Master SESI [M1]
UE MOBJ – Examen

Jean-Paul CHAPUT

Janvier 2015

Durée : 2 heures – Tous documents autorisés

Le barème est donné à titre indicatif et peut être modifié par le correcteur

Écrivez lisiblement, un texte difficilement déchiffrable sera toujours considéré comme faux

Support de Composants Paramètrables

On désire enrichir la structure de données `Netlist` de composants paramétrables. Un composant paramétrable est une `Cell` (un modèle) dont on va pouvoir faire varier certaines caractéristiques. Un exemple type est le composant transistor P dont les principales caractéristiques électriques sont la largeur (**W**) et la longueur de (**L**) de canal. On souhaite pouvoir créer des instances de la `Cell` Transistor P avec une valeur différente de (**W**,**L**) pour chacune.

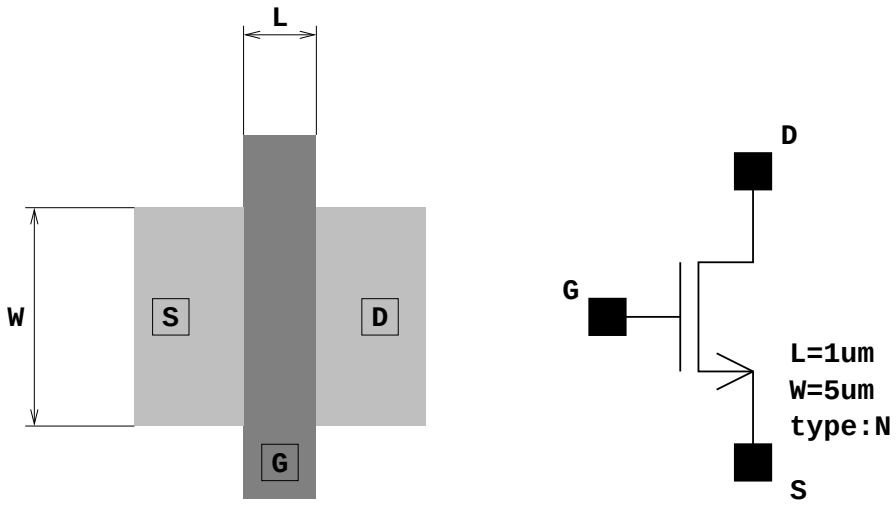

FIGURE 1 – Composant : Transistor N

Les paramètres seront donc présents à la fois dans la `Cell` et dans la classe `Instance` :

— Dans la classe `Cell` ils constitueront la liste des paramètres possibles ainsi que les valeurs par défaut.
— La classe `Instance` contiendra un *duplicata* de la liste du modèle (`Cell`) avec les valeurs spécifiques à cette instance.

Les paramètres sont d'au moins deux genres différents :

— Les paramètres *électriques*, qui donnent une *valeur* a ce paramètre. Par exemple, dans le cas du transistor N, **W** et **L** qui sont des longueurs. Mais cela peut aussi être des Volts, des Ampères, des Farads...
— Les paramètres *structurels*, qui décrivent plutôt une configuration physique du modèle. Dans le cas du transistor, cela est par exemple, son type N ou P. Mais on peut en avoir d'autres comme le type de connexion au substrat...

Pour représenter les différents types de paramètres en C++, nous adoptons l'arborescence de classe décrite 2.

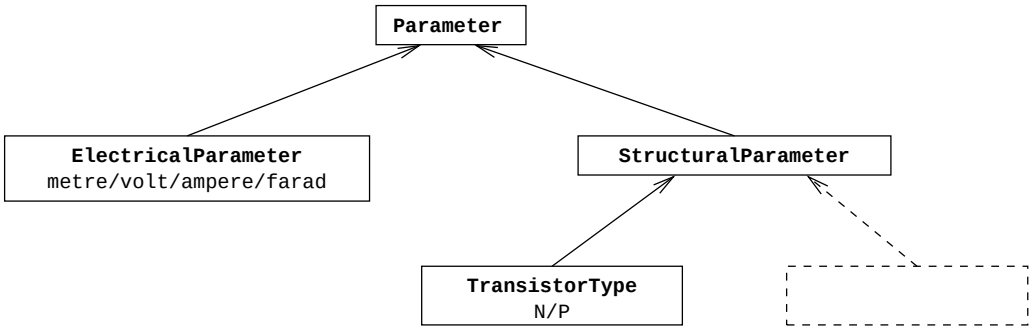

FIGURE 2 – Hiérarchie de Classes Paramètres

Spécification Détaillées des Classes Paramètres

Dans tout le code que vous devez écrire, pour plus de lisibilité, il ne sera pas nécessaire d'utiliser le préfixe de l'espace de nom de la STL `std::`.

Question 1 **5pt**

La classe de base `Parameter`

Attributs	
`name_`	Nom du paramètres (utilisé comme identifiant)
Méthodes	
`Constructeur` `Destructeur`	Prend un unique argument, le nom du paramètre
`getName()` `getType()`	Accesseur de l'attribut `name_` Indiquera si le paramètre est structurel ou électrique
`getString()`	Renvoie une chaîne de caractères (`string`) donnant la valeur du paramètre telle qu'on souhaite la voir dans le visualisateur graphique (i.e. pas sous forme XML)
`toXml()`	Écrit, dans un `ostream` passé en argument l'objet au format XML
`fromXml()`	Lit depuis un lecteur XML (`xmlTextReaderPtr`) un objet parametre et le retourne
`clone()`	Renvoie un *duplicata* de l'objet courant

Comme il s'agit d'une classe de base, elle ne devra pas être instanciable et certaines méthodes ne peuvent être définies à ce niveau. Vous expliquerez brièvement les techniques utilisées pour implanter cela.

Proposer une implantation de la classe `Parameter`.

Question 2 **5pt**

La classe `ElectricalParameter`. Toutes les méthodes héritées de la classe de base ne sont pas répétées ici. C'est a vous de voir lequelles doivent ou non être ré-implantées.

Le constructeur par copie par défaut est-il suffisant pour créer un duplicata d'un objet de cette classe ?

Quel avantage procure la méthode `clone()` par rapport au constructeur par copie ? (indice : penser au constructeur de la classe `Instance`).

Attributs	
`unit_`	Le type électrique du paramètre (Mètre, Volt, Ampère, Farad ou Ohm). On créera un type énuméré ad-hoc pour cet attribut
`value_`	La valeur numérique (`double`) du paramètre
Méthodes	
`Constructeur` `Destructeur`	Prend un unique argument, le nom du paramètre

`getType()`
`getUnit()`
`getValue()`
	Indiquera un paramètre électrique
	Accesseur de l'attribut `unit_`
	Accesseur de l'attribut `value_`

`getString()`	Example : `L:1.0um` ou `W:5.0um`
`setUnit()` `setValue()`	Modificateur de l'attribut `unit_` Modificateur de l'attribut `value_`
`fromXml()`	Lit depuis un lecteur XML (`xmlTextReaderPtr`) un objet parametre et le retourne `ElectricalParameter* fromXml(xmlTextReaderPtr)`

Question 3 **4pt**

La classe `Component` est dérivée de `Cell` et lui ajoute le support des paramètres. Elle va contenir un attribut supplémentaire : un tableau de `Parameter` ainsi que les méthodes de gestion associées.

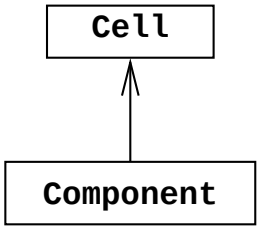

FIGURE 3 – Classe Component

Attributs	
`parameters_`	Le tableau de `Parameter`
Méthodes	
`Constructeur` `Destructeur`	Prend un unique argument, le nom du modèle
`getParameters()` `getParameter()`	Accesseur de l'attribut `parameters_` Recherche un paramètre par son nom, retourne `NULL` s'il n'est pas trouvé
`addParameter()`	Ajoute un nouveau paramètre, en fin de tableau. S'il y a un paramètre de même nom, il est remplacé par le nouveau
`removeParameter()`	Retire un paramètre du tableau. Le paramètre supprimé devra être désalloué. Créer deux surcharges pour cette méthode, une avec un pointeur sur paramètre et l'autre avec le nom du paramètre

Question 4 **6pt**

Modification de la classe `Instance`. On désire que cette classe puisse réprésenter aussi bien des `Cell` que des `Component`. A cet effet on l'enrichi d'un tableau de paramètres. Ce tableau sera vide dans le cas d'une instance de `Cell` et une copie des paramètres dans le cas d'une instance de `Component`.

Dans une `Instance`, les paramètres peuvent ensuite être remplacés par des paramètres identiques, mais avec des valeurs différentes avec la méthode `replace()` (si un paramètre de même nom n'existe pas, la fonction ne fera rien et affichera une erreur).

Attributs	
`parameters_`	Le tableau de `Parameter`
Méthodes	
`Constructeur` `Destructeur`	A mettre à jour A mettre à jour
`getParameters()` `getParameter()`	Accesseur de l'attribut `parameters_` Recherche un paramètre par son nom, retourne `NULL` s'il n'est pas trouvé
`replace()`	Remplace un paramètre existant par un autre
`fromXml()`	Lit depuis un lecteur XML (`xmlTextReaderPtr`) un objet `Instance` et le retourne

On modifiera la fonction `fromXml()` pour lire le format suivant. Pour les balises `Term` et `ElectricalParameter` on utilisera les fonctions des classes associées. On sera attentif à l'allocation et a la désallocation des paramètres.

```
<Instance name="T1" masterCell="TransistorN">
  <Terms>
    <Term name="S" net="vss"/>
    <Term name="D" net="o"/>
    <Term name="G" net="i"/>
  </Terms>
  <Parameters>
    <ElectricalParameter name="W" unit="Meter" value="5.0e-6"/>
  </Parameters>
</Instance>
```

Corrigé Examen UE MOBJ – Janvier 2015

Réponse à la question 1 **5pt**

Barème : Blocage instantiation/abstraite **1.0pt**
Méthodes virtuelles pures **0.5pt**
Attribut `name_` **0.5pt**
Implantation du constructeur **0.5pt**
Implantation du destructeur **0.5pt**
Implantation de `getName()` **0.5pt**
Implantation de `toXml()` **0.5pt**
Implantation de `fromXml()` **1.0pt**

Pour que la classe ne soit pas instantiable, il faut qu'elle soit abstraite. C'est à dire qu'elle contienne, au moins une méthode virtuelle pure. Ce sera bien le cas, car on ne peut implenter à ce niveau les méthodes `clone()`, `getType()`, `getString()` et `toXml()`.

Déclaration de la classe de base `Parameter`

```
class Parameter {
  public:
    enum Type { Structural=1, Electrical=2 };
  public:
                                Parameter  ( const std::string& );
    virtual                    ~Parameter  ();
    virtual Parameter*          clone      () const = 0;
    inline  const std::string&  getName    () const;
    virtual Type                getType    () const = 0;
    virtual std::string         getString  () const = 0;
    virtual void                toXml      ( std::ostream& ) = 0;
    static  Parameter*          fromXml    ( xmlTextReaderPtr );
  private:
    std::string  name_;
};
```

Définition des méthodes de la classe de base `Parameter`

```
Parameter::Parameter ( const string& name )
  : name_(name)
{ }

Parameter::~Parameter ()
{ }

Parameter* Parameter::fromXml ( xmlTextReaderPtr reader )
{
// Factory-like method.
  const xmlChar* ttypeTag  = xmlTextReaderConstString
                               ( reader, (const xmlChar*)"TransistorType"  );
  const xmlChar* elecTag   = xmlTextReaderConstString
                               ( reader, (const xmlChar*)"ElectricalParameter" );
  const xmlChar* nodeName  = xmlTextReaderConstLocalName     ( reader );
  int            lineNo    = xmlTextReaderGetParserLineNumber( reader );
  Parameter*     parameter = NULL;

  if (ttypeTag == nodeName) parameter =      TransistorType::fromXml( reader );
  if (elecTag  == nodeName) parameter = ElectricalParameter::fromXml( reader );

  if (parameter == NULL)
    cerr << "[ERROR] Unknown or misplaced tag <" << nodeName
         << "> (line:" << lineNo << ")." << endl;

  return parameter;
}
```

Réponse à la question 2 **5pt**

Barème :
- Constructeur par copie suffisant **0.5pt**
- Explication de la méthode `clone()` **1.0pt**
- Présence des méthodes virtuelles **0.5pt**
- Attribut `unit_` & `value_` **0.25pt**
- Implantation du constructeur **0.5pt**
- Implantation du destructeur **0.25pt**
- Implantation de `getString()` **0.5pt**
- Implantation de `toXml()` **0.5pt**
- Implantation de `fromXml()` **1.0pt**

Cette classe ne contenant pas d'attributs alloués dynamiquement, ou de pointeurs vers des objets tiers, le constructeur par copie synthétisé par défaut est suffisant.

Le constructeur par copie, n'est pas (et ne peut pas être) une méthode virtuelle. Donc pour copier des objets de type `Parameter`, il faudrait retrouver le type exact avant de pouvoir faire la copie (par exemple avec un `dynamic_cast<>`). La `clone()` permet d'encapsuler le constructeur par copie dans une méthode virtuelle, ce qui permet de dupliquer de façon transparente un objet au travers d'un pointeur de type `Parameter`.

Déclaration de la classe `ElectricalParameter`

```
class ElectricalParameter : public Parameter {
  public:
    enum Unit { Meter=1, Ohm=2, Farad=3, Volt=4, Ampere=5 };
  public:
                                ElectricalParameter ( const std::string& );
    virtual                    ~ElectricalParameter ();
    virtual ElectricalParameter* clone               () const;
    virtual Parameter::Type      getType             () const;
    virtual std::string          getString           () const;
    inline  Unit                 getUnit             () const;
    inline  double               getValue            () const;
    inline  void                 setUnit             ( Unit );
    inline  void                 setValue            ( double );
    virtual void                 toXml               ( std::ostream& );
    static  ElectricalParameter* fromXml             ( xmlTextReaderPtr );
  private:
    Unit    unit_;
    double  value_;
};
```

Fonctions `inline` de la classe `ElectricalParameter`

```
inline  ElectricalParameter::Unit   ElectricalParameter::getUnit  () const { return unit_; }
inline  double                      ElectricalParameter::getValue () const { return value_; }
inline  void                        ElectricalParameter::setUnit  ( Unit unit )
                                                                  { unit_ = unit; }
inline  void                        ElectricalParameter::setValue ( double value )
                                                                  { value_ = value; }
```

Définition des méthodes de la classe `ElectricalParameter`

```
ElectricalParameter::ElectricalParameter ( const string& name )
  : Parameter(name), unit_(Meter), value_(0.0)
{ }

ElectricalParameter::~ElectricalParameter ()
{ }

ElectricalParameter* ElectricalParameter::clone () const
{ return new ElectricalParameter ( *this ); }

Parameter::Type  ElectricalParameter::getType () const
{ return Parameter::Electrical; }

string  ElectricalParameter::getString () const
{
  ostringstream s;
  s << getName() << "=";
  switch ( unit_ ) {
    case Meter:  s << fixed << setprecision(1) << (value_*1e6) << "um"  ; break;
    case Ohm:    s << value_ << "Ohm"; break;
    case Farad:  s << value_ << "F"  ; break;
    case Volt:   s << value_ << "V"  ; break;
    case Ampere: s << value_ << "A"  ; break;
  }
  return s.str();
}

void  ElectricalParameter::toXml ( ostream& stream )
{
  stream << indent << "<ElectricalParameter␣name=\"" << getName() << "\"␣";
  switch ( unit_ ) {
    case Meter:  stream << "unit=\"Meter\"" ; break;
    case Ohm:    stream << "unit=\"Ohm\""   ; break;
    case Farad:  stream << "unit=\"Farad\"" ; break;
    case Volt:   stream << "unit=\"Volt\""  ; break;
    case Ampere: stream << "unit=\"Ampere\""; break;
  }
  stream << "␣value=\"" << value_ << "/>\n";
}

ElectricalParameter* ElectricalParameter::fromXml ( xmlTextReaderPtr reader )
{
  double value    = 0.0;
  string elecName = xmlCharToString
                    ( xmlTextReaderGetAttribute( reader, (const xmlChar*)"name" ) );
  string elecUnit = xmlCharToString
                    ( xmlTextReaderGetAttribute( reader, (const xmlChar*)"unit" ) );

  if (elecName.empty()) {
    int  lineNo = xmlTextReaderGetParserLineNumber( reader );
    cerr << "[ERROR]␣\"name\"␣attribute␣missing␣in␣<ElectricalParameter>␣tag␣(line:"
         << lineNo << ")." << endl;
    return NULL;
  }

  ElectricalParameter* elec = new ElectricalParameter ( elecName );

  if      (elecUnit == "Meter" ) elec->setUnit( Meter  );
  else if (elecUnit == "Ohm"   ) elec->setUnit( Ohm    );
  else if (elecUnit == "Farad" ) elec->setUnit( Farad  );
  else if (elecUnit == "Volt"  ) elec->setUnit( Volt   );
  else if (elecUnit == "Ampere") elec->setUnit( Ampere );

  xmlGetDoubleAttribute( reader, "value", value );
  elec->setValue( value );

  return elec;
}
```

Réponse à la question 3 **4pt**

Barème : Attribut `parameters_` et type **0.5pt**
Implantation du constructeur, héritage **0.5pt**
Implantation du destructeur **0.25pt**
Implantation de `getParameters()` **0.25pt**
Implantation de `getParameter()` **0.5pt**
Implantation de `addParameter()` **0.5pt**
Implantation de `removeParameter()` **1.0pt**

Déclaration de la classe `Component`

```
class Component : public Cell {
  public:
                                          Component       ( const std::string& );
    virtual                              ~Component       ();
    inline  const std::vector<Parameter*>& getParameters   () const;
                  Parameter*              getParameter    ( const std::string& ) const;
                  void                    addParameter    ( Parameter* );
                  void                    removeParameter ( Parameter* );
                  void                    removeParameter ( const std::string& name );
  private:
    std::vector<Parameter*>  parameters_;
};

inline  const std::vector<Parameter*>& Component::getParameters () const
                                       { return parameters_; }
```

Définition des méthodes de la classe `Component`

```
Component::Component ( const string& name )
  : Cell(name), parameters_()
{ }

Component::~Component ()
{
  while ( not parameters_.empty() )
    delete *parameters_.begin();
}

Parameter* Component::getParameter ( const string& name ) const
{
  for ( size_t i=0 ; i<parameters_.size() ; ++i ) {
    if (parameters_[i]->getName() == name) return parameters_[i];
  }
  return NULL;
}

void  Component::addParameter ( Parameter* param )
{
  for ( size_t i=0 ; i<parameters_.size() ; ++i ) {
    if (parameters_[i]->getName() == param->getName()) {
      cerr << "[ERROR] Duplicate parameter <" << param->getName()
           << "> in <" << getName() << "> (overriding)." << endl;
      delete parameters_[i];
      parameters_[i] = param;
      return;
    }
  }

  parameters_.push_back( param );
}

void  Component::removeParameter ( Parameter* param )
{
  vector<Parameter*>::iterator iparam = parameters_.begin();
  for ( ; iparam != parameters_.end() ; ++iparam ) {
    if (*iparam == param) parameters_.erase( iparam );
  }
}

void  Component::removeParameter ( const string& name )
{
  vector<Parameter*>::iterator iparam = parameters_.begin();
  for ( ; iparam != parameters_.end() ; ++iparam ) {
    if ((*iparam)->getName() == name) parameters_.erase( iparam );
  }
}
```

Réponse à la question 4 **6pt**

Barème : Attribut `parameters_` et type **0.5pt**
Modification du constructeur, héritage **1.0pt**
Implantation du destructeur **1.0pt**
Implantation de `getParameters()` **0.5pt**
Implantation de `getParameter()` **0.5pt**
Implantation de `replace()` **1.0pt**
Implantation de `fromXml()` **1.5pt**

Déclaration de la classe `Instance`

```
class Instance {
  public:
    // Methodes modifiees.
                                          Instance      ( Cell* owner, Cell* model, const std::string& );
                                         ~Instance      ();
    inline const std::vector<Parameter*>& getParameters () const;
           Parameter*                     getParameter  ( const std::string& ) const;
           bool                           replace       ( Parameter* );
    static Instance*                      fromXml       ( Cell*, xmlTextReaderPtr );
  private:
    // Attributs additionnel.
    std::vector<Parameter*>  parameters_;
};

inline const std::vector<Parameter*>& Instance::getParameters () const { return parameters_; }
```

Définition des méthodes additionnelles de la classe `Instance` (1)

```
Instance::Instance (  Cell* owner, Cell* model, const string& name )
  : // Autres attributs...
  , parameters_()
{
  const vector<Term*>& terms = masterCell_->getTerms();
  for ( vector<Term*>::const_iterator iterm=terms.begin()
      ; iterm != terms.end() ; ++iterm ) {
    new Term( this, (*iterm)->getName(), (*iterm)->getDirection() );
  }
  owner_->add( this );

  Component* comp = dynamic_cast<Component*>( model );
  if (comp) {
    const vector<Parameter*> parameters = comp->getParameters();
    for ( size_t i=0 ; i<parameters.size() ; ++i )
      parameters_.push_back( parameters[i]->clone() );
  }
}

Instance::~Instance ()
{
  for ( vector<Term*>::const_iterator iterm=terms_.begin()
      ; iterm != terms_.end() ; ++iterm ) {
    delete *iterm;
  }
  owner_->remove( this );

  while ( not parameters_.empty() )
    delete *parameters_.begin();
}
```

Définition des méthodes additionnelles de la classe `Instance` (2)

```
Parameter* Instance::getParameter ( const string& name ) const
{
  for ( size_t i=0 ; i<parameters_.size() ; ++i ) {
    if (parameters_[i]->getName() == name) return parameters_[i];
  }
  return NULL;
}

bool  Instance::replace ( Parameter* replacement )
{
  for ( size_t i=0 ; i<parameters_.size() ; ++i ) {
    if (parameters_[i]->getName() == replacement->getName()) {
      delete parameters_[i];
      parameters_[i] = replacement;
      cerr << "␣␣␣␣Replacing␣parameter␣" << replacement->getName() << endl;
      return true;
    }
  }
  cerr << "[ERROR]␣No␣parameter␣<" << replacement->getName()
       << ">␣in␣model␣of␣instance␣<" << getName() << ">." << endl;
  return false;
}
```

Définition des méthodes additionnelles de la classe `Instance` (3)

```
Instance* Instance::fromXml ( Cell* cell, xmlTextReaderPtr reader )
{
  cerr << "Instance::fromXml()" << endl;

  const xmlChar* instanceTag   = xmlTextReaderConstString( reader, (const xmlChar*)"Instance" );
  const xmlChar* termsTag      = xmlTextReaderConstString( reader, (const xmlChar*)"Terms" );
  const xmlChar* parametersTag = xmlTextReaderConstString( reader, (const xmlChar*)"Parameters" );
  const xmlChar* nodeName      = NULL;
  int            lineNo        = -1;

  enum State { BeginInstance = 1
             , EndInstance
             , BeginTerms
             , EndTerms
             , BeginParameters
             , EndParameters
             , ParseError
             };

  Cell*     masterCell = NULL;
  Instance* instance   = NULL;
  State     state      = BeginInstance;
  int       status     = 1;

  while ( true ) {
    if (state  != BeginInstance) status = xmlTextReaderRead(reader);
    if (status != 1) {
      if (status != 0) {
        cerr << "[ERROR] Instance::fromXml(): Unexpected termination of the XML parser." << endl;
      }
      break;
    }

    switch ( xmlTextReaderNodeType(reader) ) {
      case XML_READER_TYPE_COMMENT:
      case XML_READER_TYPE_WHITESPACE:
      case XML_READER_TYPE_SIGNIFICANT_WHITESPACE:
        continue;
    }

    nodeName = xmlTextReaderConstLocalName     ( reader );
    lineNo   = xmlTextReaderGetParserLineNumber( reader );

    // Continue page suivante.
```

```
switch ( state ) {
  case BeginInstance:
    if (nodeName == instanceTag) {
      cerr << "␣␣Instance␣<Instance>␣read" << endl;
      string instanceName   = xmlCharToString
               ( xmlTextReaderGetAttribute( reader, (const xmlChar*)"name" ) );
      string masterCellName = xmlCharToString
               ( xmlTextReaderGetAttribute( reader, (const xmlChar*)"masterCell" ) );

      if (instanceName.empty() or masterCellName.empty()) {
        cerr << "[ERROR]␣\"name\"␣or␣\"masterCell\"␣attribute␣missing␣in␣<Instance>␣tag␣(line:"
             << lineNo << ")." << endl;
        return NULL;
      }

      masterCell = Cell::find( masterCellName );
      if (not masterCell) {
        cerr << "[ERROR]␣Instance␣<" << instanceName << ">␣with␣unknown␣master␣Cell␣<"
             << masterCellName << ">␣(line:" << lineNo << ")." << endl;
        return NULL;
      }

      instance = new Instance ( cell, masterCell, instanceName );
      state    = BeginTerms;
      cerr << "␣␣Instance␣<" << instanceName << ">␣created" << endl;
      continue;
    }
    break;
  case BeginTerms:
    if (   (nodeName == termsTag)
       and (xmlTextReaderNodeType(reader) == XML_READER_TYPE_ELEMENT) ) {
      cerr << "␣␣Instance␣<Terms>␣read" << endl;
      state = EndTerms;
      continue;
    }
    break;
  case EndTerms:
    if (   (nodeName == termsTag)
       and (xmlTextReaderNodeType(reader) == XML_READER_TYPE_END_ELEMENT) ) {
      state = BeginParameters;
      cerr << "␣␣Instance␣</Terms>␣read" << endl;
      continue;
    } else {
      if (Term::fromXml(instance,reader)) continue;
    }
    break;
  case BeginParameters:
    if ( (nodeName == parametersTag)
        and (xmlTextReaderNodeType(reader) == XML_READER_TYPE_ELEMENT) ) {
      state = EndParameters;
      cerr << "␣␣Instance␣<Parameters>␣read" << endl;
      continue;
    }

// Continue page suivante.
```

```
      case EndParameters:
        if (   (nodeName == parametersTag)
           and (xmlTextReaderNodeType(reader) == XML_READER_TYPE_END_ELEMENT) ) {
          state = EndInstance;
          continue;
        } else {
          Parameter* parameter = Parameter::fromXml( reader );
          if (parameter) {
            Component* comp = dynamic_cast<Component*>( masterCell );
            if (comp) {
              if (instance->replace(parameter)) continue;
            }
            delete parameter;
            continue;
          }
          break;
        }
      case EndInstance:
        if (nodeName == instanceTag) {
          if (xmlTextReaderNodeType(reader) != XML_READER_TYPE_END_ELEMENT)
            cerr << "[ERROR] No closing Instance tag (line:" << lineNo << ")." << endl;
          return instance;
        }
      default:
        break;
    }
    break;
  }

  cerr << "[ERROR] Instance::fromXml(): Unknown or misplaced tag <" << nodeName
       << "> (line:" << lineNo << ")." << endl;
  return NULL;
}
```