UE MOBJ [4L103]

Jean-Paul CHAPUT
Jean-Paul.Chaput@lip6.fr

SESI

2020-2021

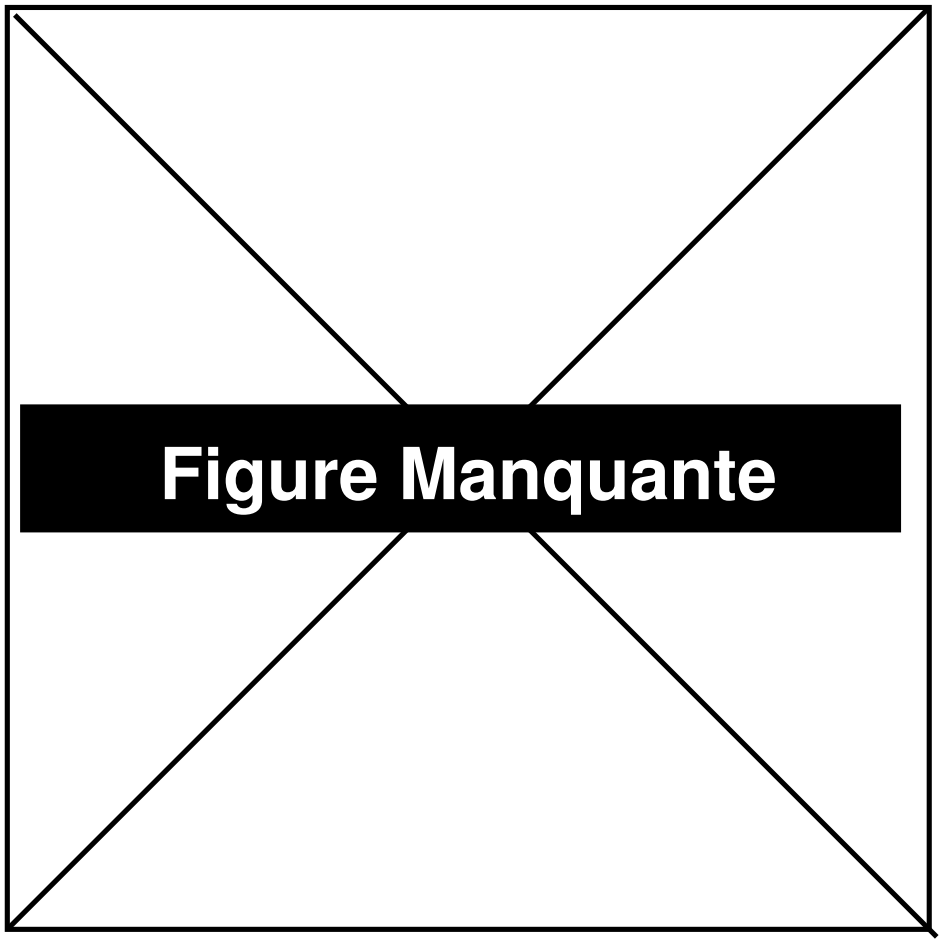
Table des matières

Concepts Présentés au Cinquième Cours

1. Concept de `netlist`.
2. Structure de données pour représenter une `netlist`.

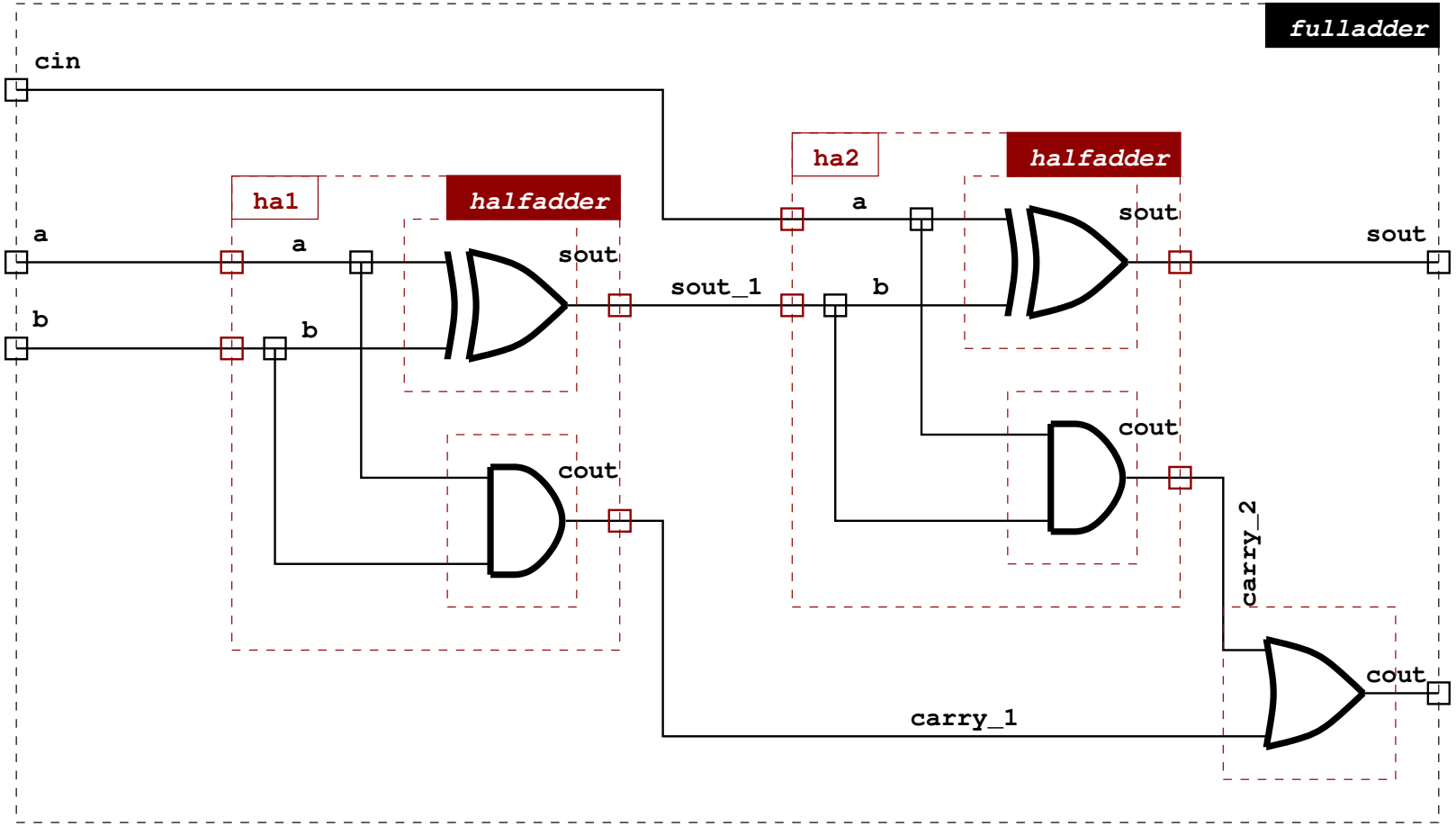
Pas de notes pour ce transparent.

IV Architecture Logicielle



- Les trois composants nécessaires:
    1. La *Structure de Données* (attention à la confusion avec une Base de Données). Constitue le coeur du programme, c'est la représentation des données dans la mémoire de la machine, en C++.
    2. Les *parseur/driver* XML (`libxml2`, en C), assurent la communication avec l'extérieur, à savoir, le système de fichiers. Ils permettent de conserver l'information entre deux exécutions du programme.
    3. *L'interface graphique* (en Qt), affiche les données sous forme d'un schéma. Qt étant lui-même écrit en C++, il est donc nécessaire, pour pouvoir l'utiliser, de maîtriser ce langage.

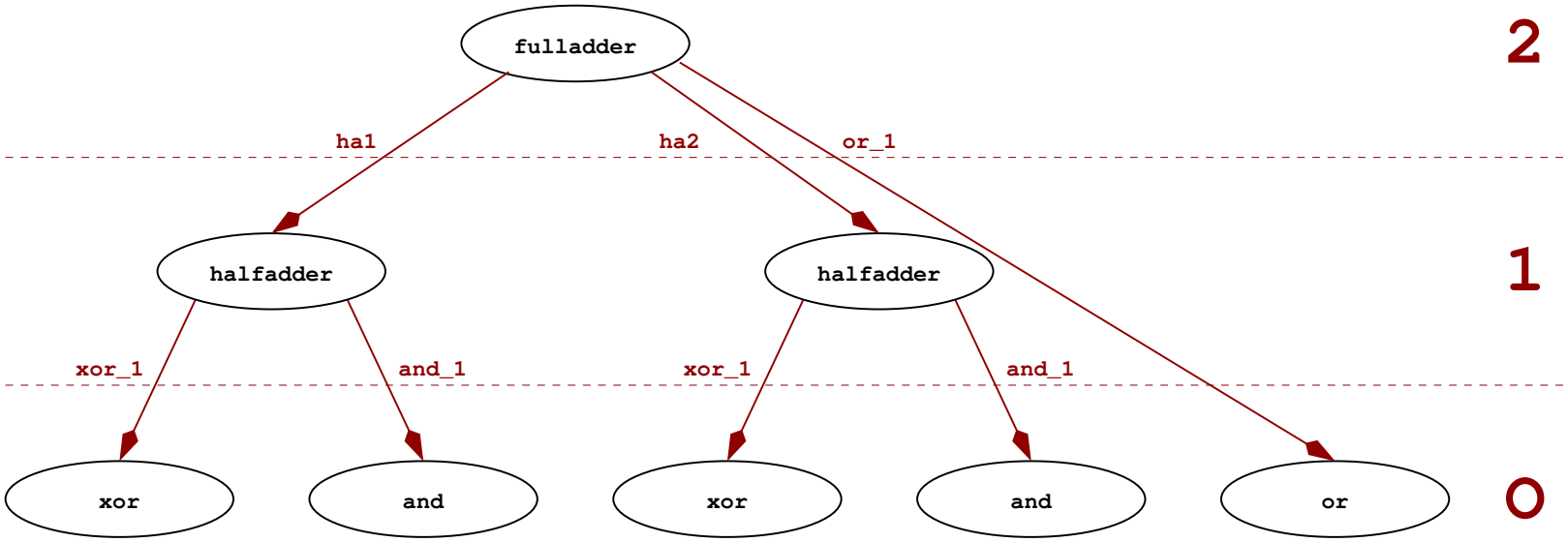
IV Description Complète d'un Circuit



Un élément de base d'un additionneur.

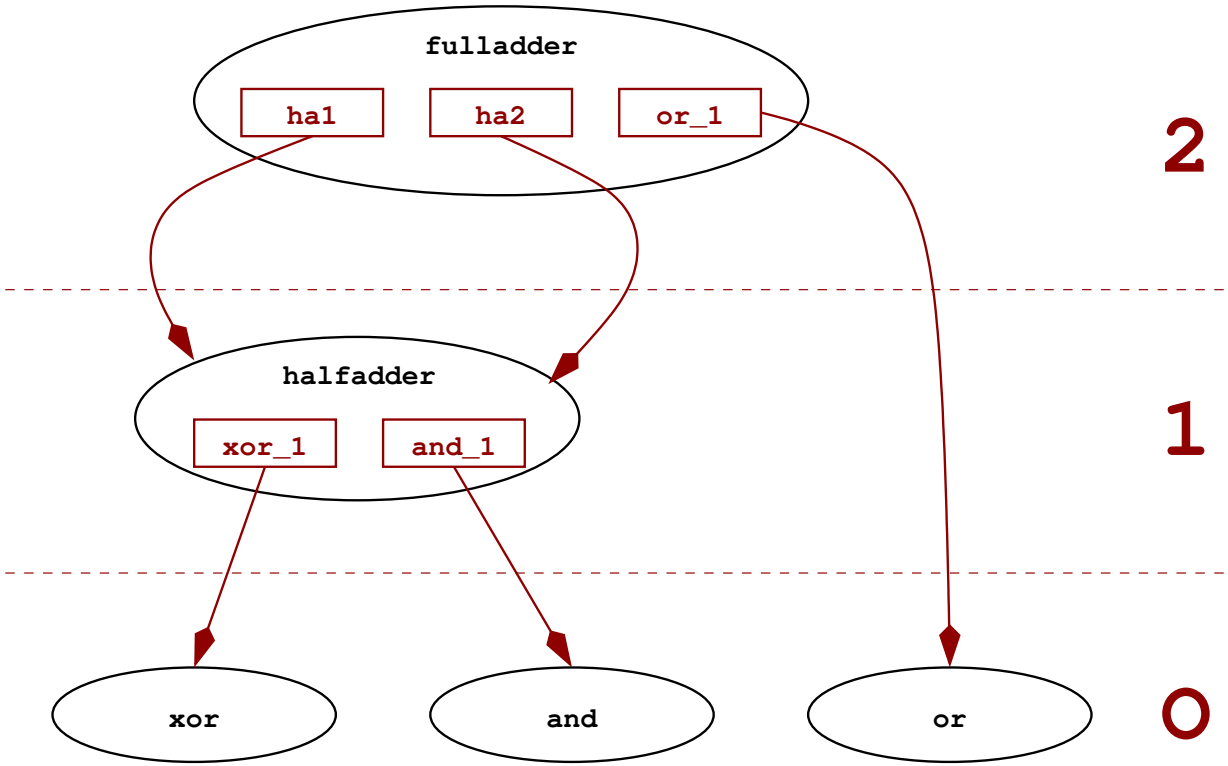
- Pour faire un visualisateur, il faut avoir quelque chose à visualiser. C’est même le plus important...
- Exemple de description complète d’un circuit.
- Description «à plat», description «hiérarchique».
- Décomposition de n’importe quelle fonction logique en un assemblage de portes élémentaires.

IV.1 Graphe de la Hiérarchie Dépliée



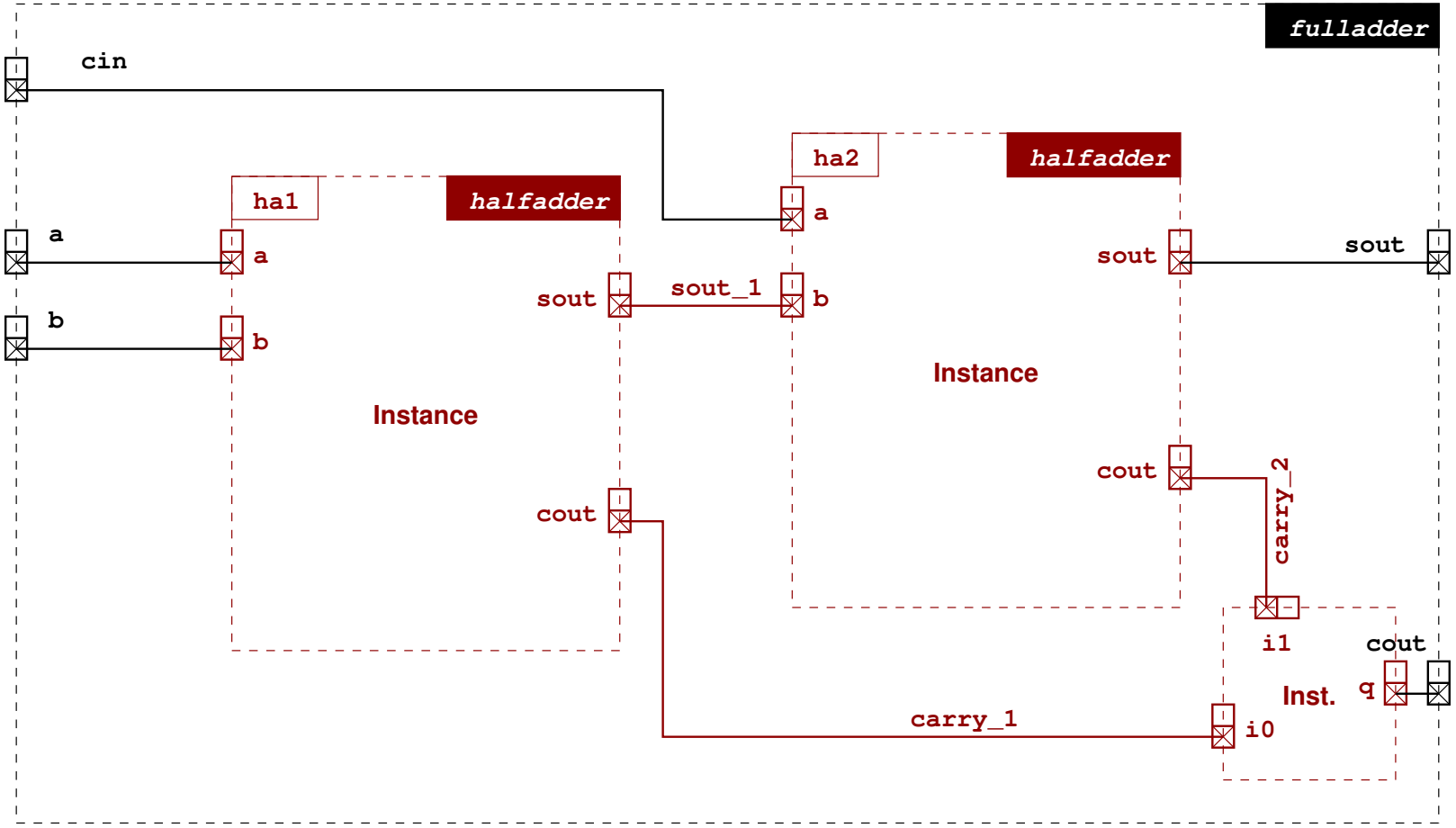
- Les graphes sont des structures mathématiques récurrentes dans le domaine de la CAO-VLSI.
- Quant on introduit un graphe, il est important de définir ce que représentent les noeuds et les arcs.
- Dans le graphe de représentation de la hiérarchie, un noeud représente un modèle.
- Un arc représente la relation «possède une instance du modèle».
- Définition de la profondeur de la hiérarchie: longueur du plus long chemin par lequel on peut atteindre le niveau zéro.
- Le niveau zéro de la hiérarchie est celui des modèles ne comportant aucune instance.

IV.1 Graphe de la Hiérarchie Repliée



- Pour des raisons d'encombrement mémoire, on n'utilise pas l'arbre d'instanciation déplié, mais le graphe replié.
- Les modèles sont mis en commun dans l'arbre. Dans notre exemple cela représente un gain de 3 modèles.
- En pratique, on peut atteindre des taux de compression de la taille mémoire supérieur à 10.
- *In fine* le taux de compression dépend du nombre d'instances dans une hiérarchie.

IV.1 Description d'un Niveau Hiérarchique



- Description «épurée» (réelle) d'un niveau hiérarchique.
- Nature de cette description: hypergraphe.
- Dans un graphe simple, une arête relie deux sommets (noeuds) entre eux. Dans un hypergraphe, une hyper-arête peut relier un nombre quelconque de noeuds entre eux.
- La différence entre un graphe et un hypergraphe se joue sur la nature des arêtes.
- Dans le cas d'une `netlist`, les noeuds sont les instances ou les terminaux du modèle et les hyper-arêtes les signaux. Ici, c'est un mauvais exemple, car tous les signaux n'ont que deux connexions…

IV.2 Élements Constitutifs d'une `Netlist`

- Un modèle, celui pour lequel nous décrivons la `Netlist`.
- Des instances.
- Les terminaux *externes*, accrochés au modèle.
- Les terminaux *internes*, accrochés à une instance.
- Des nets (signaux).

- Par abus de language, quand on parle de `Netlist`, c'est toujours en référence à un modèle déterminé.
- Une `Netlist` ne décrit qu'un niveau de la hiérarchie.

IV.3 Programmation Modulaire

- La cellule (modèle): `Cell`.
- Les terminaux: `Term`.
- Les instances: `Instance`.
- Les nets: `Net`.
- Un objet `Node`, qui va servir à stocker la position des terminaux.

- Introduction des différents objets de la base de données.
- L'intérêt d'une classe `Node` ne pourra être justifié que plus loin.
- Génie Logiciel: lorsque l'on commence à avoir beaucoup de classes on écrit chacune dans un couple de fichiers (`.h`, `.cpp`).
- Dans le cadre de la programmation modulaire, on écrit chaque classe dans un «module» (ou unité de compilation en français) qui a vocation à être compilé séparément. Chaque module est composé d'une paire (`.h`, `.cpp`).
- Un des objectifs de la programmation modulaire est d'augmenter la vitesse de compilation. Un ensemble de modules se compile bien plus rapidement qu'un unique `.cpp` qui contiendrait tout le programme (bien que ce soit possible).

IV.3 Programmation Modulaire – Double Inclusion

```
// Term.h
#ifndef  SCHEMATIC_TERM_H
#define  SCHEMATIC_TERM_H
class Term {
    // ...
};
#endif
```

```
// Instance.h
#ifndef  SCHEMATIC_INSTANCE_H
#define  SCHEMATIC_INSTANCE_H
#include "Term.h"

class Instance {
    // ...
};
#endif
```

```
// Cell.h
#ifndef  SCHEMATIC_CELL_H
#define  SCHEMATIC_CELL_H
#include "Term.h"
#include "Instance.h"
class Cell {
    // ...
};
#endif
```

- Permet de s’assurer que quelque soit la séquence d’inclusion un même `.h` n’est pas inclus deux fois.

IV.3 Programmation Modulaire – *Forward* Déclaration

```
// Instance.h
#ifndef  SCHEMATIC_INSTANCE_H
#define  SCHEMATIC_INSTANCE_H
#include "Cell.h"
class Instance {
  // ...
  Cell* getCell() const;
};

#endif
```

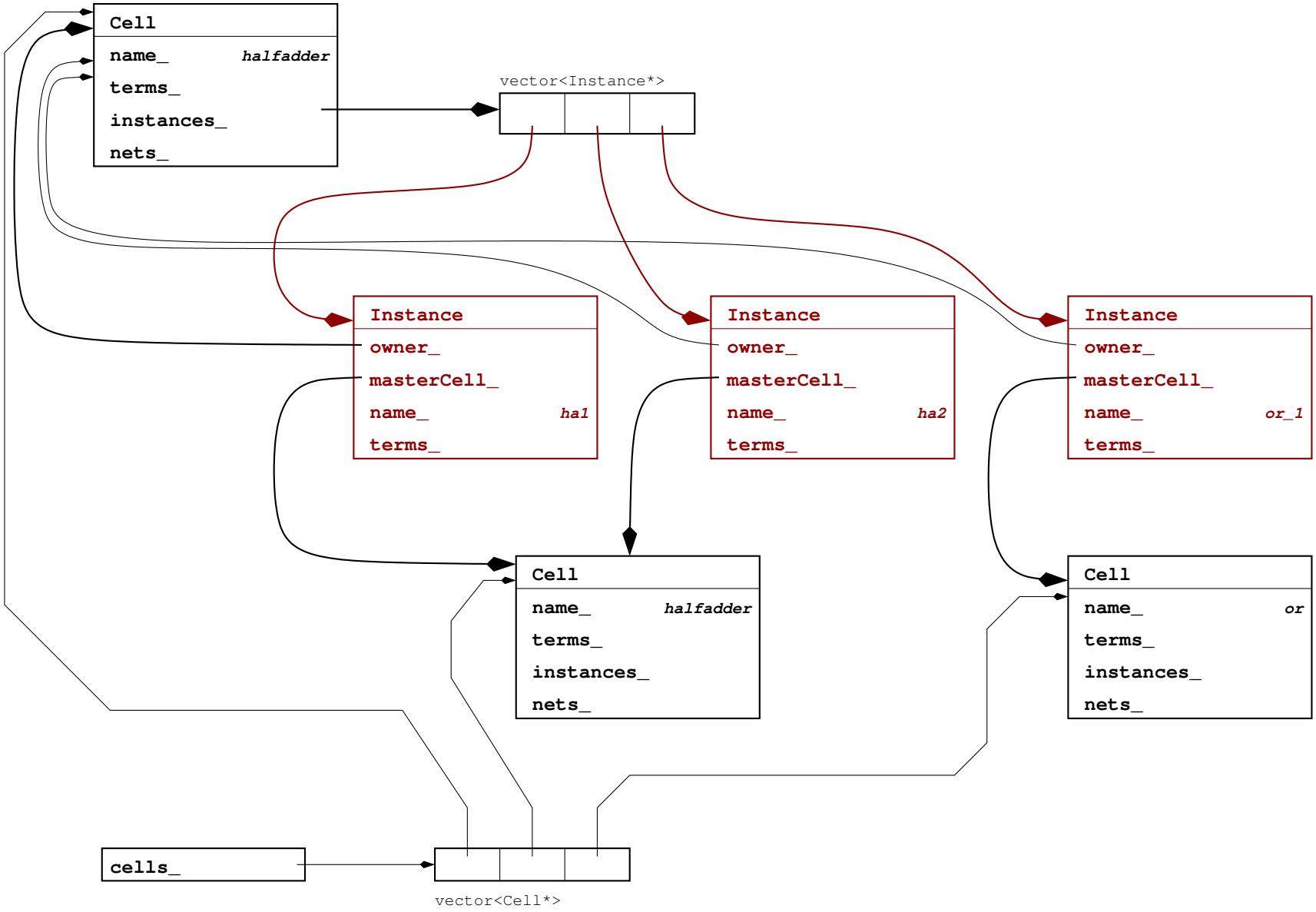
```
// Cell.h
#ifndef  SCHEMATIC_CELL_H
#define  SCHEMATIC_CELL_H
#include "Instance.h"
class Cell {
  // ...
  Instance* getInstance
    (const std::string&) const;
};
#endif
```

```
// Cell.cpp
#include "Instance.h"
#include "Cell.h"

Instance* Cell::getInstance(const std::string&) const
{ }
#endif
```

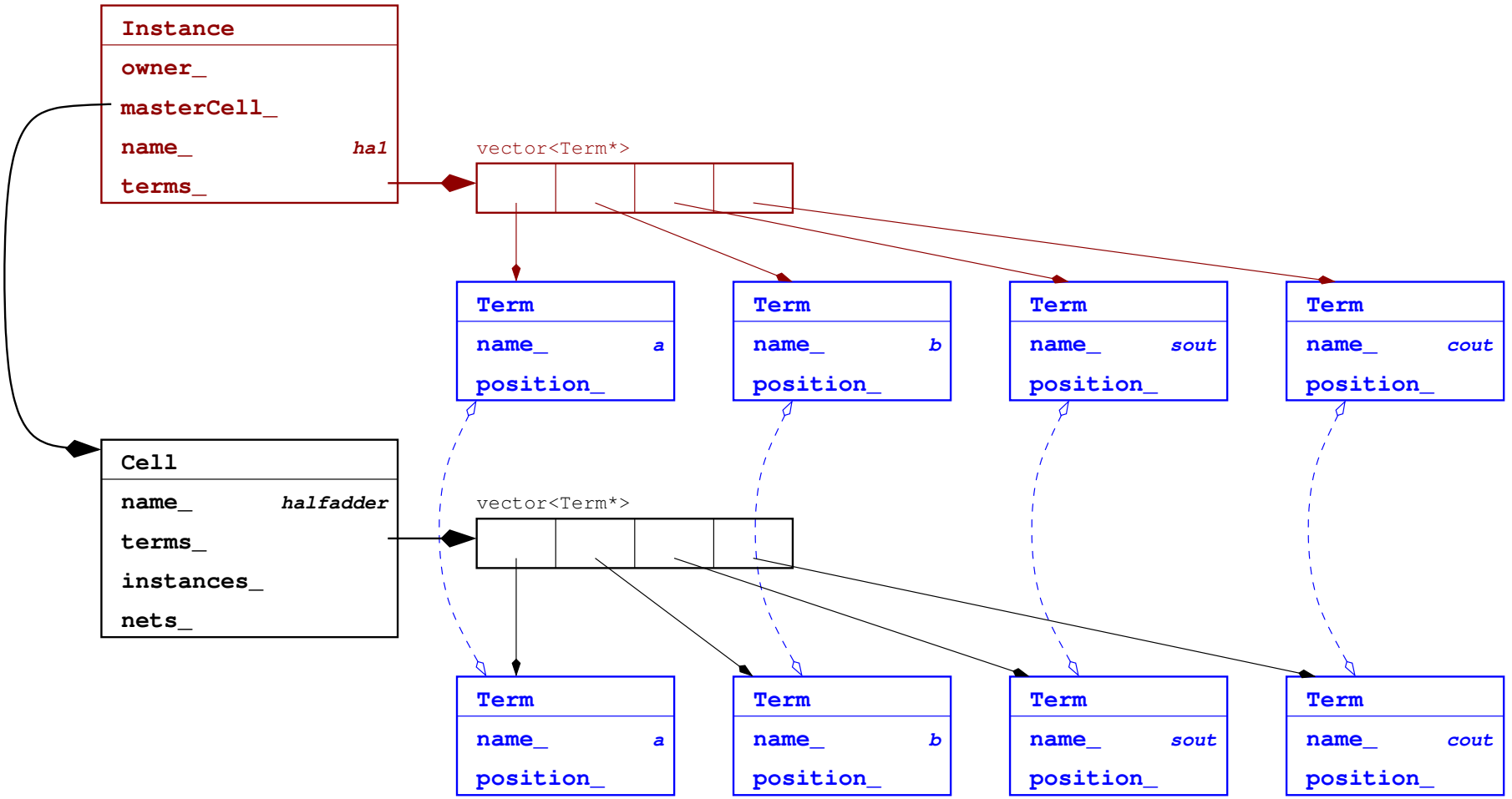
- Séquence d'inclusion: Cell.cpp inclut Instance.h qui inclut Cell.h qui a besoin de Instance, qui n'est pas encore défini...
- Pour compiler `Cell`, on voit que nous avons besoin de `Instance`. La première idée consiste à inclure leur `.h`, mais on va rapidement créer une boucle.
- La solution consite à utiliser une *forward* déclaration. On peut alors utiliser (dans le `.h`) des pointeurs et des référence sur la classe. Et c'est généralement suffisant.
- Il faut, au minimum, transformer l'include de `Cell.h` dans `Instance.h` en forward déclaration. On transforme aussi `Instance.h` en forward déclaration dans `Cell.h`. Cela permet, potentiellement d'accélérer la compilation (moins d'objets à gérer par chaque module).
- Remarque: détailler le module `Cell` complet pour voir où sont faits les includes.
- Autre intérêt pour les modules incluant `Cell` qui n'ont pas forcément besoin des autres.

IV.4 Relations entre `Cell` et `Instance` (1/2)



- Le dessin précédent est la représentation qu'un humain a d'une `Netlist`. Mais ce n'est pas ce qu'une machine peut comprendre.
- La représentation machine, ou encore, comment l'information est organisée en mémoire, c'est la relation entre les objets.
- Les relations entre objets (graphe) se font principalement par pointeurs.
- Le programme ne peut se promener dans la structure de données qu'au travers des chemins définis par les pointeurs.
- L'instance est un *résumé* d'un modèle dans un autre modèle. Elle duplique partiellement les informations de ce modèle. Elle est associée de façon unique à un modèle.
- Noter la variable `cells_` qui contient la liste de tous les modèles déjà créés. Car il faut bien les retrouver d'une façon ou d'une autre.

IV.4 Relations entre `Cell` et `Instance` (2/2)



- Informations dupliquées du modèle dans une instance: les terminaux.
- La connexion des terminaux d'une instance est spécifique à celle-ci. Elle a **les mêmes terminaux** que le modèle, mais connectés différemment.
- Il est clair que `ha1` et `ha2`, instances de `halfadder` ne sont pas connectés de façon identique.

IV.5 La Classe Cell – Attributs

```
class Cell {
  private:
    static  std::vector<Cell*>       cells_;
            std::string              name_;
            std::vector<Term*>       terms_;
            std::vector<Instance*>   instances_;
            std::vector<Net*>        nets_;
            unsigned int             maxNetIds_;
};
```

- C’est la `Cell` qui contient la description de la `netlist`.
- Un nom de modèle.
- Une liste de terminaux (externes) comme pour les instances. C’est une liste qui est dupliquée lors de la création d’une instance.
- Une liste d’instances.
- Une liste de nets (la `netlist`).
- `maxNetIds_` génère les identifiants uniques pour les signaux. On ne fait pas confiance aux noms, car un signal peut potentiellement en avoir plusieurs. On préfère des identifiants numériques que l’on garantit uniques.
- `cells_` est mis en attribut statique, simplement pour ne pas laisser trainer une variable isolée…

IV.5 La Classe Cell – Accesseurs Divers

```
class Cell {
                    Cell          ( const std::string& );
                   ~Cell          ();
  const std::string&              getName       () const;
  const std::vector<Instance*>&   getInstances  () const;
  const std::vector<Term*>&       getTerms      () const;
  const std::vector<Net*>&        getNets       () const;
        Instance* getInstance     ( const std::string& ) const;
        Term*     getTerm         ( const std::string& ) const;
        Net*      getNet          ( const std::string& ) const;
};
```

- Ces différentes méthodes permettent d'accéder/rechercher individuellement un objet, ou bien d'obtenir l'ensemble des objets d'un certain type.
- Noter le type de retour `const std::vector<Net*>&`.
  - Le template du conteneur `std::vector<Net*>` est bien utilisé comme un type.
  - La référence `&`, indique qu'il n'y aura pas de duplication du conteneur (copie) mais passage d'un simple pointeur.
  - Le qualificateur `const`, indique que le conteneur ne pourra être modifié. En cohérence avec la *constness* de la méthode. On ne modifie les attributs qu'en passant par les modificateurs, qui garantissent le maintient de la cohérence de la structure de données.

IV.5 La Classe Cell – Modifieurs

```
class Cell {
  public:
    void           add      ( Instance* );
    void           add      ( Term* );
    void           add      ( Net* );
    void           remove   ( Instance* );
    void           remove   ( Term* );
    void           remove   ( Net* );
    bool           connect  ( const std::string& name, Net* );
    unsigned int   newNetId ();
};
```

- `_newNetId()`, bien que publique ne doit être utilisée qu'en interne.
- Ajout & retrait des trois types d'objets composant la `Netlist`.
- Utilisation de la surcharge de fonction.

IV.5 Méthodes & Fonctions Statique (1)

```
// Cell.h
class Cell {
  public:
    static std::vector<Cell*>& getAllCells ();
    static Cell*               find ( const std::string& );
  //...
};

// Cell.cpp
vector<Cell*>  Cell::cells_;
```

- Attention, dans ce contexte, le mot clé `static` n'a pas du tout la même signification qu'en C (rappeler).
- `static` signifie ici, une méthode ou un attribut partagé par *tous* les objets de la classe.
- Les méthodes et attributs statiques peuvent être utilisés en l'absence d'objet.
- Corrollaire: n'étant pas créés avec les objets, les attributs statiques doivent être initialisés dans le `.cpp`.

IV.5 Méthodes & Fonctions Statiques (2)

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

Cell::Cell ( const string& name ) : name_      (name)
                                  , terms_     ()
                                  , instances_ ()
                                  , nets_      ()
                                  , maxNetIds_ (0) {
  if (find(name)) {
    cerr << "[ERROR] Attempt to create duplicate of Cell <"
         << name << ">.\n" << "          Aborting..." << endl;
    exit( 1 );
  }
  cells_.push_back( this );
}
```

- Une fonction membre statique ne peut accéder qu'aux attributs statiques et/ou appeler d'autres fonctions membres statiques.
- Une fonction membre ordinaire peut accéder aux attributs statiques et aux fonctions membres statiques.
- Nous pouvons maintenant écrire le constructeur de la class `Cell`.
- Noter la syntaxe d'initialisation des conteneurs, à ne surtout pas oublier. Ce n'est pas parce qu'il s'agit d'attributs *conteneurs* qu'ils ne doivent pas apparaître dans l'initialisation.

IV.5 Destructeur de la Classe Cell

```
Cell::~Cell ()
{
  for ( vector<Cell*>::iterator icell=cells_.begin()
      ; icell != cells_.end() ; ++icell ) {
    if (*icell == this) {
      cells_.erase( icell );
      break;
    }
  }
}
```

- Le constructeur ajoute le nouveau modèle au tableau `cells_`, donc le destructeur l'en retire.
- Maintient automatique de la cohérence de la structure de données.

IV.6 La Classe Instance – Attributs

```
class Instance {
  private:
    Cell*                owner_;
    Cell*                masterCell_;
    std::string          name_;
    std::vector<Term*>   terms_;
    Point                position_;
};
```

- Un nom d'instance.
- Une instance est associée à deux modèles (`Cell`) différents et à ne surtout pas confondre. D'une part le modèle qu'il référence (i.e. dont il est une instance), et d'autre part le modèle *dans lequel* il est instancié (le modèle courant).
- Du point de vue du graphe hiérarchique, c'est l'arc. (d'où les deux modèles).
- Enfin il possède une liste de connecteurs (`Term`).
- La liste des connecteurs est un *duplicata* de celle du modèle. Elles doivent, bien entendu, être identiques.
- Justification de la duplication: accrochage aux signaux différents pour chaque instance (donc on ne peut utiliser les `Term` du modèle).
- Enfin, comme nous réalisons un visualisateur de schémas, sa position.

IV.6 La Classe Instance – Méthodes

```
class Instance {
    Instance  ( Cell* owner, Cell* model, const std::string& );
   ~Instance  ();
    const std::string&
              getName        () const;
    Cell*     getMasterCell  () const;
    Cell*     getCell        () const;
    const std::vector<Term*>&
              getTerms       () const;
    Term*     getTerm        ( const std::string& ) const;
    Point     getPosition    () const;
    bool      connect        ( const std::string& name, Net* );
    void      add            ( Term* );
    void      remove         ( Term* );
    void      setPosition    ( const Point& );
    void      setPosition    ( int x, int y );
};
```

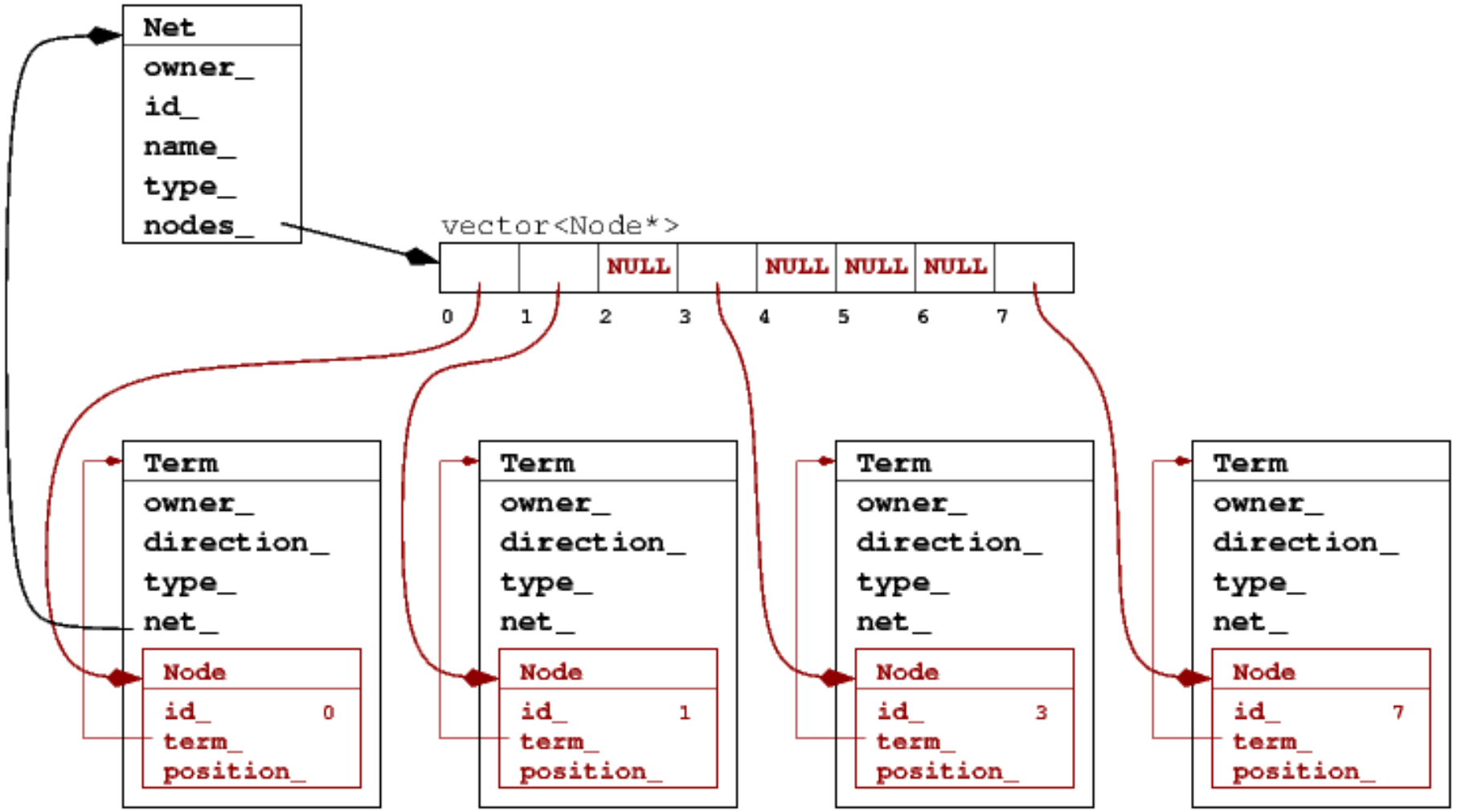
- Accès à la liste des connecteurs par référence. Quizz: pourquoi?
- Méthode `connect()`, créé une association entre un connecteur et un signal.

IV.7 La Classe Node – Attributs

```
class Node {
  public:
    static const size_t  noid; // numeric_limits<size_t>::max();
  public:
                      Node         ( Term*, size_t id=noid );
                     ~Node         ();
    inline  Point     getPosition  () const;
    inline  void      setPosition  ( const Point& );
    inline  void      setPosition  ( int x, int y );
    inline  size_t    getId        () const;
            Net*      getNet       () const;
    inline  Term*     getTerm      () const;
    inline  void      setId        ( size_t );
            void      toXml        ( std::ostream& ) const;
  protected:
    size_t  id_;
    Term*   term_;
    Point   position_;
};
```

- Classe dédiée pour stocker la position d'un objet `Term`.
- `noid` est une constante signifiant que l'index du noeud n'est pas initialisé. Sa valeur est celle du plus grand nombre que peut valoir le type `size_t`.
- Cette classe vous est fournie.

IV.7 Relations entre Net, Term et Node



- Gnagnagna.

IV.8 La Classe Term – Attributs

```
class Term {
  public:
    enum Type       { Internal=1, External=2 };
    enum Direction  { In=1, Out=2, Inout=3, Tristate=4, Transcv=5
                    , Unknown=6 };
  private:
    void*           owner_;
    std::string     name_;
    Direction       direction_;
    Type            type_;
    Net*            net_;
    Node            node_;
};
```

- Définit un terminal (logique) d'entrée/sortie d'un modèle **ou** d'une instance.
- Est distingué par un nom.
- Est relié à un signal du niveau hiérarchique *n*.
- Problème de l'*ownership* du `Term`. Il peut appartenir à une instance ou à un modèle.
- La distinction se fait par le type.
- La direction est une information sur le type électrique du fil.
- L'attribut `Node` est *inclus* dans l'objet (niché). Illustre le fait que les attributs peuvent être eux-même des objets arbitrairement complexes. En fait, nous l'avions déjà vu avec les conteneurs dans `Cell`.

IV.8 La Classe Term – Méthodes

```
class Term {
  public:
    Term      ( Cell* , const std::string& name, Direction );
    Term      ( Instance*, const Term* modelTerm );
   ~Term      ();
    bool                isInternal   () const;
    bool                isExternal   () const;
    const std::string&  getName      () const;
    Node*               getNode      ();
    Net*                getNet       () const;
    Cell*               getCell      () const;
    Cell*               getOwnerCell () const;
    Instance*           getInstance  () const;
    Direction           getDirection () const;
    Point               getPosition  () const;
    Type                getType      () const;
    void                setNet       ( Net* );
    void                setNet       ( const std::string& );
    void                setPosition  ( const Point& );
    void                setPosition  ( int x, int y );
    void                setDirection ( Direction );
};
```

- La position est gérée par l'attribut `Node`.
- Détailler le code de `getInstance()` et `getCell()`.

IV.8 La Classe Term – Propriétaire

```
Cell* Term::getCell () const
{ return (type_ == External) ? static_cast<Cell*>(owner_)
                             : NULL; }

Instance* Term::getInstance () const
{ return (type_ == Internal) ? static_cast<Instance*>(owner_)
                             : NULL; }
```

- C’est le type qui permet de savoir si le propriétaire est une `Instance` ou une `Cell`.
- Pour gagner un pointeur en mémoire, on stocke le pointeur vers le propriétaire sous forme de `void*`. Pour le re-convertir vers le bon type, utiliser `static_cast`.

IV.9 La Classe Net – Attributs

```
class Cell;
class Node;

class Net {
  private:
    Cell*                owner_;
    std::string          name_;
    unsigned int         id_;
    Term::Type           type_;
    std::vector<Node*>   nodes_;
};
```

- La `Cell` à laquelle appartient ce signal.
- Identification par `_id` plutôt que nom pour garantir l'unicité. Nous verrons dans la `Cell` comment est généré ce numéro unique.
- Un nom (pour les humains).
- On réutilise (`Term::Type`, i.e. interne ou externe).
- L'ensemble des `Node` auxquel il est connecté. Les objets `Node` permettant, à leur tour, d'accéder au `Term`. On verra dans la suite du cours pourquoi on passe par `Node` et non pas directement par `Term`.

IV.9 La Classe Net – Méthodes

```
class Net {
  public:
                          Net             ( Cell*
                                          , const std::string&
                                          , Term::Type );
                         ~Net             ();
    Cell*                 getCell         () const;
    const std::string&    getName         () const;
    unsigned int          getId           () const;
    Term::Type            getType         () const;
    Node*                 getNode         ( size_t id ) const;
    const std::vector<Node*>&
                          getNodes        () const;
    size_t                getFreeNodeId   () const;
    void                  add             ( Node* );
    bool                  remove          ( Node* );
};
```

- Le `Net` contient un tableau de pointeurs sur `Node`.
- L'index (`id_`) dans objet `Node` correspond à sa position dans le tableau de `Net`.
- Le tableau **peut contenir des trous**, c'est à dire des cases ne correspondant à aucun noeud. Dans ces cas, on prendra soin de mettre un pointeur `NULL`.
- Dans la fonction `add()`, si le noeud passé en argument possède **déjà** un index, le mettre dans la bonne case du tableau, éventuellement en agrandissant celui-ci jusqu'à la bonne taille.
- La méthode `remove()` ne réduit pas la taille du tableau, elle se contentera d'écrire `NULL` dans la case correspondante.
- Tout cela pour pouvoir désigner (et retrouver) un `Node` par son index de façon aussi commode que par un pointeur. Sera indispensable dans la suite (sauvegarde sur disque, et référence pour les lignes).

IV.10 Règles d'Implantation

✏ La structure de données ne gère que des pointeurs. Pourquoi?

```
class Net {
  public:
     Net ( Cell*, const std::string& name, Term::Type dir );
    ~Net ();
  private:
     Net ( const Net& );
  //...
};
```

- On voit assez bien que tous les objets de la base sont fortement interconnectés. Un objet n'existe jamais tout seul.
- La gestion des relations entre les objets est, le plus possible laissée à la charge de la base.
- De plus on ne veut pas avoir d'objets copiés intempestivement (pas deux signaux identiques mais dans deux objets différents).
- Enfin, on désire pouvoir utiliser les pointeurs comme identifiant. (comparer deux variables pointeurs)
- Pour toutes ces raisons, on ne travaille que sur des pointeurs, pour éviter les duplications.
- Et *par sécurité* on bloque l'usage du contructeur par copie.
- Technique: on le met en `private` et on ne fournit pas de définition de la fonction (ceinture & bretelles).

IV.11 Problématique de la Destruction

- Lorsqu'un objet de la base est détruit, il peut entraîner la destruction «en cascade» d'autres objets, car la base doit garantir, autant que possible la cohérence.
- Exemple: si un `Net` est détruit, il doit se «décrocher» des `Term` qui le réfèrent.
- Autre exemple: la destuction d'une `Cell` doit entraîner celle de tous ses sous-objets, et éventuellement des instances qui le réfère en tant que modèle.
- Conclusion: les corps des destructeurs ne seront pas vides cette fois-ci...

Pas de notes pour ce transparent.

Surcharge des Opérateurs pour les Calculs

```
class LogicValue {
  public:
    LogicValue& operator= ( const LogicValue& );
};

LogicValue  operator or ( const LogicValue& lhs
                        , const LogicValue& rhs )
{ return ou(lhs,rhs); }

int main ( int argc, char* argv[] ) {
  r.fromInt ( ou ( a, b ).toInt() );
  r = a or b;
}
```

Pas de notes pour ce transparent.

Surcharge des Opérateurs pour l'Affichage

```
class LogicValue {
  friend ostream& operator<< ( ostream&, const LogicValue& );
};

ostream& operator<< ( ostream& o, const LogicValue& v )
{ v.print(o); return o; }


int main ( int argc, char* argv[] ) {
  std::cout << "␣"; a.print ( std::cout );
  std::cout << "␣"; b.print ( std::cout );
  std::cout << "␣|␣␣"; r.print ( std::cout );
  std::cout << std::endl;

  std::cout << "␣" << a << "␣" << b << "␣|␣" << r << std::endl;
}
```

Pas de notes pour ce transparent.