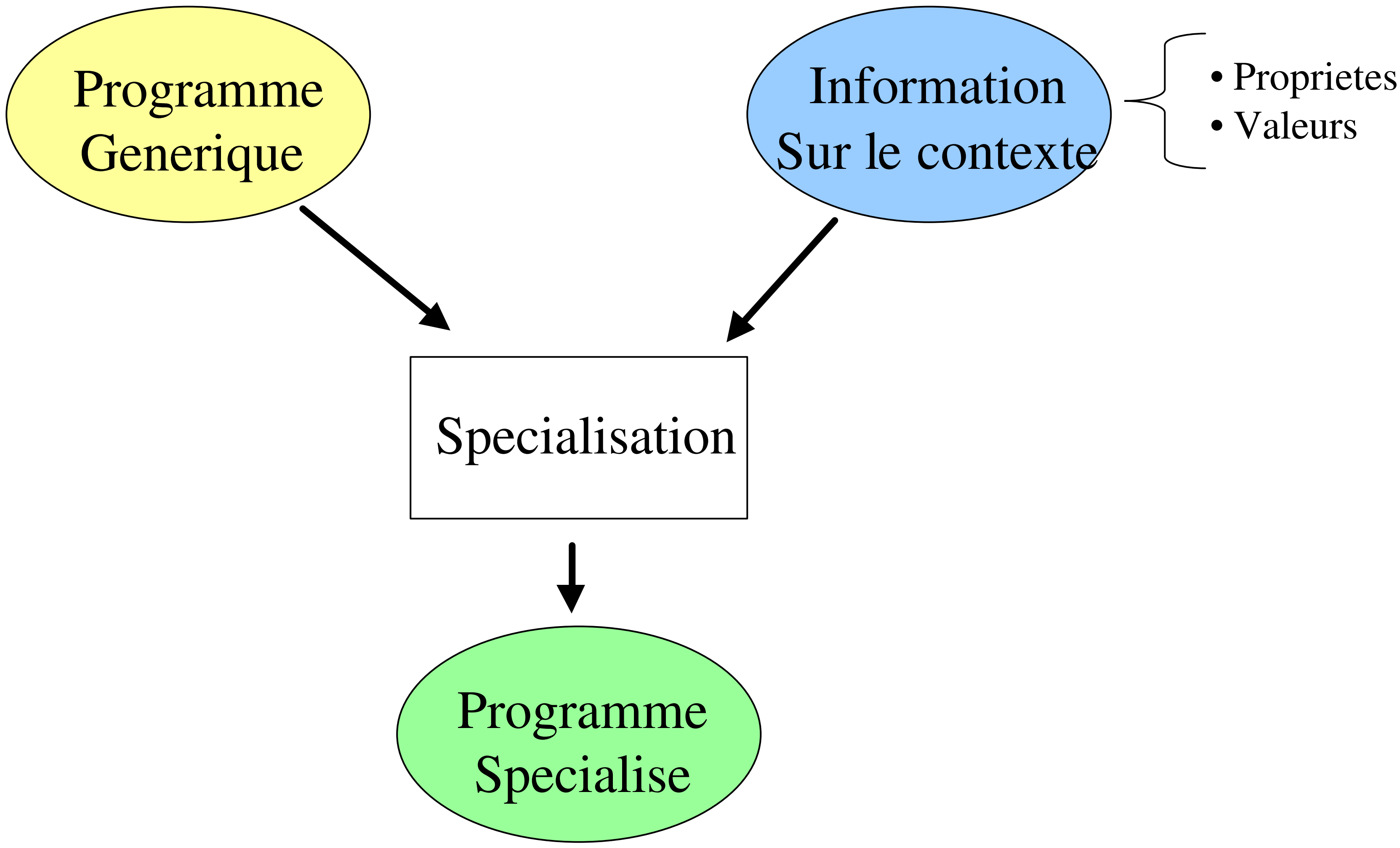
Declaration de specialisation de programmes C

Anne-Françoise Le Meur, Charles Consel
Groupe Compose -- http://compose.labri.fr

Specialisation de programme



Example: power

```
int power(int base, int expon)
{
    int accum = 1;
    while (expon > 0) {
        accum = accum * base;
        expon = expon - 1;
    }
    return(accum);
}
```

Specialisation: expon = 4

```
int power_expon (int base)
{
    return base*base*base*base;
}
```

Où en est la specialisation

– On y travaille depuis longtemps (+ vingt ans)
– Nombreuses analyses de programmes
– Implementation de specialiseurs pour
  » C (tempo, cmix)
  » Java (jspec)
  » scheme (schism)
– Specialisation a la compilation et a l ’execution

Efficace?

- Bons resultats pour differentes applications
  - Systemes
    - RPC
    - IPC
  - Calculs scientifiques
    - FFT
  - Graphiques

Mais...

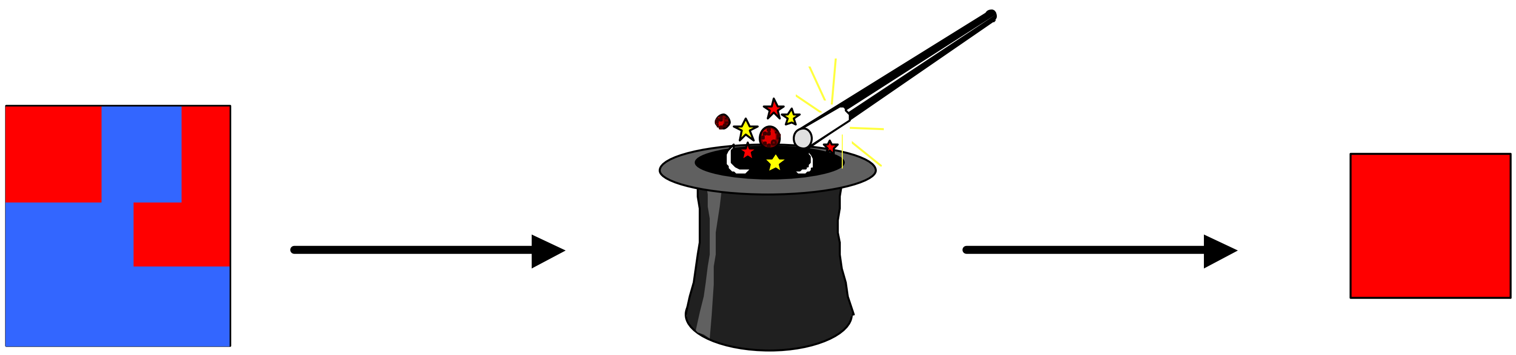
Constat :

la specialisation n'est utilisee que par des

experts en specialisation de programmes!!

Pourquoi?

– Specialiser un programme n’est pas un processus facile.

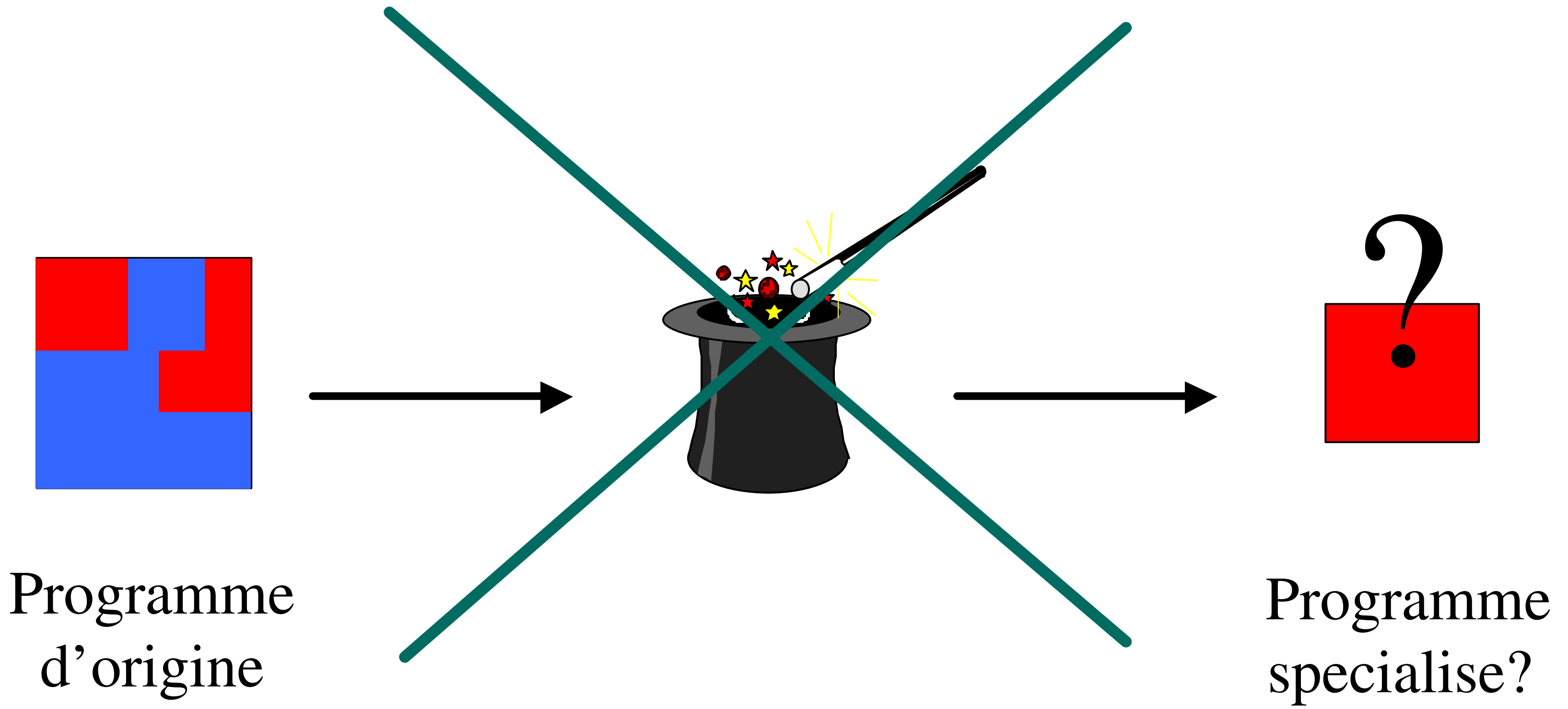


Programme d’origine

Programme specialise

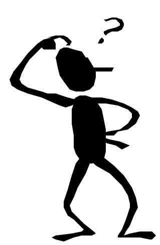
Pourquoi?

– Specialiser un programme n'est pas un processus facile.



Le parcours de l'utilisateur

- 1ere etape: on appuie sur le bouton "specialise moi"
  - » ca n'a pas trop l'air de marcher.
    - – code trop complexe
- 2eme etape: on relève les manches
  - » on etudie le code pour identifier les fragments de code qui semblent etre specialisables.
    - – pas facile de comprendre un programme
- 3eme etape: on donne ces petits bouts de code au specialiseur
  - » resultat toujours incertain. Pas grand chose a l'air de se passer.



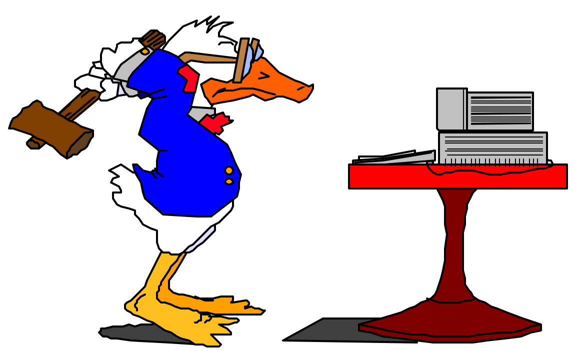
La parcours de l'utilisateur

- 4eme etape: On prend la doc du specialiseur...On a oublie de fournir au specialiseur des informations sur le context.
  - comment preciser ce contexte?
    - etude plus approfondie de la doc (flags, options)
    - beaucoup de choses a preciser (BT, alias)

```
struct vector { int size; int* data}
Pour exprimer qu'un pointeur sur struct vector est static:
struct vector tmp;
void set_analysis_context(struct vector **u, struct vector **v) {
(*u) = &tmp;
(*v) = &tmp;}

et puis pour dire que le champs size est static
static_locations := ["vector.size"];
```

La parcours de l'utilisateur

- 5eme etape: on essaie de fournir le contexte tant bien que mal
  - pas tres concluant
- 6eme etape: on regarde les fichiers resultant des diverses analyses pour essayer de comprendre
- 7eme etape: on fait des modifications dans le code et les declarations du contexte (au petit bonheur) et on itere.
  - comment deviner ce que font les analyses?
- 54eme etape:



Enfin...

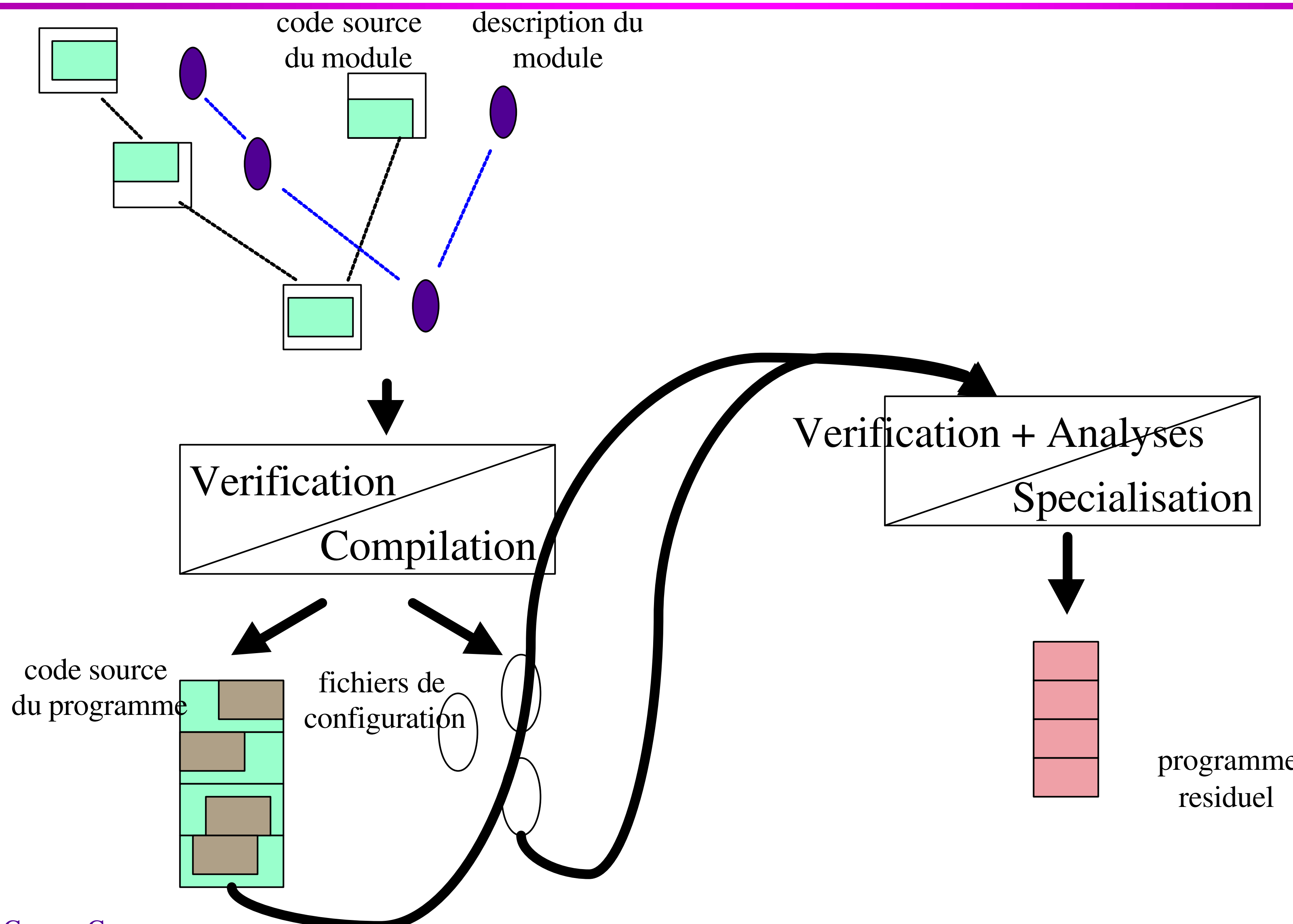
– 77eme etape: on arrive a un resultat qui semble satisfaisant.

  » MAIS comment savoir si toutes les opportunites de specialisation qu'on avait reperees dans le code ont vraiment ete specialisees??
  » Et si elles n'ont pas pu etre specialisees, pourquoi?

Pour resumer

- Le processus de specialisation est complique
  - » structure du code n’est pas appropriee. Les fragments de code a specialiser sont disperses.
    - – il faudrait les identifier une fois pour toute. (qui le fait?)
  - » utilisation du specialiseur est complexe
    - – on ne devrait pas avoir besoin de connaitre les details
  - » la specialisation n’est pas previsible
    - – il faudrait pouvoir declarer son intention!

Ce qu'on aimerait



Notre approche

- Un langage pour declarer les opportunites de specialisation (lors de la creation du code)
  - » Scenarios de specialisation
    - – quoi, quand (quels contextes: S, D)
  - » Composition de scenarios
- Compilation des declarations
- Verification ajoutees dans les analyses pour detecter les incoherences.

Declarer la genericite

- Donnees (invariant)
  - Parametres
  - Globales
  - Structures de donnees

- Programmmes (modules)
  - Specialisation de module
    - declaration de scenarios de specialisation

mini-printf

```
void mini_printf(char * fmt, int value)
{
 int i;
 for(i = 0 ; *fmt != '\0' ; fmt++)
    if(*fmt != '%')
      putchar(*fmt);
    else
      switch(*++fmt) {
        case 'd'  : putint(value); break;
        case '%': putchar('%'); break;
        default   : error(); break;
      }
}
```

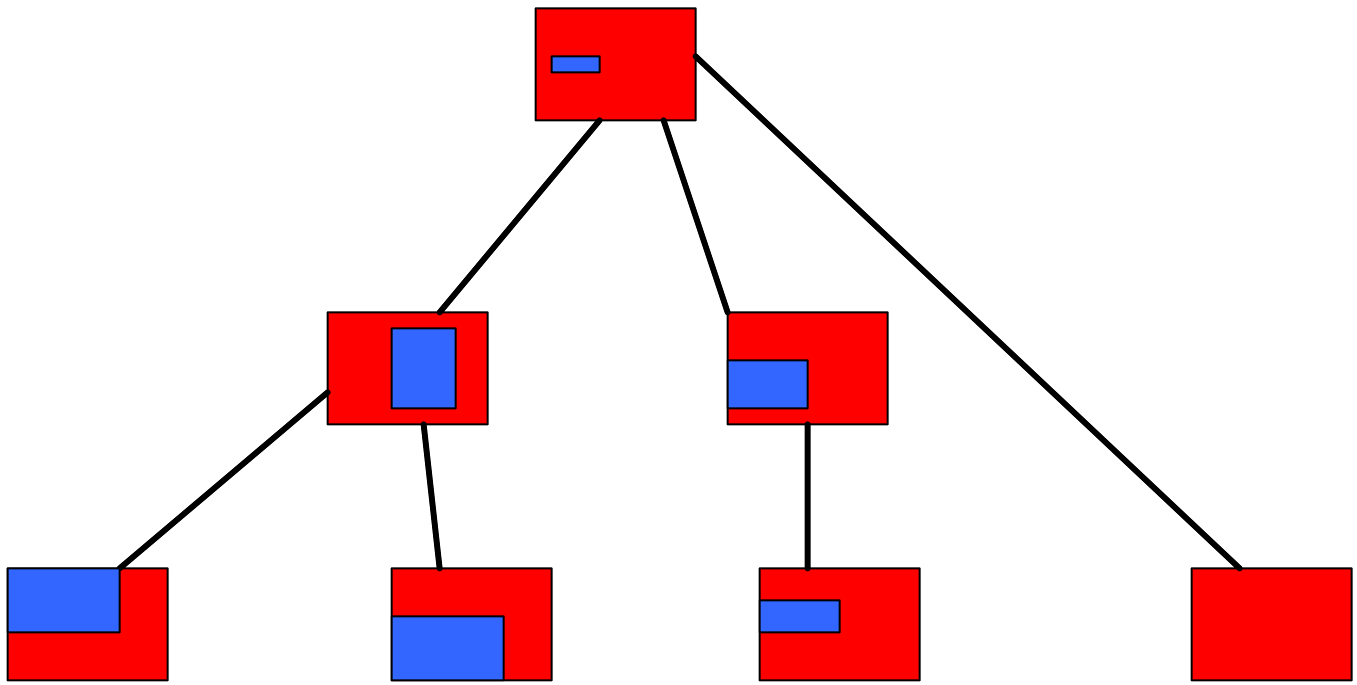
```
Module mini_printf {
Defines {
    From mprintf.c {
      Btmprintf mini_printf(S(char) S(*) , D(int))
              }
}
```

Example: dotproduct

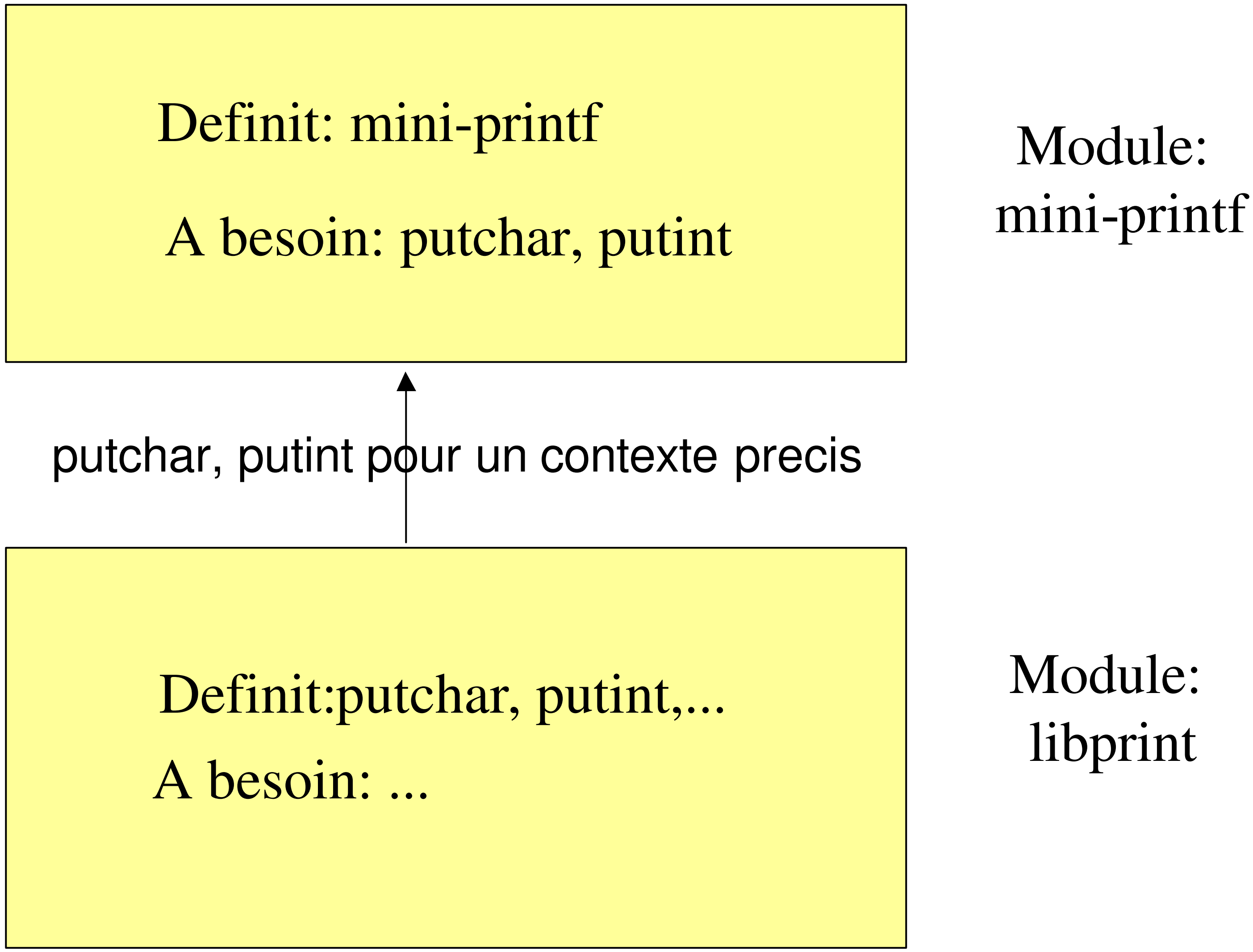
```
// dotproduct.h
struct vector{int size; int* values};

// dotproduct.c
extern void mini_print();

int dotproduct(vector* v1,vector* v2) {
        int i;
        int sum = 0;
        for (i=0; i<v1.size; i++)
           sum += v1.values[i] *
             v2.values[i];
             mini_printf( "n=%d" , sum);
        return sum;
}
```

```
Module dotproduct {
Defines{
    From dotproduct.h {
        BtVect1  struct vector{S(int); D(int)D(*)};
        BtVect2  struct vector{S(int); S(int)S(*)}; }
    From /mylibs/dotproduct.c {
      Btdot1 dotproduct(BtVect1(vector) S(*),
                        BtVect1(vector) S(*)) ;
      Btdot2 dotproduct(BtVect2(vector) S(*),
                        BtVect1(vector) S(*))
    }
  }
}
```

Hierarchie de modules



Exemple: mini-printf



Exemple: mini-printf

Definit: mini-printf

A besoin: putchar, putint

Module: mini-printf

Btputchar2,Btputint1

```
Module libprint{
Imports {...}
Defines {
   From libprint.c {
     extern Btputchar1 putchar(D(int));
     extern Btputchar2 putchar(S(int));
     extern Btputint1 putint(D(int) );
  .... }.....;}
Exports {  Btputint1, Btputchar1,... }}
```

Module: libprint

Exemple: mini-printf

```
Module mini_printf {
Imports {From libprint.mdl { Btputint1; Btputchar2}}
Defines {
   From mprintf.c {
     intern Btmprintf mini_printf(S(char) S(*) , D(int))
            {needs: Btputint1, Btputchar2} ; };}
Exports {  Btmprintf; }}
```

Module: mini-printf

Btputchar2,Btputint1

```
Module libprint{
Imports {...}
Defines {
   From libprint.c {
     extern Btputchar1 putchar(D(int));
     extern Btputchar2 putchar(S(int));
     extern Btputint1 putint(D(int) );
  .... }.....;}
Exports {  Btputint1, Btputchar1,... }}
```

Module: libprint

Resultat de la specialisation

Specialisation: *fmt = “n = %d”

```
void mini_printf_fmt (int  value)
{
     putchar(’n’);
     putchar(’ ’);
     putchar(’=’);
     putchar(’ ’);
     putint(value);
}
```

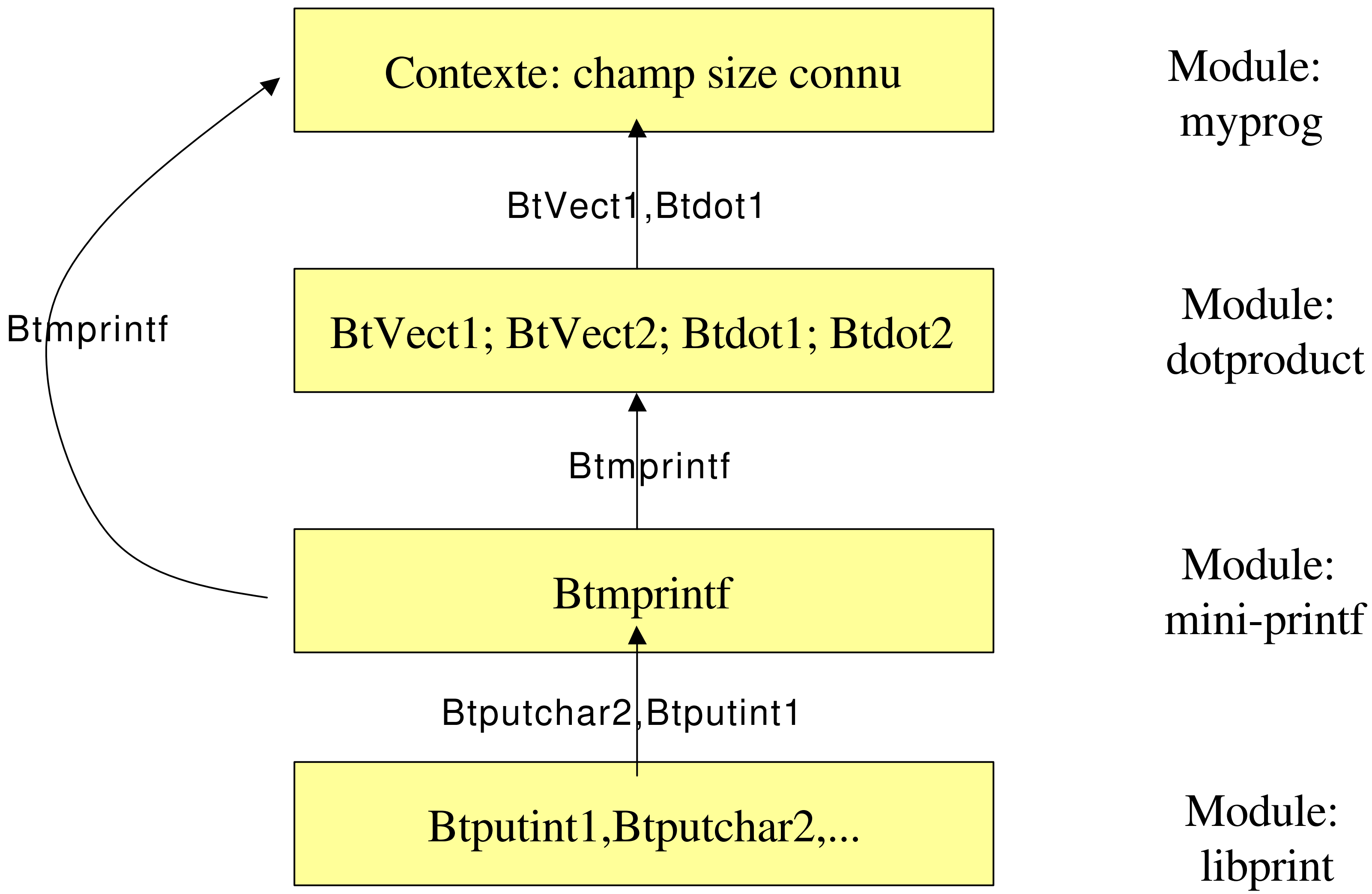
Important: Notion de contrat de specialisation

Example: dotproduct

```
// dotproduct.h

struct vector{int size; int* values};


// dotproduct.c
extern void mini_print();


int dotproduct(vector* v1,vector* v2) {
        int i;
        int sum = 0;
        for (i=0; i<v1.size; i++)
           sum += v1.values[i] *
             v2.values[i];
             mini_printf( "n=%d" , sum);
        return sum;
}
```

```
Module dotproduct {
   Imports { From mini_printf.mdl { Btmprintf; }}
   Defines {
   From dotproduct.h {
      BtVect1  struct vector{S(int); D(int)D(*)};
      BtVect2  struct vector{S(int); S(int)S(*)}; }
   From /mylibs/dotproduct.c {
     intern Btdot1 dotproduct(BtVect1(vector) S(*),
   BtVect1(vector) S(*)) {needs: Btmprintf} ;
     intern Btdot2 dotproduct(BtVect2(vector) S(*),
   BtVect1(vector) S(*)) {needs: Btmprintf}; }    }
  Exports { BtVect1; BtVect2; Btdot1; Btdot2 } }
```

Exemple: dotproduct



Exemple : dotproduct

```
int dotproduct(vector* v1,vector* v2) {
        int i;
        int sum = 0;
        for (i=0; i<v1.size; i++)
           sum += v1.values[i] *
             v2.values[i];
             mini_printf( "n=%d" , sum);
        return sum;
}
```

Specialization: v1.size = 3

```
// dotproduct.spe.c
int dotproduct(vector* v1, vector* v2) {

        int sum = v1.values[0] * v2.values[0] +
                        v1.values[1] * v2.values[1] +
                        v1.values[2] * v2.values[2];
        putchar('n');
        putchar(' ');
        putchar('=');
        putchar(' ');
        putint(sum);
        return sum;
}
```

Fichier de Description

- Permet au programmeur d'exprimer
  - Qu'est ce qui peut etre specialise et quand
  - Comment composer les modules et donc les scenarios
- La structure du code est guidee par la decomposition d'un scenario en plusieurs scenarios
- Renseigne l'utilisateur sur le contexte dans lequel un programme peut se specialiser: contrat
- Bons supports pour faciliter la specialisation de systemes de grandes tailles

Compilation des declarations

//// dotproduct.actx.c

```
#include "dotproduct.h"
struct vector tmp;
void set_analysis_context(struct vector **u, struct vector **v) {
(*u) = &tmp;
(*v) = &tmp;

}
```

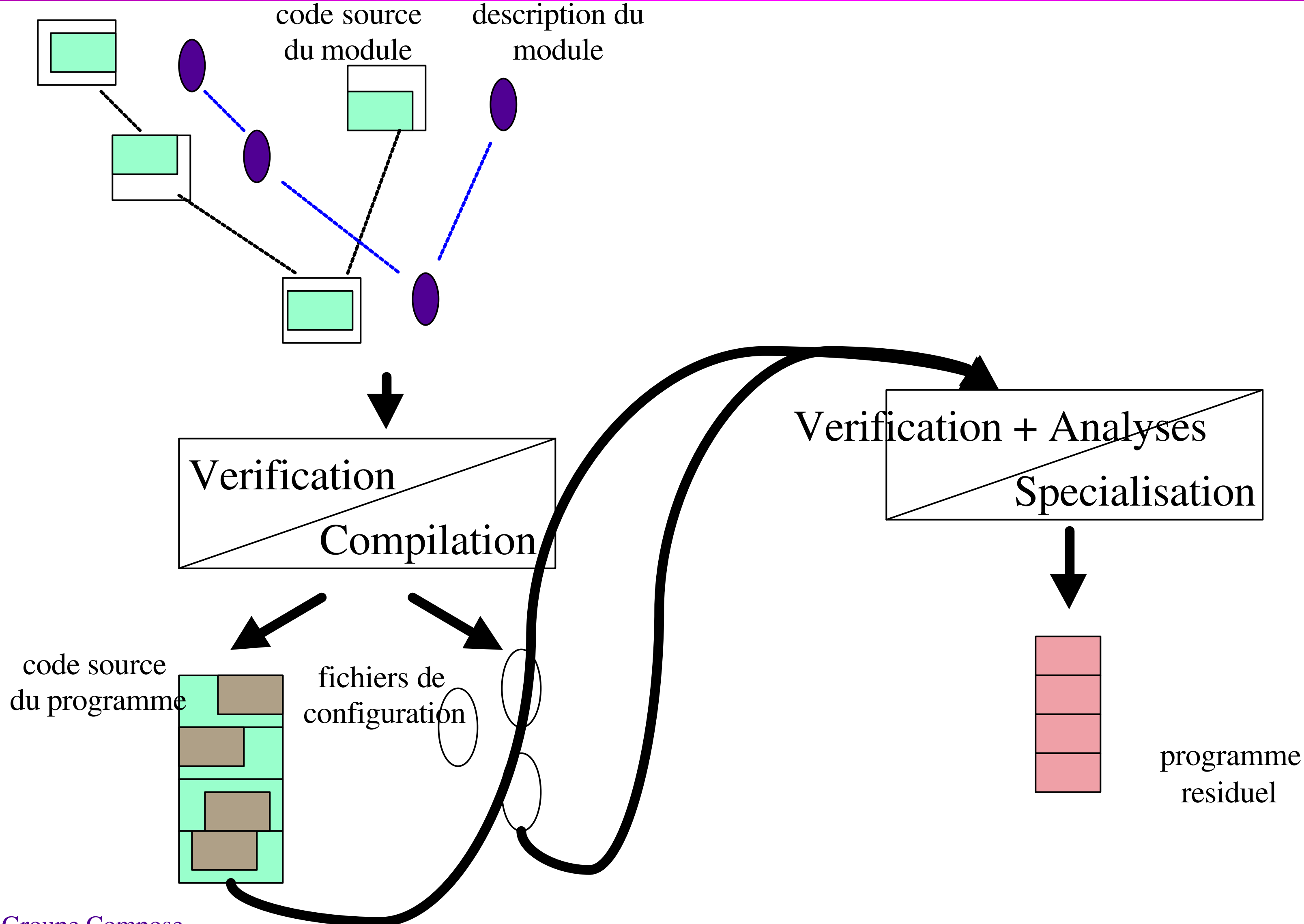
dotproduct (btvec1(struct vector) S(*), btvec1(struct vector)S(*))

//// dotproduct.config.sml

```
entry_point := "dotproduct(_,_)";
static_locations := ["vector.size"];
```

Btvec1 {S(int); D(int) D(*)} vector}

Mecanismes


code source
du module
description du
module
Verification
Compilation
Verification + Analyses
Specialisation
code source
du programme
fichiers de
configuration
programme
residuel

Conclusion

– Pour rendre la specialisation utilisable

  » langage de declaration + compilation des declaration + processus de specialisation

– Pour rendre la specialisation previsible

  » langage de declaration + verification

– Tempo: http://compose.labri.u-bordeaux.fr/prototypes/tempo