aucune note de type "Signal Language Body" n'est attachée à l'opération, le corps du process associé à l'opération sera vide.

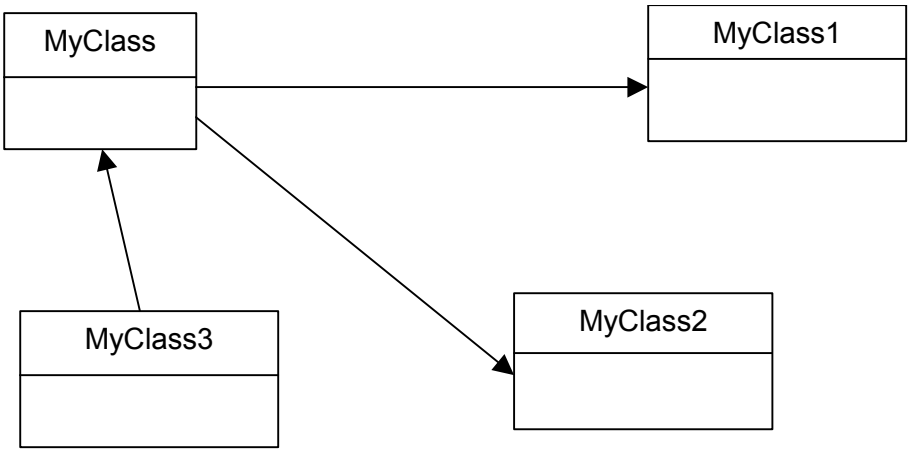
2.2.3.4.4 Déclarations locales

Si une note de type "Signal Language Local Declarations" est attachée à l'opération, son contenu sera placé en déclarations locales du process associé à l'opération.

Si aucune note de type "Signal Language Local Declarations" n'est attachée à l'opération, le process associé à l'opération ne possèdera pas de déclarations locales.

2.2.3.5 Déclaration des *use module*

Supposons que la structure de notre système décrite par le diagramme de classe ci-dessous :



A la fin du module correpondant à la classe **MyClass**, on déclare un *use* du module correspondant à chaque classe "utilisée" :

*use **MyClass1**;*

*use **MyClass2**;*

Si la classe est active, on déclare en plus un ***use*** du module *General*.

use General;

3. PERSPECTIVES

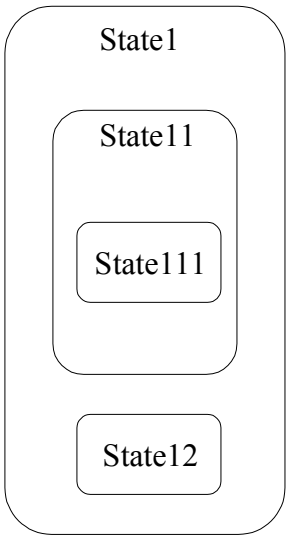
Traitement des machines à états présentant des états composites :

La traduction en langage Signal s'effectue selon un schéma qui équivaut à une mise à plat du modèlen dans lequel seuls les sous-états atomiques sont utilisés.

Règle de nommage des sous-états

Le nom d'un sous-état en langage Signal rappellera tous états parents.

Soit la hiérarchie d'états suivante :



Les états S1, S11, S111 et S12 seront respectivement nommés dans le programme Signal : S1, S1_S11, S1_S11_S111, S1_S11_S12.

Principe de mise à plat des daigrammes d'états-transitions

Soit T une transition de l'état S1 vers l'état S2, déclenchée par la réception d'un événement instanciant un signal des paquetages InputSignals ou InternalSignals, caractérisée par event/operation()

Cas S1 atomique et S2 atomique :

On définit une seule transition T de S1 vers S2.

Cas S1 composite et S2 atomique :

On définit, pour chaque sous-état atomique de S1, une transition vers S2, caractérisée par event/operation().

Cas S1 atomique et S2 composite :

On définit une transition T de S1 vers le sous-état atomique d'entrée de S2, caractérisée par event/operation().

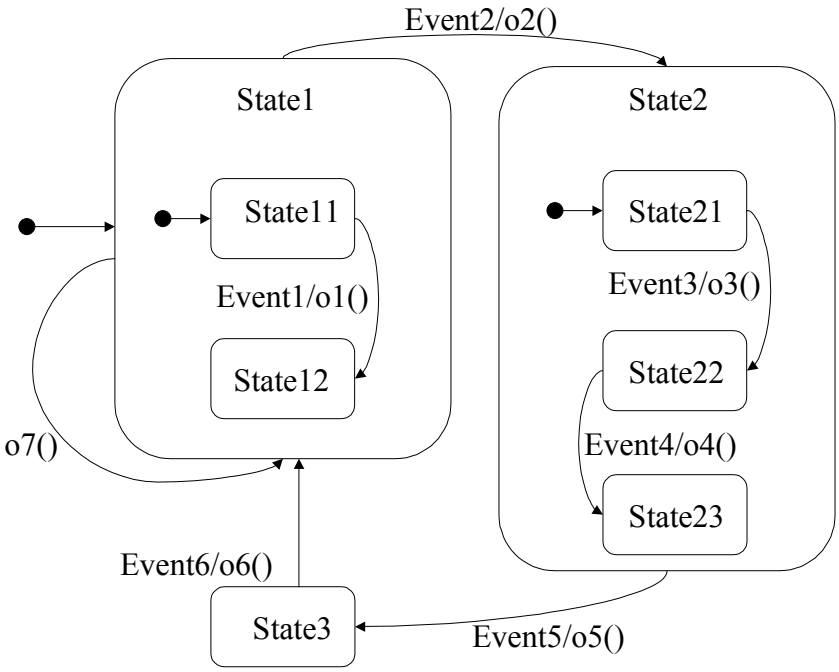
Cas S1 composite et S2 composite :

On définit, chaque sous-état atomique de S1, une transition vers le sous-état atomique d'entrée de S2, caractérisée par event/operation().

Soit T une transition periodique de l'état composite S :

On définit une transition périodique dont le compteur commence à l'instant de l'entrée dans le sous-état atomique d'entrée de S, et qui est déclenchée tant que la classe se trouve dans l'un des sous-états atomiques de S.

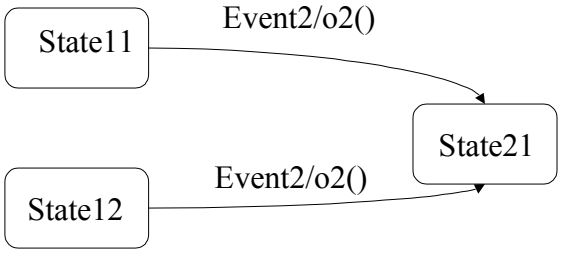
Soit le diagramme d'états-transitions suivants :



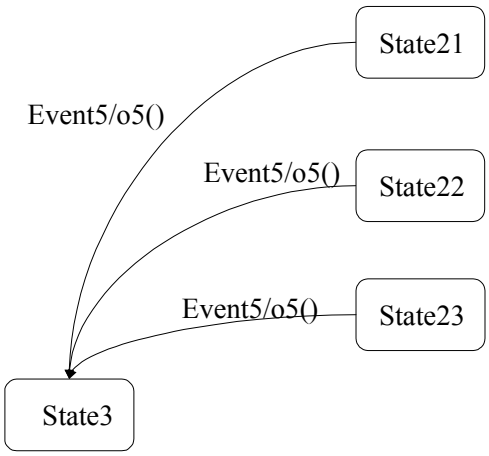
On note Ti la transition caractérisé par

Les transitions T1, T3, T4 ne posent pas de problème.

Traitement de la transition T2 caractérisée par Event2/o1() :



Traitement de la transition T5 caractérisée par Event5/o5() :



Traitement de la transition T6 caractérisée par Event6/o6() :