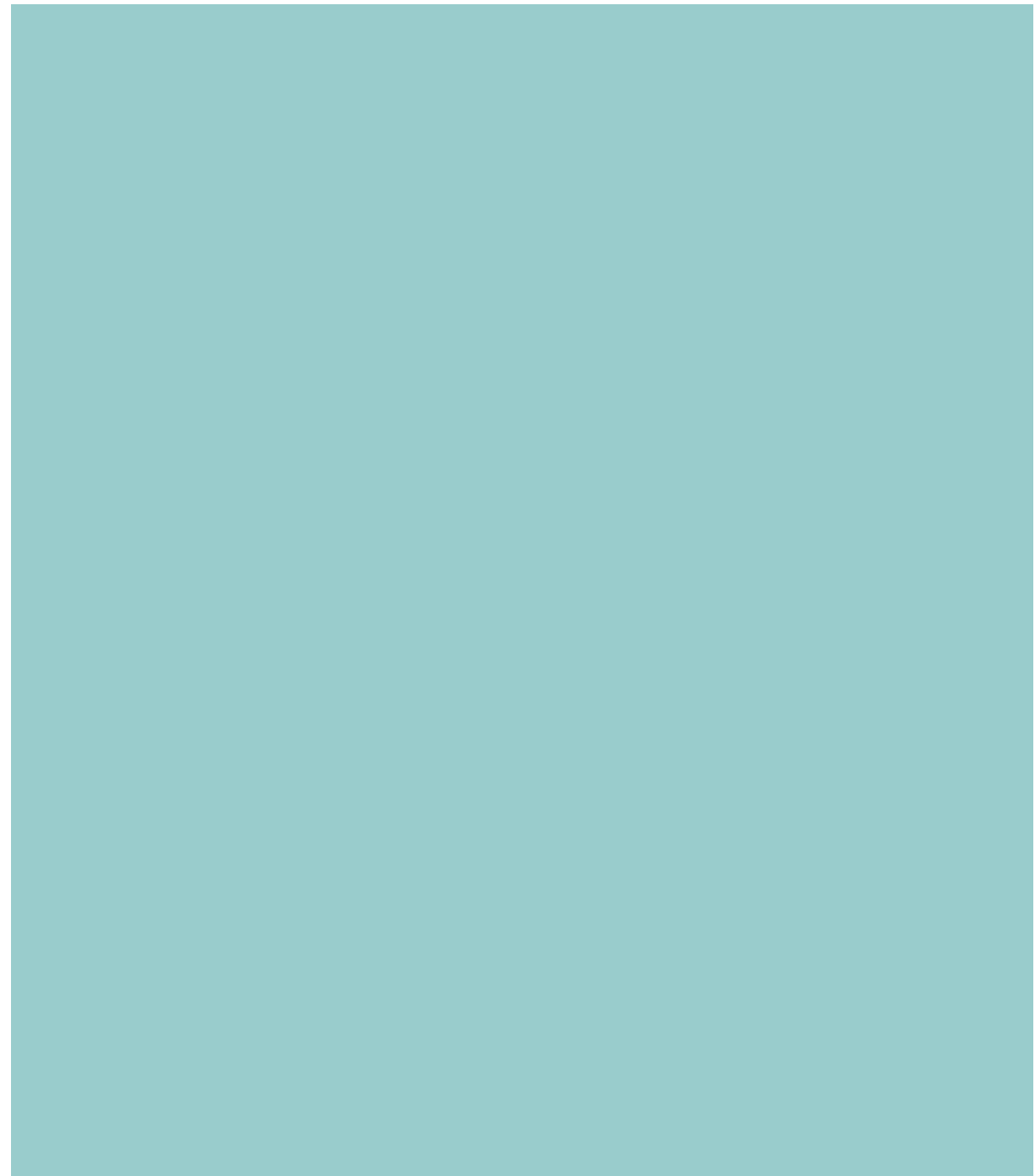


FMU BASED SIMULATION FRAMEWORK

Editeur graphique



Contenu

1.1. Présentation générale	3
2.2. Editeur graphique.....	4
1.1 Lancement de l'éditeur	4
1.2 Référencement des composants.....	4
1.3 Création de document.....	4
1.4 Edition d'un modèle	4
1.5 Barre d'outils	4

1.6 Edition d'un modèle multi-pages	5
1.7 Création d'un tunnel entre composants	5
1.8 Paramétrage d'un composant	5
2.2.Simulation d'un modèle	6
2.1 Résolution causale et retard	6
2.2 Simulation dans l'éditeur	6
2.3 Simulation dans FBSF.....	6
3.3.Conception d'un composant	7
3.1 Composant logique.....	7
3.2 Connexion a la zone publique	9
3.3 Master-composant	10
3.4 Héritage de composants.....	11
3.5 Composant a nombre de sockets variables.....	12

1. Présentation générale

L'éditeur graphique permet de concevoir des modèles par instanciation et assemblage de composants en provenance d'une bibliothèque. L'instanciation s'effectue par une opération de glisser-déposer depuis une bibliothèque de composants vers le modèle. L'assemblage a lieu par établissement de liens graphiques entre les composants.

Un document est l'élément qui porte le modèle c'est-à-dire l'assemblage de composants. Le document peut être composé d'une ou plusieurs pages et porte la mémoire des attributs graphiques et des paramètres des composants.

Un composant graphique est un objet qui implémente:

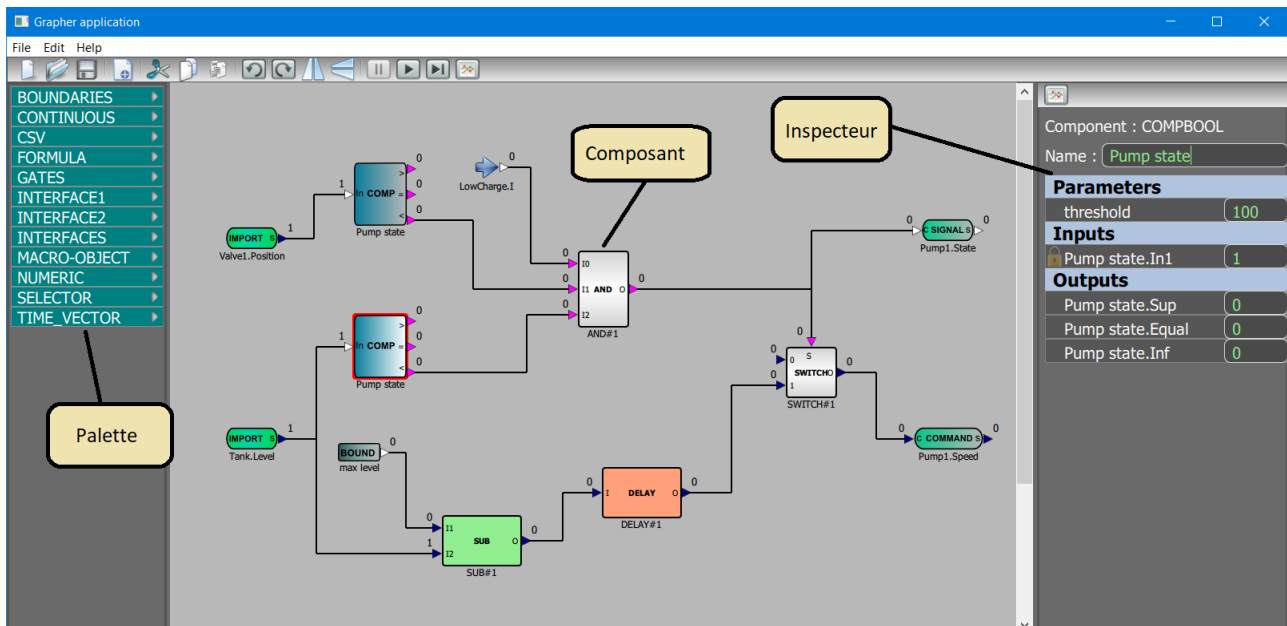
- une interface graphique
- une logique de comportement

Les composants sont classés par types et organisés en bibliothèque:

Les composants de type logique sont connectables par liens graphique tracés entre les ports orientés ou «sockets» des composants.

Les sockets indiquent leur type par un code couleur :

- bleu : type réel
- orange : type int
- rose : type boolean
- blanc : non typée



1. Editeur graphique

1.1. Lancement de l'éditeur

Le lancement de l'éditeur est réalisé par la commande suivante :

```
FbsfEditor.exe [-no-logfile] [-no-update]
```

-no-logfile: permet de produire les traces dans la console au lieu du fichier.

-no-update: commutateur de mise à jour ou non du fichier des composants

1.2 Référencement des composants

Les composants sont organisés sous un dossier Library par domaine en l'occurrence dossier «LOGIC» .

Pour être utilisables, les composants doivent être référencés dans le fichier **qmldir** localisé à la racine du dossier domaine de la bibliothèque selon le format suivant:

	<Identifiant du type>	<version>	<chemin relatif>
ex:	AND	1.0	GATES/AND.qml

Le fichier qmldir est **mis à jour automatiquement** à chaque lancement de l'éditeur en fonction du contenu du dossier des composants.

Une option «-no-update» sur la ligne de commande de l'éditeur permet d'inhiber cette opération dans le cas d'une gestion manuelle.

1.3 Création de document

La création du document implique le choix initial du type («Logic» ou «Graphic»).

Le choix du type permet de présenter les composants par catégorie dans la palette.

1.4 Edition d'un modèle

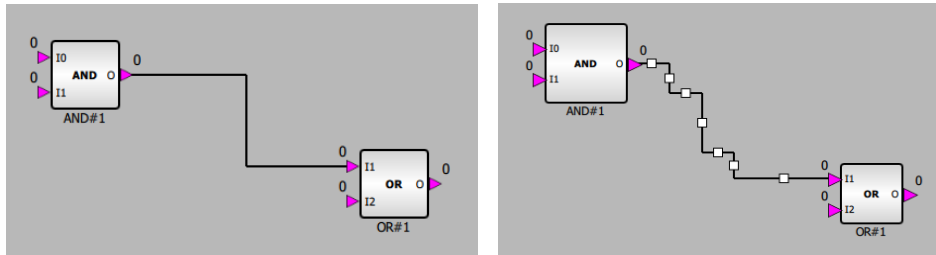
Un modèle est développé par l'instanciation, le paramétrage et l'assemblage de composants issus d'une bibliothèque.

L'instanciation est réalisée par Drag and Drop depuis la palette située a gauche.

La liaison des composants logique est réalisée par:








- Sélection par clic gauche d'une socket source,
- déplacement du curseur,
- sélection par clic gauche d'une socket cible

Note: la touche «echap» permet de supprimer le lien en cours de traçage et , par sélection, les poignées en milieux de segment permettent de déplacer ou d'aligner des segments.



1.5 Barre d'outils

Une barre d'outils propose les fonctions suivantes:

	Nouveau document
	Ouvrir un document
	Sauver un document
	Créer une page
	Couper une sélection
	Copier une sélection
	Coller une sélection

1.6 Edition d'un modèle multi-pages

Un nouveau modèle est par défaut composé d'une seule page mais des pages supplémentaires peuvent être ajoutées par clic sur le bouton «New page»:



Le passage d'une page à l'autre est rendu possible par les fonctionnalités d'exploration offertes par les boutons en bas à droite (renommage, précédente, suivante et menu) :



1.7 Création d'un tunnel entre composants

Le tracé d'un lien graphique entre deux composants peut être interrompu (i.e. rendu invisible) par la mise en place d'un tunnel. La création des ports d'entrée et de sortie du tunnel s'effectue par un shift + clic-gauche (en fin de tracé de la liaison pour une entrée de tunnel).

Ces ports se matérialisent par le pictogramme suivant:



Un tunnel peut être utilisé pour alléger limiter le nombre de fil sur une page.

Un tunnel doit être utilisé dans le cas où l'on souhaite connecter deux composants situés sur des pages différentes d'un même document.

Un port de sortie du tunnel doit être nommé selon la règle **<node name>.<socket name>** pour être connecté à une entrée de tunnel.

1.8 Paramétrage d'un composant

Le paramétrage d'un composant a lieu en cliquant sur celui-ci et en modifiant les valeurs dans l'inspecteur à la droite de l'écran.

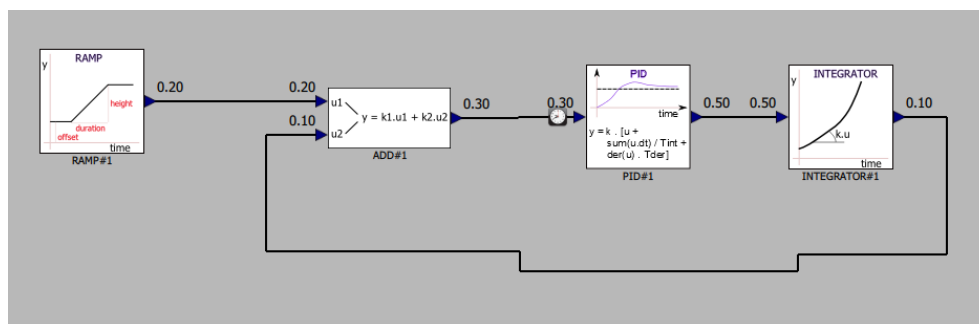
Le actions possibles sont :

- Saisie du nom de l'instance,
- Modification des valeurs de paramètres,
- forçage d'une entrée avec verrou.

2. Simulation d'un modèle

2.2 Résolution causale et retard

La résolution d'un modèle se déroule de manière chaînée, c'est-à-dire composant après composant, selon un ordre défini par la chaîne de causalité ou d'entrée-sortie. Le pas de temps de simulation est fixe. Lorsqu'une boucle est détectée dans la chaîne de causalité, un retard d'un pas de temps est automatiquement positionné par le système. Ce retard est matérialisé par la présence d'une petite horloge comme sur la copie d'écran ci-dessous :



2.3 Simulation dans l'éditeur

La simulation dans l'éditeur est possible grâce aux boutons pause/play/step by step suivants:



Dans l'éditeur, la simulation a lieu avec un pas de temps de 0.2 s.

2.4 Simulation dans FBSF

La simulation dans FBSF nécessite un enregistrement du modèle et renseignement d'un fichier de configuration contenant les éléments suivants (en bleu les identifiants applicatifs):

```
<model version="1.0">  
  <name>LogicModel</name>  
  <module>ModuleLogic</module>  
  <type>visual</type>  
  <document>./LogicModel.qml</document>  
</model>
```

3. Conception d'un composant

3.2 Composant logique

Un composant est implémenté en langage QML pour ses propriétés et Javascript pour le comportement. Il est possible de spécifier:

- Son aspect graphique avec :
 - Une image localisée au même endroit que le fichier qml:

```
source: "nom de l'image" (png ou svg)
```

- Ou par défaut un rectangle simple avec possibilité de spécifier:

La couleur par gradients

```
gradient1: "grey"  
gradient2: "lightgrey"
```

Les dimensions en pixels

```
shape.width: 100  
shape.height: 100
```

- Ses connexions positionnées et typées :

- Entrée (mot clé "**input**")

```
socketsLeft: [  
  Socket {id: in1;socketId: "I1";direction: "input"},  
  Socket {id: in2;socketId: "I2";direction: "input"}  
]
```

- Sortie (mot clé "**output**")

```
socketsRight: [  
  Socket {id: out;socketId:"O";direction: "output"}  
]
```

- Type (mot clé "**type**" par défaut «notype»)

Types définis et couleurs associées : "**real**" ► , "**int**" ► et "**bool**" ►

```
socketsRight: [  
  Socket {id: out;socketId:"O";direction: "output";type:"bool"}  
]
```

- Ses propriétés internes (attributs de type standard QML)

```
property int value : 0  
property real : 1.0
```

- Ses paramètres, constantes et variables publiques (visibles depuis l'inspecteur d'objet)

```
property var parameters : {"step" : 10 , "max" : 100}  
property var constants : {"k" : 10 , "c1" : true}  
property var variables : {'value' : 0}
```


- Ses variables d'état (gestion d'état en fichier)

```
property var statesvar :{"varList":[],"tIndex":0}
```

- Les attributs hérités **nodeName**, **nodeType**, **simulationTime** et **timeStep** disponible pour les traces et le traitement.
- Une fonction d'initialisation

```
function initialize()
{
    console.log(nodeName,"initialize timeStep :", timeStep)
}
```

- Un comportement périodique

```
function compute()
{
    variables.value += parameters.step
    out.value = variables.value
}
```

les identifiants «**parameters**», «**constants**», «**variables**» et «**statesvar**» sont des mots clés, ils permettent d'accéder aux variables et paramètres par leur nom.

ex: **variables.value**, **parameters.step** , **constants.k** ou **statesvar. tIndex**

Les valeurs des sockets en entrée et en sortie sont accessibles par:

<id>.**value** (ou id et value sont des attributs de la socket)

ex: **out.value**

- Un comportement événementiel:
 - Avant Sauvegarde et après Restitution d'état par clic, par exemple pour sauver et restituer la valeur exportée depuis le composant :

```
property real command: 0.0
property var statesvar : {"command" : 0.0}
SignalReal {tag1:nodeName;value:command}
function stateSaving()
{
    console.log(nodeName,"saved time:",simulationTime)
    statesvar.command=command
}
function stateRestored()
{
    command=statesvar.command
    console.log(nodeName,"restore time:",simulationTime)
}
```

- Modification d'un paramètre depuis l'inspecteur (signal)

l'argument "**name**" identifie le paramètre modifié

```
onParameterModified :{ if (name==="param1") .....}
```

3.3 Connexion a la zone publique

Abonnement a une donnée

L'abonnement a une donnée de la zone publique est réalisée par les déclarations **SubscribeReal** et **SubscribeInt**

```
import QtQuick 2.0
import Grapher 1.0
import fbsfplugins 1.0
//~~~~~
// Subscription to a external value
//~~~~~
property real mValue: 0
SubscribeReal{
    tag1: nodeName;
    onValueChanged: {mValue = value;} }
```

Publication d'une donnée

L'abonnement a une donnée de la zone publique est réalisée par les déclarations **SignalReal** et **SignalInt**

```
import QtQuick 2.0
import Grapher 1.0
import fbsfplugins 1.0
//~~~~~
// Notification of a command value
//~~~~~
property real command: NaN
property var statesvar : {"command" : NaN}

SignalReal {
    tag1      : nodeName
    unit      : "var unit"
    description : "var description"
    value     : mCommand
}

function stateSaving()      { statesvar.command=command }
function stateRestored()   { command=statesvar.command }
function compute()         { command = inputvalue }
```

Note : Dans le cas d'une publication depuis un composant :

- La donnée publiée doit être initialisée a NaN pour permettre le fonctionnement du trigger de type onValueChanged qui publie la valeur en zone d'échange.
- La gestion d'état (sauvegarde et restitution) de la donnée publiée doit être traitée depuis les triggers **stateSaving** et **stateRestored** .

3.4 Master-composant

Il est possible d'encapsuler un document en tant que composant .

Le document est localisé dans un dossier «Macro» au niveau du répertoire du master-composant.

Les ports in et out du document sont connectés aux sockets du master-composant avec la fonction:

bindPorts(<master socket id>,<document port name>)

Les fonctions d'initialisation et de traitement cyclique du master-composant propagent la logique d'appel vers le document.

```
import QtQuick 2.0
import Grapher 1.0

Node
{
    source:nodeType + ".svg"
    //~~~~~
    socketsLeft: [
        Socket {id: in1;socketId: "In1";direction: "input";type:"real"},
        Socket {id: in2;socketId: "In2";direction: "input";type:"real"}
    ]
    socketsRight: [
        Socket {id: out;socketId:"Out";direction: "output";type:"int"}
    ]
    //~~~~~
    property Document doc
    Component.onCompleted:
    {
        doc=loadDocument("Macro/test1.qml")
        bindPorts(in1,"InPort.1")
        bindPorts(in2,"InPort.2")
        bindPorts(out,"COUNTER#1.count")
    }
    //~~~~~
    function initialize()
    {
        doc.initialize(timeStep,pastSize,futurSize,timeShift)
    }
    function compute()
    {
        doc.compute(simulationTime)
    }
}
```

3.5 Héritage de composants

Il est possible de coder un composant générique (ex : NodeBase.qml) et d'en hériter depuis d'autres composants.

```
import QtQuick 2.0
import Grapher 1.0
NodeBase
{
    property bool mFirstStep : true;
    .....
```

Si le composant de base déclare des sockets, il est nécessaire de déclarer un alias sur l'id de socket pour permettre d'y accéder depuis un composant de plus bas niveau.

```
property alias input : u
socketsLeft: [Socket {id: u;socketId: "u";direction: "input";type: "bool"} ]
```

Gestion des états :

première solution :

Dans **NodeBase** on déclare var statesvar :{"mFirstStep": true; "yOld":0.0};

puis dans **NodeChild** :

```
if (statesvar.mFirstStep){.....}
```

et puisque statesvar est déclaré au niveau supérieur on utilise directement :

```
statesvar.yOld
```

La variable yOld est ajoutée aux statesvar automatiquement.

Dans ce cas la variable état "mFirstStep" est déclarée par tous les composant qui héritent de **NodeBase** mais elle est instanciée dans chaque composant et dans le clic les instances n'ont donc pas la même valeur.

Ce n'est pas ce qu'on appelle une variable static de classe en C++, partagée par toutes les instances. De ce fait, on perd la globalisation de la valeur de mFirstStep.

seconde solution :

déclarer une autre variable état "initialStep" en statesvar au niveau du composant en bout d'héritage (**NodeChild**) .

```
property var statesvar :{"InitialStep": true,"yOld":0};
```

et utiliser les triggers pour gérer les états depuis la propriété globale:

```
function stateSaving()
{
    statesvar.InitialStep=mFirstStep
}
function stateRestored()
{
    mFirstStep=statesvar.InitialStep
}
```

3.6 Composant a nombre de ports variables

Il est possible de créer un composant dont le nombre de port est variable.

Le nombre de port est ajustable par un menu «bouton droit» sur le composant.

La suppression d'un port est effectué par «bouton droit» sur la socket



La position du groupe de port est : **left, right, top, bottom**

Pour des ports appariés: **leftright** et **topbottom** avec la logique d'exclusivité suivante :

leftright ou (left et/ou right)

topbottom ou (top et/ou bottom)

Le nombre de ports localisés est déclaré comme une propriété du composant :

```
property var ports :{"left" :2}
```

Les attributs des ports sont déclarés dans une boucle lors de la construction du composant :

```
for(var i=0;i<ports.left;i++)  
    socketContainerLeft.setPort("I","input","bool");
```

L'exemple de code suivant implémente une porte ET a entrées variables:

```
import QtQuick 2.0  
import Grapher 1.0  
import QtQuick.Controls 1.1  
Node  
{  
    property var ports :{"left" :2}  
    // ~~~~~ Add sockets dynamically ~~~~~  
    Component.onCompleted:  
    {  
        for(var i=0;i<ports.left;i++)  
            socketContainerLeft.setPort("I","input","bool");  
    }  
    //~~~~~ set output socket ~~~~~  
    socketsRight: [Socket {id: out; socketId: "O";  
                        direction: "output";type:"bool"} ]  
    function compute()  
    {  
        out.value = 1  
        for (var i=0;i< socketsLeft.length;i++)  
            out.value &= socketsLeft[i].value  
    }  
}
```

L'exemple de code suivant implémente une logique de parité dynamique:

```
property var ports :{"leftright":2}  
Component.onCompleted: {  
    for(var i=0;i<ports.leftright;i++)  
    {  
        socketContainerLeft.setPort("I","input","real");  
        socketContainerRight.setPort("O","output","real");  
    }  
}
```