

# Modélisation Transactionnelle des Systèmes sur Puces en SystemC

## Ensimag 3A — filière SLE

### Grenoble-INP

#### Communications haut-niveau

Matthieu Moy  
(transparents originaux de Jérôme Cornet)

Matthieu.Moy@imag.fr

2015-2016



# Planning approximatif des séances

- 1 Introduction : les systèmes sur puce
- 2 Introduction : modélisation au niveau transactionnel (TLM)
- 3 Introduction au C++
- 4 Présentation de SystemC, éléments de base
- 5 **Communications haut-niveau en SystemC**
- 6 Modélisation TLM en SystemC
- 7 TP1 : Première plateforme SystemC/TLM
- 8 Utilisations des plateformes TLM
- 9 TP2 (1/2) : Utilisation de modules existants (affichage)
- 10 TP2 (2/2) : Utilisation de modules existants (affichage)
- 11 Notions Avancé en SystemC/TLM
- 12 TP3 (1/3) : Intégration du logiciel embarqué
- 13 TP3 (2/3) : Intégration du logiciel embarqué
- 14 TP3 (3/3) : Intégration du logiciel embarqué
- 15 05/01 : Intervenant extérieur : Jérôme Cornet
- 16 Perspectives et conclusion

# Sommaire

- 1 (Ré)visions de C++ : épisode 2
- 2 SystemC : Communications haut-niveau

# Sommaire

- 1 (Ré)visions de C++ : épisode 2
- 2 SystemC : Communications haut-niveau

# Sommaire de cette section

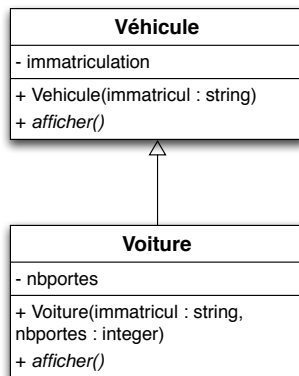
## 1 (Ré)visions de C++ : épisode 2

- Méthodes virtuelles
- Héritage multiple
- Héritage virtuel

# Méthodes virtuelles

- Définition : fonctions que l'on peut ré-implémenter dans une classe fille, avec liaison dynamique

- Exemple :



# Méthodes virtuelles

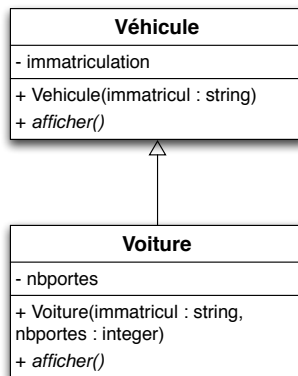
- Définition : fonctions que l'on peut ré-implémenter dans une classe fille, avec liaison dynamique

## Question



Quel est l'équivalent en Java ?

- Exemple :



## Exemple (déclaration)

```
class Vehicule
{
    public:
        Vehicule(const string & immatricul);

        // fonction virtuelle
        virtual void afficher();

    private:
        string immatriculation;
};
```



## Exemple (implémentation)

```
Vehicule::Vehicule(const string & immatricul)
{
    immatriculation = immatricul;
}

void Vehicule::afficher()
{
    cout << "Immatriculation : " << immatriculation
        << endl;
}
```

## Exemple (déclaration)

```
class Voiture : public Vehicule
{
    public:
        Voiture(const string & immatricul,
                int nombredeportes);

        // fonction virtuelle
        virtual void afficher();

    private:
        int nbportes;
};
```

## Exemple (implémentation)

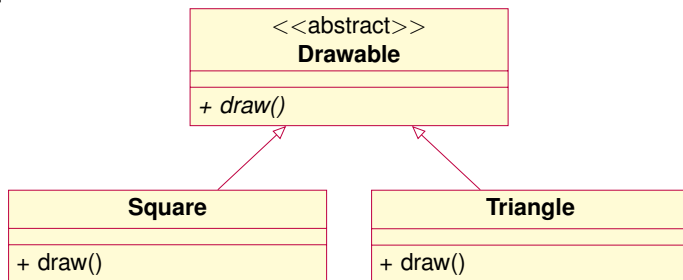
```
Voiture::Voiture(const string & immatricul,
                 int nombreportes)
    : Vehicule(immatricul)
{
    // suite des initialisations
    nbportes = nombreportes;
}

void Voiture::afficher()
{
    // appel de la fonction virtuelle de la classe mere
    Vehicule::afficher();

    cout << "Nb de portes : " << nbportes << endl;
}
```

# Méthodes virtuelles pures

- Définition : méthodes virtuelles pour lesquelles
  - ▶ On ne donne pas d'implémentation dans la classe mère,
  - ▶ On **force** l'implémentation dans les classes filles.
- Exemple :



- Une classe contenant une méthode virtuelle pure est abstraite

## Exemple (déclaration)

```
class Drawable
{
    public:

        ...

        // methode virtuelle pure
        // pas d'implementation associee dans le .cpp
        virtual void draw() = 0;
        // le "= 0" est la syntaxe pour "virtuelle pure"
        // rien a voir avec une initialisation.

    private:

        ...

};
```

## Exemple (déclaration)

```
class Ligne : public Drawable
{
    public:
        ...

        // methode virtuelle
        virtual void draw();

    private:
        ...
};
```

## Exemple (déclaration)

```
// debut du fichier .cpp

...

void Ligne::draw()
{
    // instructions de dessin de la ligne
    ....
}
```

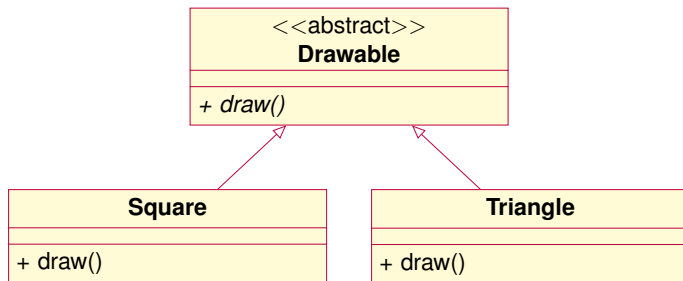
# Exemple complet minimaliste

`code/dessiner/dessiner.cpp`



# Classes abstraites

- Définition : classe contenant au moins une méthode **virtuelle pure**
- Exemple précédent : classe Drawable



- Impossible d'instancier un objet d'une classe abstraite

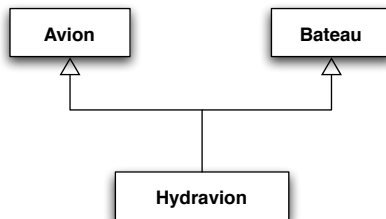
# Sommaire de cette section

## 1 (Ré)visions de C++ : épisode 2

- Méthodes virtuelles
- Héritage multiple
- Héritage virtuel

# Héritage multiple : présentation

- Possibilité d'hériter de **plusieurs classes**

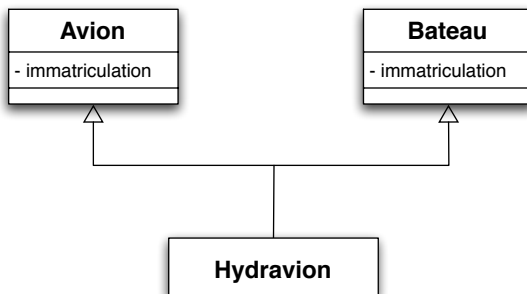


- Syntaxe :

```
class Hydravion : public Avion, public Bateau
{
    ...
};
```

# Héritage multiple : problème des homonymes

- Ambiguïté lorsque les deux classes mères ont des attributs/méthodes de même nom



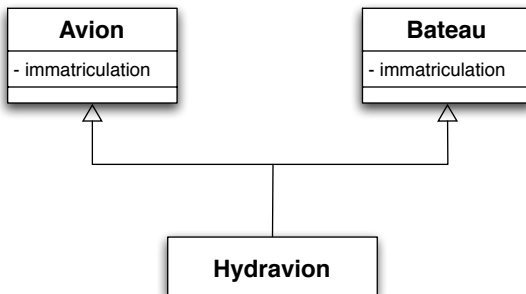
## Question



Quelle méthode choisir ?

# Héritage multiple : problème des homonymes

- Ambiguïté lorsque les deux classes mères ont des attributs/méthodes de même nom



- Résolution : emploi de l'opérateur de résolution de portée  
`Avion::immatriculation`, `Bateau::immatriculation`

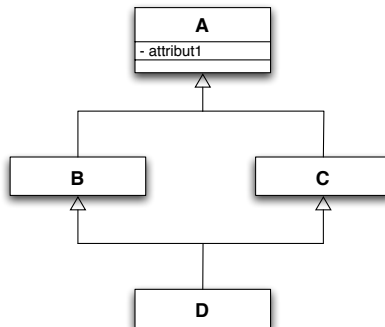
# Sommaire de cette section

## 1 (Ré)visions de C++ : épisode 2

- Méthodes virtuelles
- Héritage multiple
- Héritage virtuel

# Problème d'origine

- Problème dans la situation d'héritage multiple :



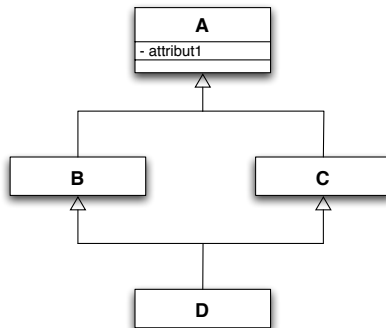
## Question



Quel est le problème ?

# Problème d'origine

- Problème dans la situation d'héritage multiple :



- `attribut1` est hérité en double par D !
- Données de A en double dans D, double appel du constructeur de A à la construction de D



## Solution : héritage virtuel

- Rien à voir avec les méthodes virtuelles !
- Utilisation du mot-clé `virtual`
- Sur l'exemple précédent :

```
class B : virtual public A
{
    public:
        B();
    ...
};
```

```
class C : virtual public A
{
    public:
        C();
    ...
};
```

## Solution : héritage virtuel

- Déclaration de la classe D :

```
class D : virtual public A,  
    // pour pouvoir appeler directement son constructeur  
    public B, public C  
    // C'est vraiment d'elles qu'on herite.  
{  
    public:  
        D();  
        ...  
};
```

- Implémentation de la classe D :

```
D::D() : A(),  
        B(),  
        C()  
{ /* suite des initialisations */ }
```

# Bilan sur l'héritage virtuel

- Permet d'éviter les ambiguïtés en cas d'héritage multiple
- À utiliser à bon escient !
  - ▶ Si les classes héritant d'une même classes de base sont susceptibles d'être dérivées en même temps
- Suite du cours : utilisation bien spécifique (`sc_interface`)

# Sommaire

- 1 (Ré)visions de C++ : épisode 2
- 2 **SystemC : Communications haut-niveau**

# Sommaire de cette section

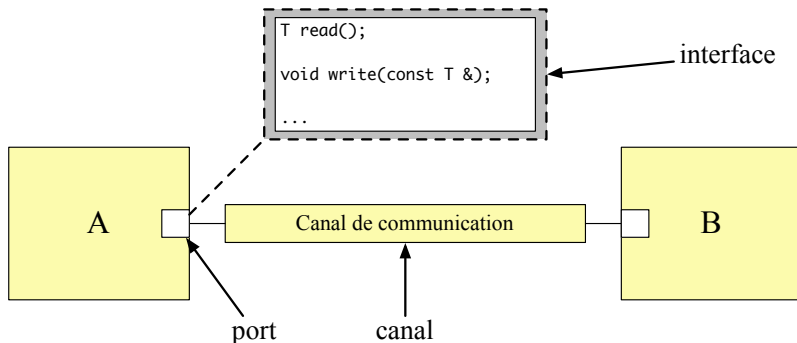
## 2 SystemC : Communications haut-niveau

- Objectifs
- Interfaces
- Ports génériques
- Canaux de communication primitifs
- Canaux prédéfinis

# Objectifs

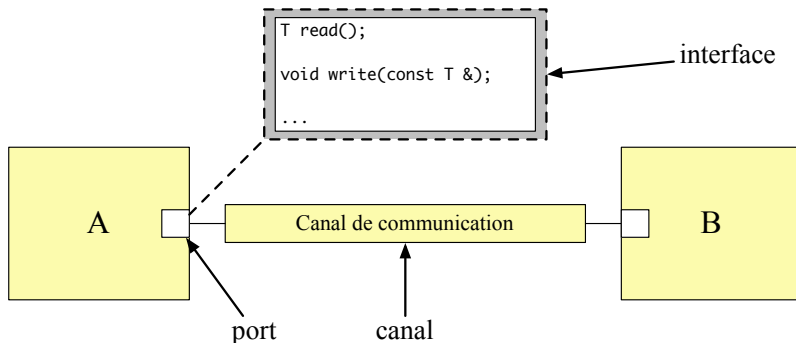
- Comprendre le cadre global de définition des communications en SystemC
- Définition de nouveaux modes de communications
- Étude des communications haut-niveau pré-définies

# Concepts



- But :  $\approx$  Appel de méthode distante
- $\Rightarrow$  Permettre à A d'appeler des fonctions de B (ou du canal) ...

# Concepts



- But :  $\approx$  Appel de méthode distante
- $\Rightarrow$  Permettre à A d'appeler des fonctions de B (ou du canal) ...  
... sans connaître B ni le canal a priori !



# Sommaire de cette section

## 2 SystemC : Communications haut-niveau

- Objectifs
- Interfaces
- Ports génériques
- Canaux de communication primitifs
- Canaux prédéfinis

# Interfaces

- Élément définissant les actions possibles pour réaliser une communication
- En pratique :
  - ▶ **Interface** SystemC : classe **abstraite** dérivant de `sc_interface`
  - ▶ **Actions possibles** : méthodes de cette classe
  - ▶ Généricité sur le type des données des communications
- Exemple : communication rendez-vous avec valeur
  - ▶ Lecture de valeur : action `get`
  - ▶ Écriture de valeur : action `put`
  - ▶ Deux modules communiquant : l'un en lecture, l'autre en écriture

# Interfaces

- Élément définissant les actions possibles pour réaliser une communication
- En pratique :
  - ▶ **Interface** SystemC : classe **abstraite** dérivant de `sc_interface`
  - ▶ **Actions possibles** : méthodes de cette classe
  - ▶ Généricité sur le type des données des communications
- Exemple : communication rendez-vous avec valeur
  - ▶ Lecture de valeur : action `get`
  - ▶ Écriture de valeur : action `put`
  - ▶ Deux modules communiquant : l'un en lecture, l'autre en écriture
- En deux temps :
  - 1 On dit que le canal accepte les actions `put/get` via une interface,
  - 2 On dit ce que fait le canal dans ces cas là.

## Exemple

- Exemple : communication rendez-vous avec valeur

```
template<typename T>  
class rendezvous_in_if : virtual public sc_interface
```

## Exemple

- Exemple : communication rendez-vous avec valeur

```
template<typename T>  
class rendezvous_in_if : virtual public sc_interface
```

### Question



Pourquoi “virtual” ?

## Exemple

- Exemple : communication rendez-vous avec valeur

```
template<typename T>
class rendezvous_in_if : virtual public sc_interface
{
    public:
        // methode virtuelle pure
        virtual T get() = 0;
};
```

```
template<typename T>
class rendezvous_out_if : virtual public sc_interface
{
    public:
        // methode virtuelle pure
        virtual void put(const T & val) = 0;
};
```

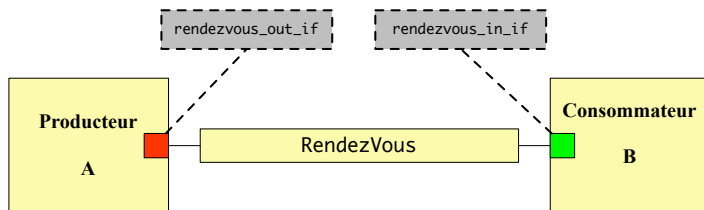
# Sommaire de cette section

## 2 SystemC : Communications haut-niveau

- Objectifs
- Interfaces
- Ports génériques
- Canaux de communication primitifs
- Canaux prédéfinis

# Ports génériques

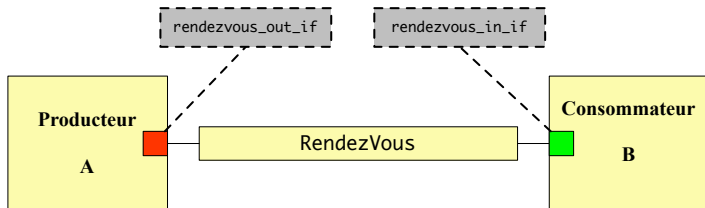
- Objets fournissant un point de connexion dans le module
- En pratique :
  - ▶ Objet de la classe `sc_port`
  - ▶ Généricité sur l'**interface**
  - ▶ Utilisation : `sc_port<interface>`
- Exemple : communication rendez-vous avec valeur





# Ports génériques : à l'intérieur

- Surcharge des opérateurs `*` et `->` :
- `port->foo()`  $\Leftrightarrow$  `canal.foo()`
- $\Rightarrow$  permet d'utiliser le canal sans savoir a priori lequel c'est.



## Exemple de code de modules

- Exemple : communication rendez-vous avec valeur

```
SC_MODULE (Producteur)
{
    sc_port<rendezvous_out_if<int> > sortie;

    SC_CTOR (Producteur);
    void production();
};

SC_MODULE (Consommateur)
{
    sc_port<rendezvous_in_if<int> > entree;

    SC_CTOR (Consommateur);
    void consommation();
};
```

# Utilisation (1/2)

## ● Exemple : Producteur

```
Producteur::Producteur(sc_module_name name)
    : sc_module(name)
{
    SC_THREAD(production);
}

void Producteur::production()
{
    for (int i=0; i<10; i++)
    {
        cout << "Envoi de " << i << endl;

        // attention -> n'a rien a voir avec un pointeur
        // ~ raccourci pour sortie.get_interface()->put(i)
        sortie->put(i);
    }
}
```

## Utilisation (2/2)

- Exemple : Consommateur

```
Consommateur::Consommateur(sc_module_name name)
                        : sc_module(name)
{
    SC_THREAD(consoommation);
}

void Consommateur::consoommation()
{
    while (true)
    {
        int valeur_recue = entree->get();

        cout << "Recu : " << valeur_recue << endl;
    }
}
```

# Retour sur RTL

- Éléments utilisés précédemment :
  - ▶ `sc_in<type>` : « raccourcis » pour  
`sc_port<sc_signal_in_if<type> >`
  - ▶ `sc_out<type>` : « raccourcis » pour  
`sc_port<sc_signal_out_if<type> >`
- Question ?

# Sommaire de cette section

## 2 SystemC : Communications haut-niveau

- Objectifs
- Interfaces
- Ports génériques
- Canaux de communication primitifs
- Canaux prédéfinis

# Canal de communication

- Définition : objet gérant les communications entre plusieurs modules
- Canal de communication **primitif** : canal construit dans le cadre de base fourni par SystemC
- Donne la sémantique des communications
- Donne les connexions autorisées
- En pratique :
  - ▶ Classe dérivant de `sc_prim_channel`
  - ▶ Implémente des interfaces de communications
  - ▶ Généricité sur le type des données des communications

# Exemple

- Exemple : communication rendez-vous avec valeur
  - ▶ Action `get` : lecture bloquante si pas de donnée disponible
  - ▶ Action `put` : écriture bloquante si pas de lecture par le module qui lit
  - ▶ Connexions uniquement entre deux modules



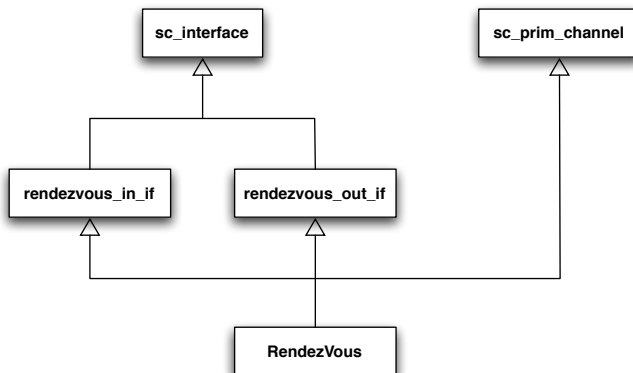
# Déclaration du canal correspondant

```
template<typename T>
class RendezVous : public sc_prim_channel,
                   public rendezvous_in_if<T>,
                   public rendezvous_out_if<T>
{
public:
    RendezVous(const char *name);

    virtual T get();
    virtual void put(const T & val);

private:
    ...
}
```

# Organisation des classes



# Implémentation du canal correspondant

- Constructeur :

```
template<typename T>
RendezVous<T>::RendezVous(const char *name)
                        : sc_prim_channel(name)
{
    ...
}
```

# Implémentation du canal correspondant

- Accès en écriture :

```
template<typename T>
void RendezVous<T>::put(const T & val)
{
    ...
}
```

# Implémentation du canal correspondant

- Accès en lecture :

```
template<typename T>
T RendezVous<T>::get ()
{
    . . .
}
```

# Implémentation du canal correspondant

- Accès en écriture :

```
template<typename T>
void RendezVous<T>::put(const T & val)
{
    // "Ecrire" la valeur

    // Dire au processus qui lit que l'on a écrit

    // Attendre que le processus qui lit ait lu

}
```

# Implémentation du canal correspondant

- Accès en écriture :

```
template<typename T>
void RendezVous<T>::put(const T & val)
{
    // "Ecrire" la valeur
    shared_value = val;

    // Dire au processus qui lit que l'on a écrit

    // Attendre que le processus qui lit ait lu

}
```

# Implémentation du canal correspondant

- Accès en écriture :

```
template<typename T>
void RendezVous<T>::put(const T & val)
{
    // "Ecrire" la valeur
    shared_value = val;

    // Dire au processus qui lit que l'on a écrit
    put_ok = true;
    put_event.notify();

    // Attendre que le processus qui lit ait lu

}
```



# Implémentation du canal correspondant

- Accès en écriture :

```
template<typename T>
void RendezVous<T>::put(const T & val)
{
    // "Ecrire" la valeur
    shared_value = val;

    // Dire au processus qui lit que l'on a écrit
    put_ok = true;
    put_event.notify();

    // Attendre que le processus qui lit ait lu
    if (!get_ok)
        wait(get_event);

    get_ok = false;
}
```

# Implémentation du canal correspondant

- Accès en lecture :

```
template<typename T>
T RendezVous<T>::get ()
{
    // Attendre l'écriture de la valeur

    // Dire au processus qui écrit que l'on a lu

    // Retourner la valeur
}
```

# Implémentation du canal correspondant

- Accès en lecture :

```
template<typename T>
T RendezVous<T>::get ()
{
    // Attendre l'écriture de la valeur
    if (!put_ok)
        wait(put_event);
    put_ok = false;

    // Dire au processus qui écrit que l'on a lu

    // Retourner la valeur
}
```

# Implémentation du canal correspondant

- Accès en lecture :

```
template<typename T>
T RendezVous<T>::get ()
{
    // Attendre l'écriture de la valeur
    if (!put_ok)
        wait(put_event);
    put_ok = false;

    // Dire au processus qui écrit que l'on a lu
    get_ok = true;
    get_event.notify();

    // Retourner la valeur

}
```

# Implémentation du canal correspondant

- Accès en lecture :

```
template<typename T>
T RendezVous<T>::get ()
{
    // Attendre l'écriture de la valeur
    if (!put_ok)
        wait(put_event);
    put_ok = false;

    // Dire au processus qui écrit que l'on a lu
    get_ok = true;
    get_event.notify();

    // Retourner la valeur
    return shared_value;
}
```

# Déclaration complète

```
template<typename T>
class RendezVous : public sc_prim_channel,
                   virtual public rendezvous_in_if<T>,
                   virtual public rendezvous_out_if<T>
{
    public:
        RendezVous(const char *name);

        virtual T get();
        virtual void put(const T & val);

    private:
        T          shared_value;
        bool        get_ok, put_ok;
        sc_event    get_event, put_event;
};
```

# Implémentation du canal correspondant

- Constructeur complet :

```
template<typename T>
RendezVous<T>::RendezVous(const char *name)
                        : sc_prim_channel(name)
{
    shared_value = 0;
    get_ok = false;
    put_ok = false;
}
```



# Démo




# Sommaire de cette section

## 2 SystemC : Communications haut-niveau

- Objectifs
- Interfaces
- Ports génériques
- Canaux de communication primitifs
- Canaux prédéfinis

# Canaux prédéfinis dans SystemC

- `sc_mutex`

- ▶ Canal « exclusion mutuelle »
- ▶ Opérations : `lock()`, `unlock()` ...
- ▶ Verrouillage bloquant, déverrouillage non bloquant
- ▶ Version non bloquante du verrouillage : `trylock()`
- ▶   $\neq$  `pthread_mutex_t`

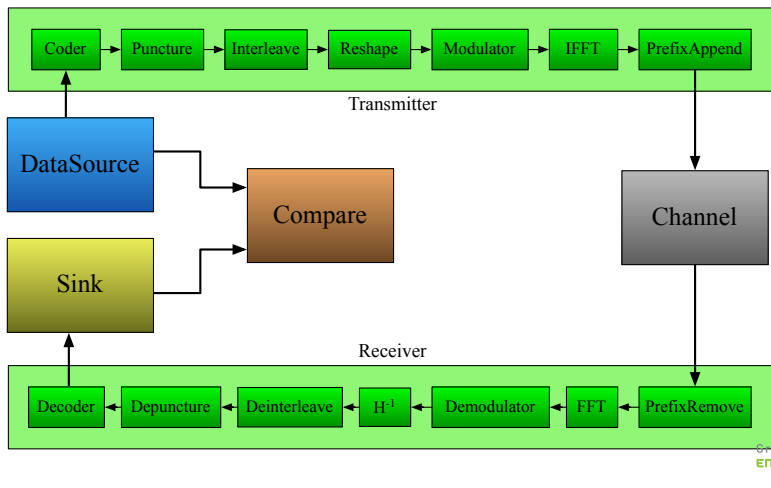
- `sc_fifo`

- ▶ File d'attente de taille fixe
- ▶ Opérations : `read()`, `write()` ...
- ▶ Versions non bloquantes

- D'autres non présentés : `sc_semaphore`, `sc_buffer`...

## Exemple d'utilisation de `sc_fifo`

- Modélisation flot de données (**dataflow**)
- Ex : traitement du signal (couche physique d'un modem radio)



# Conclusion

- Mécanisme général de définition des communications
- Réutilisation des éléments de base

# Conclusion

- Mécanisme général de définition des communications
- Réutilisation des éléments de base

## Question



Cela suffit pour modéliser des comportements initiateur/cible ?