The SANTE Method: Value Analysis, Program Slicing and Test Generation for C Program Debugging

Omar Chebaro [1,2] **Nikolai Kosmatov** [1]

Alain Giorgetti [2] Jacques Julliand [2]

[1] CEA LIST, Paris
[2] LIFC, University of Franche-Comté, Besançon

EPFL, April 24th, 2012

Outline

- **Context and objectives**
- **SANTE: Basic options**
- **SANTE: Advanced options**
- **Experiments**
- **Conclusion & Perspectives**

Software in critical systems

Airplanes, automobiles, etc..

Trajectory, behavior, transmission, etc..

Phone, TV, banking, smart cards, electronic wallets, etc..

Medical devices, pacemaker, glucose control, robotic surgery, etc..

- Software testing: usually accounts for **50%** of software development **cost.**
- **High cost** of software failures (59.5 billion dollars per year in USA [NIST 2002]).
- In critical applications, bugs might entail **human damages**

Verification remains a crucial part in software development process

The C language is risky!

C Language:

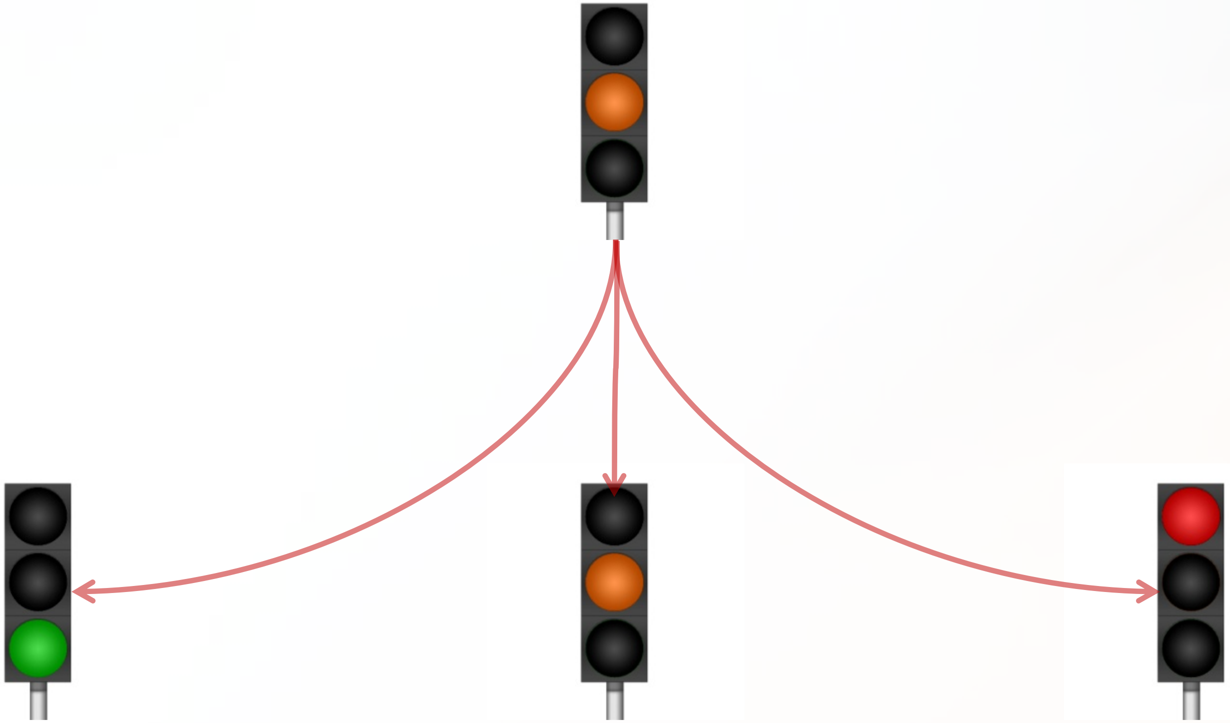
- Low-level operations
- Widely used for critical software
- Lack of security mechanisms

Runtime errors are common:

- Division by 0
- Invalid array index
- Invalid pointer
- Non initialized variable
- Out-of-bounds shifting
- Arithmetical overflow
- …

Terminology

Alarm: a reported threat



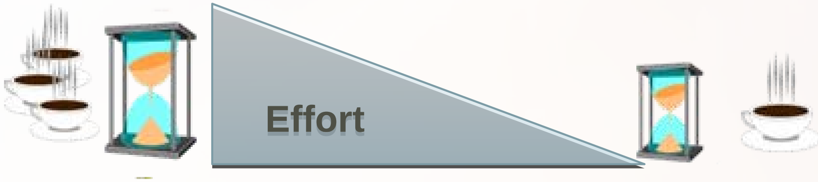
False alarm: an alarm signaling an error that never appears at runtime

Error (bug): an element in the code causing incorrect behavior of the software

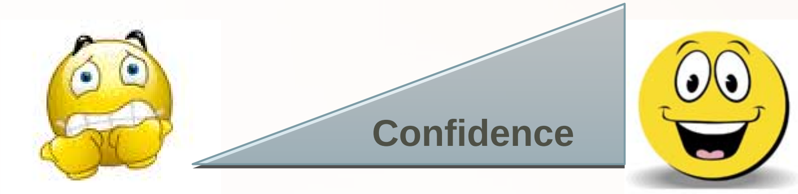
Objectives

1. **Classify as many alarms as possible**

2. **As automatically as possible**



3. **Provide the validation engineer with the most precise information on the detected errors**

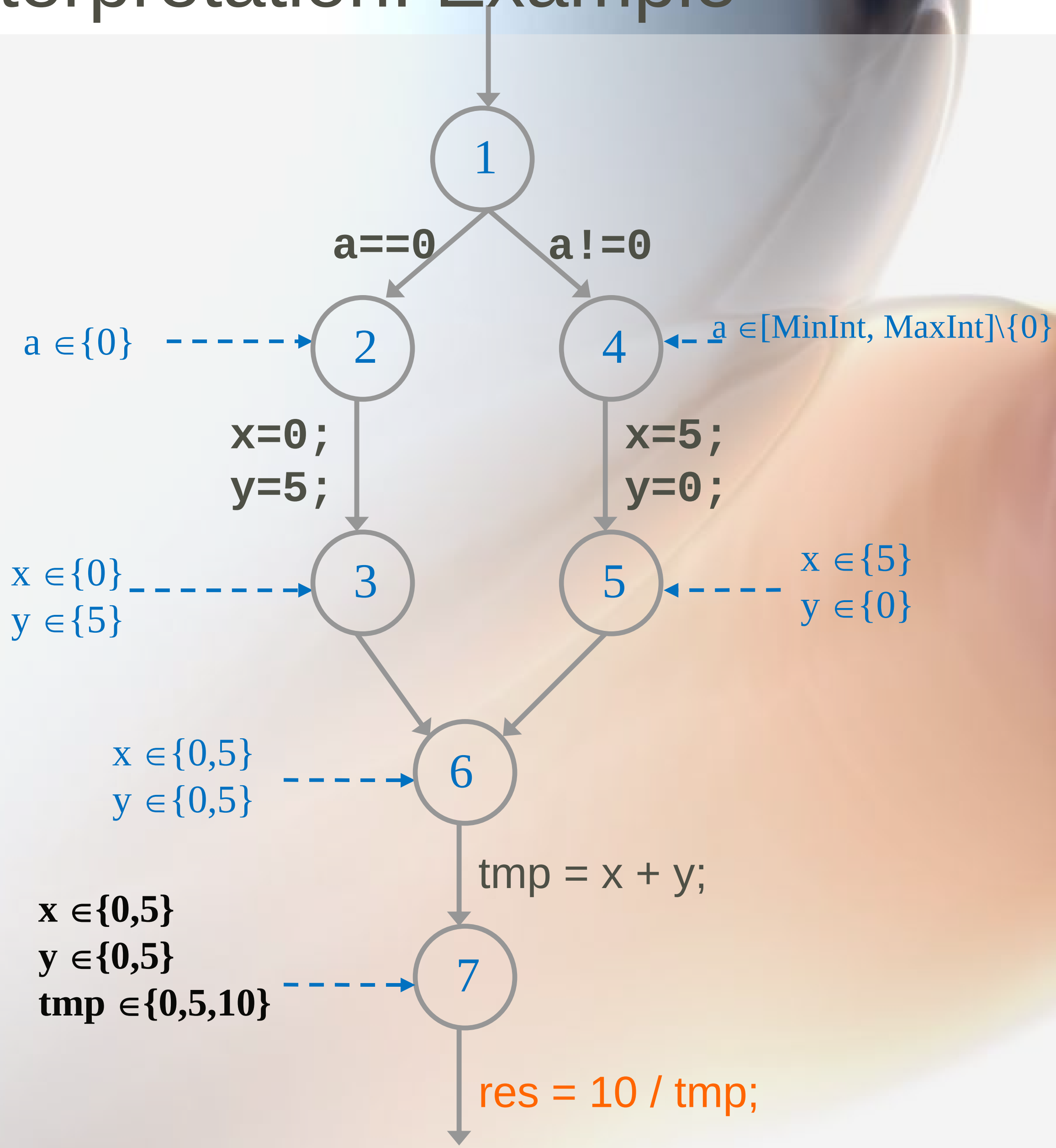


Abstract Interpretation

- Constructs an abstraction including all concrete executions
- Says: YES (error), NO (safe) or MAYBE (?)
- Value analysis
  - Based on abstract interpretation
  - Computes an overapproximation of possible values of variables at each instruction
  - Signals alarms
  - Sometimes wrongly (false alarms)
  - Computes properties over a huge (infinite) space

Abstract Interpretation: Example

```
void f ( int a ) {
   if(a == 0){
         x = 0;   y = 5;
   } e l s e {
         x = 5;   y = 0;
   }
   tmp = x + y;
   res = 10 / tmp;
}
```

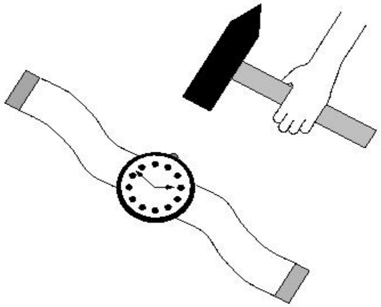


Program Slicing

1. Simplifies the program using control and data dependencies
2. Preserves the behaviors w.r.t. a criterion
3. Slicing criterion: $C = (l, \{x_1, \ldots, x_n\})$

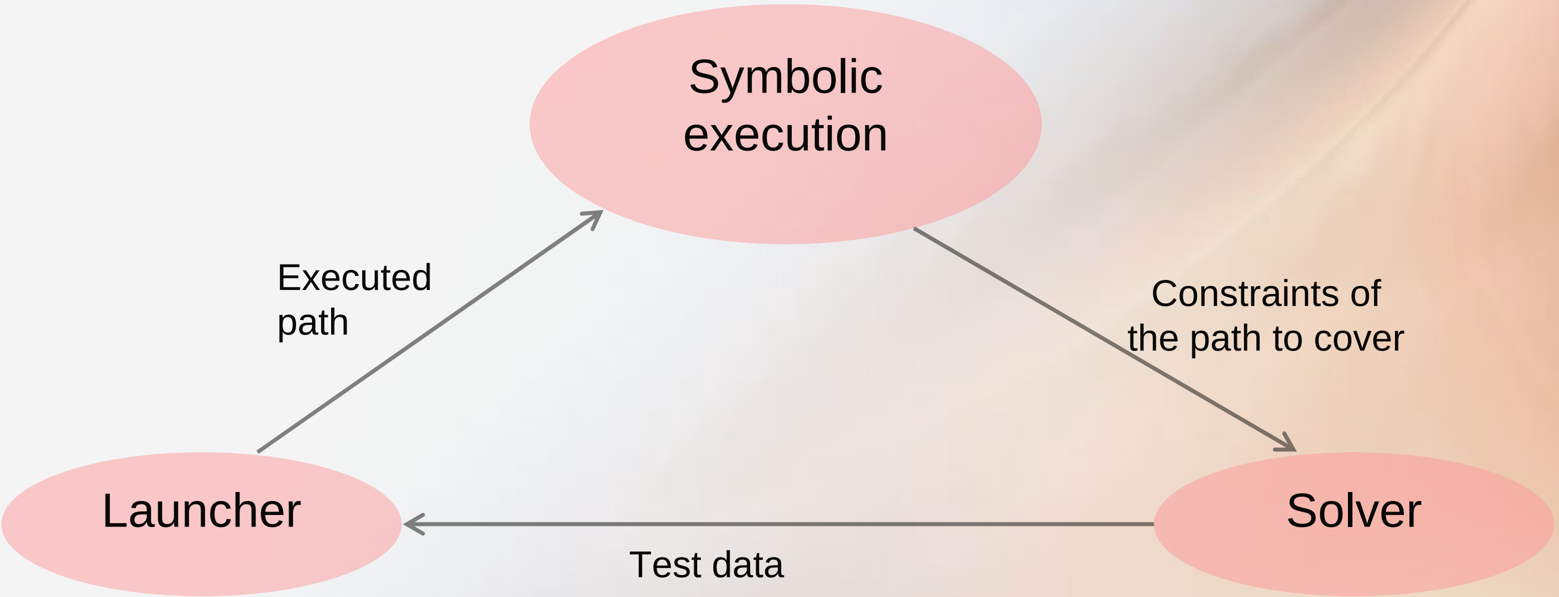
   preserves the executions reaching the point l and keeps at l the same values for the variables $x_1, \ldots, x_n$

Dynamic Analysis



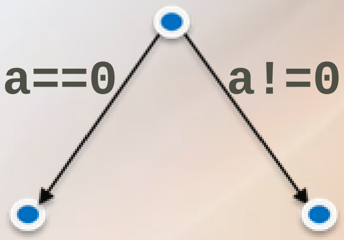
Concolic test generation:

Combines concrete and symbolic execution



Test Generation: Example

```
void f ( int a ){
    if(a == 0){
        x = 0;   y = 5;
    } else {
      x = 5;   y = 0;
    }
    tmp = x + y;
    res = 10 / tmp;
}
```



SANTE (Static ANalysis and TEsting)

- **Static Analysis**: Analyzes the source code without executing it
- **Test Generation**: Executes the program to find anomalies or defects.

Value Analysis	Test Generation
Abstraction	Concrete execution
Complete	Incomplete
Imprecise	Precise
Frama-C tool	PathCrawler tool

Combines static analysis and test generation for runtime error detection in C programs.

Tools used by SANTE

Frama-C:

- Developed at CEA LIST
- Framework for static analysis of C programs
- Organized in a plugin architecture
- Value analysis plugin: computes sound over-approximated sets of possible values => alarms
- Slicing plugin: simplifies the code
- http://frama-c.com

PathCrawler:

- Developed at CEA LIST, uses the COLIBRI constraint solver
- Concolic test generation tool for C programs
- all-path or k-path criterion
- k-path: paths with at most k consecutive loop iterations
- Explores paths in a depth-first search
- http://pathcrawler-online.com

Outline

- Context and objectives
- **SANTE: Basic options**
- SANTE: Advanced options
- Experiments
- Conclusion & Perspectives

Example

- Function: eurocheck
- An open source program
- Validates serial numbers of euro banknotes.
- Precondition:

  str is null or a zero terminated string
- Contains 1 bug

```
int eurocheck(char *str){
  unsigned char sum;
  char c[9][3]={"ZQ","YP","XO",
  "WN","VM","UL","TK","SJ","RI"};
  unsigned char checksum[12];
  int i = 0, len = 0;
  if(str[0]>=97 && str[0]<=122)
    str[0]-=32; //capitalize
  if(str[0]<'I' || str[0]>'Z')
    return 2; //invalid char
  if(strlen(str) != 12)
    return 3; //wrong length
  len = strlen(str);
  checksum[i]=str[i];
  for(i=1;i<len;i++){
    if(str[i]<48 || str[i]>57)
      return 4; //not a digit
    checksum[i] = str[i]-48;}
  sum=0;
  for(i=1;i<len;i++)
    sum+=checksum[i];
  while(sum>9)
    sum=((sum/10)+(sum%10));
  for(i=0;i<9;i++)
    if(checksum[0]==c[i][0])
      break;
  if(sum!=i)
    return 5; //wrong checksum
  return 0;} //OK
```

Example

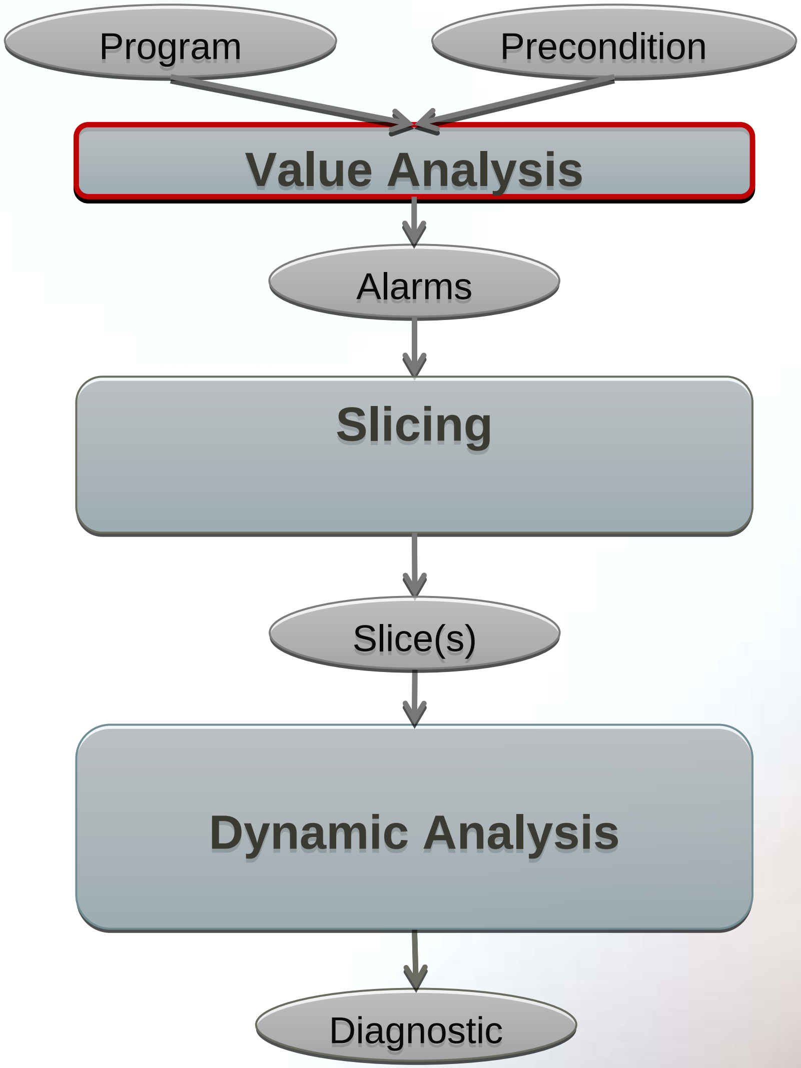
- Function: eurocheck
- An open source program
- Validates serial numbers of euro banknotes.
- Precondition:

  str is null or a zero terminated string

- Contains 1 bug

```
int eurocheck(char *str){
  unsigned char sum;
  char c[9][3]={"ZQ","YP","XO",
  "WN","VM","UL","TK","SJ","RI"};
  unsigned char checksum[12];
  int i = 0, len = 0;
  if(str[0]>=97 && str[0]<=122)
    str[0]-=32; //capitalize
  if(str[0]<'I' || str[0]>'Z')
    return 2; //invalid char
  if(strlen(str) != 12)
    return 3; //wrong length
  len = strlen(str);
  checksum[i]=str[i];
  for(i=1;i<len;i++){
    if(str[i]<48 || str[i]>57)
      return 4; //not a digit
    checksum[i] = str[i]-48;}
  sum=0;
  for(i=1;i<len;i++)
    sum+=checksum[i];
  while(sum>9)
    sum=((sum/10)+(sum%10));
  for(i=0;i<9;i++)
    if(checksum[0]==c[i][0])
      break;
  if(sum!=i)
    return 5; //wrong checksum
  return 0;} //OK
```

SANTE: value analysis



Program

Precondition

Value Analysis

Alarms

Slicing

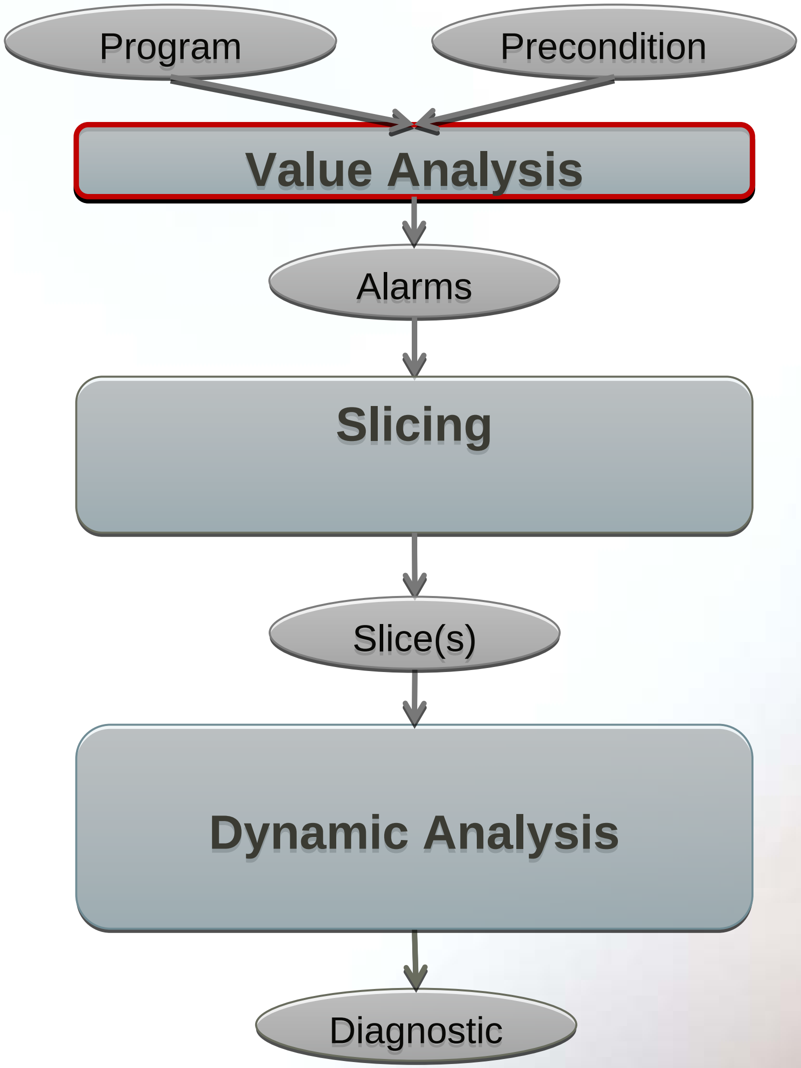
Slice(s)

Dynamic Analysis

Diagnostic

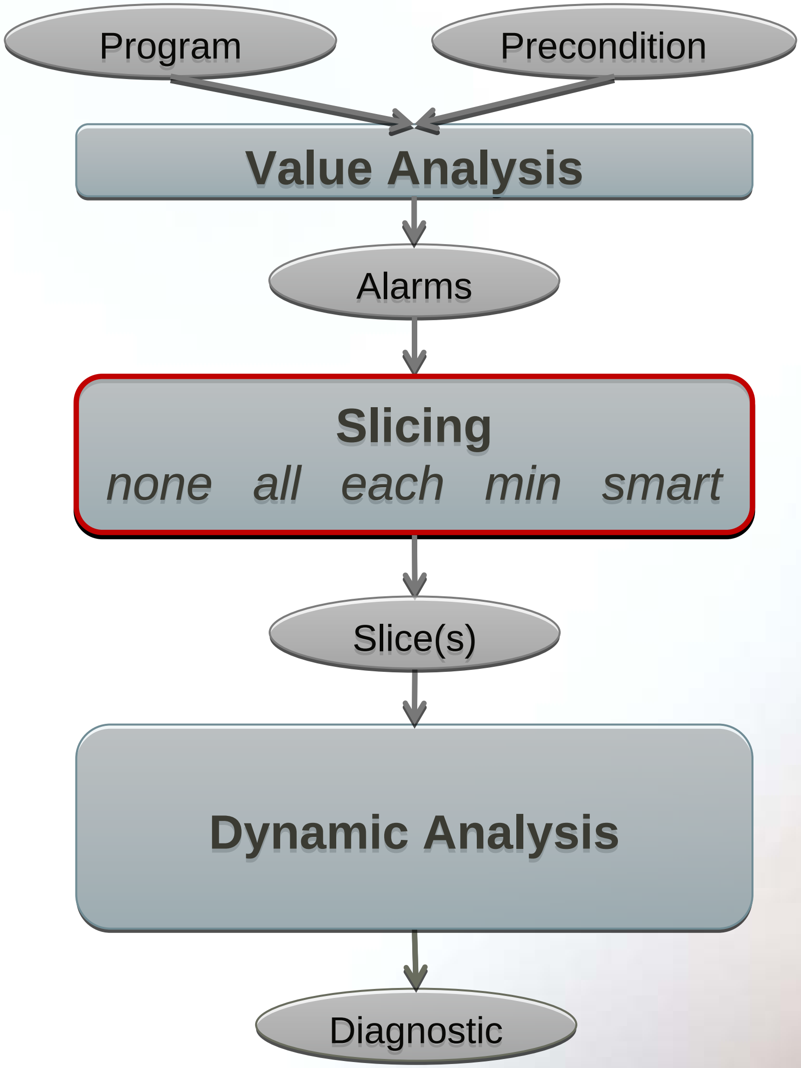
```
int eurocheck(char *str){
  unsigned char sum;
  char c[9][3]={"ZQ","YP","XO",
  "WN","VM","UL","TK","SJ","RI"};
  unsigned char checksum[12];
  int i = 0, len = 0;
  if(str[0]>=97 && str[0]<=122)
    str[0]-=32; //capitalize
  if(str[0]<'I' || str[0]>'Z')
    return 2; //invalid char
  if(strlen(str) != 12)
    return 3; //wrong length
  len = strlen(str);
  checksum[i]=str[i];
  for(i=1;i<len;i++){
    if(str[i]<48 || str[i]>57)
      return 4; //not a digit
    checksum[i] = str[i]-48;}
  sum=0;
  for(i=1;i<len;i++)
    sum+=checksum[i];
  while(sum>9)
    sum=((sum/10)+(sum%10));
  for(i=0;i<9;i++)
    if(checksum[0]==c[i][0])
      break;
  if(sum!=i)
    return 5; //wrong checksum
  return 0;} //OK
```

SANTE: value analysis



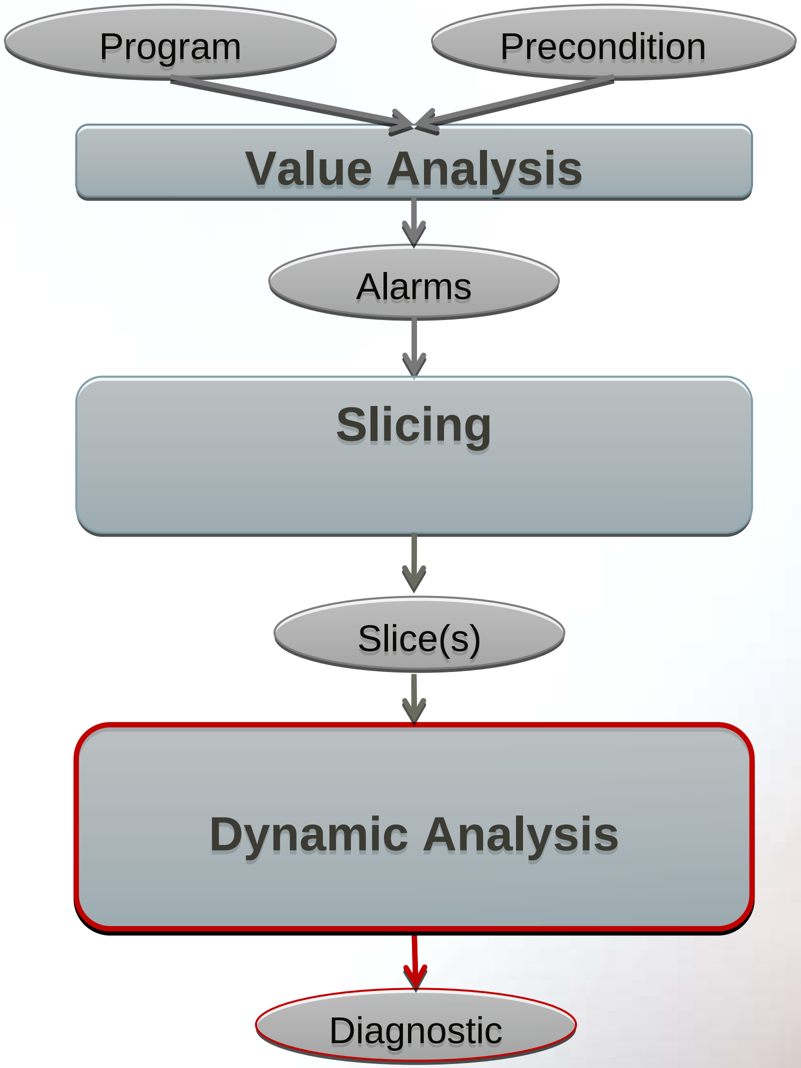
```
int i = 0, len = 0;
//@ assert(\valid(str+0));
if(str[0]>=97 && str[0]<=122)
  str[0]-=32; //capitalize
if(str[0]<'I' || str[0]>'Z')
  return 2; //invalid char
if(strlen(str) != 12)
  return 3; //wrong length
len = strlen(str);
//@ assert(\valid(str+i));
checksum[i]=str[i];
for(i=1;i<len;i++){
  //@ assert(\valid(str+i));
  if(str[i]<48 || str[i]>57)
    return 4; //not a digit
  //@ assert(0<=i && i<12);
  checksum[i] = str[i]-48;}
sum=0;
for(i=1;i<len;i++)
  //@ assert(0<=i && i<12);
  sum+=checksum[i];
while(sum>9)
  sum=((sum/10)+(sum%10));
for(i=0;i<9;i++)
```

SANTE: Slicing



```
5    int i = 0, len = 0;
6_0  //@ assert(\valid(str+0));
6    if(str[0]>=97 && str[0]<=122)
7      str[0]-=32; //capitalize
8    if(str[0]<'I' || str[0]>'Z')
9      return 2; //invalid char
10   if(strlen(str) != 12)
11     return 3; //wrong length
12   len = strlen(str);
13_0 //@ assert(\valid(str+i));
13   checksum[i]=str[i];
14   for(i=1;i<len;i++){
15_0   //@ assert(\valid(str+i));
15     if(str[i]<48 || str[i]>57)
16       return 4; //not a digit
17_0   //@ assert(0<=i && i<12);
17     checksum[i] = str[i]-48;}
18   sum=0;
19   for(i=1;i<len;i++)
20_0   //@ assert(0<=i && i<12);
20     sum+=checksum[i];
~~21   while(sum>9)~~
~~22     sum=((sum/10)+(sum%10));~~
~~23   for(i=0;i<9;i++)~~
```

SANTE: Dynamic Analysis



Program

Precondition

Value Analysis

Alarms

Slicing

Slice(s)

Dynamic Analysis

Diagnostic

```
int eurocheck(char *str){
  unsigned char sum;
  char c[9][3]={"ZQ","YP","XO",
  "WN","VM","UL","TK","SJ","RI"};
  unsigned char checksum[12];
  int i = 0, len = 0;
  if(str[0]>=97 && str[0]<=122)
    str[0]-=32; //capitalize
  if(str[0]<'I' || str[0]>'Z')
    return 2; //invalid char
  if(strlen(str) != 12)
    return 3; //wrong length
  len = strlen(str);
  checksum[i]=str[i];
  for(i=1;i<len;i++){
    if(str[i]<48 || str[i]>57)
      return 4; //not a digit
    checksum[i] = str[i]-48;}
  sum=0;
  for(i=1;i<len;i++)
    sum+=checksum[i];
  while(sum>9)
    sum=((sum/10)+(sum%10));
  for(i=0;i<9;i++)
    if(checksum[0]==c[i][0])
      break;
  if(sum!=i)
    return 5; //wrong checksum
  return 0;} //OK
```

Adding Error Branches

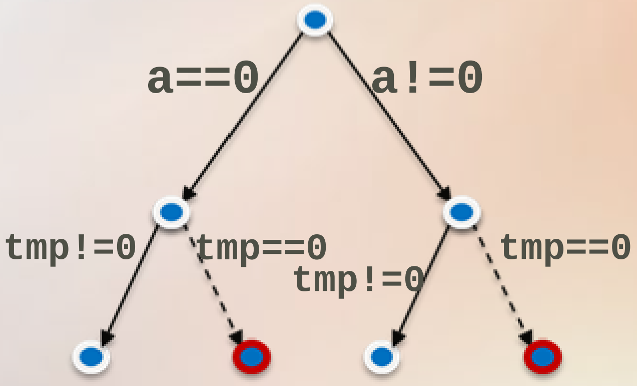
```
//@ assert( i >= 0 && i < 12 );
 checksum[i] = str[i]-48;
```

```
if( i < 0 || i >= 12 ) error(); else
 checksum[i] = str[i]-48;
```

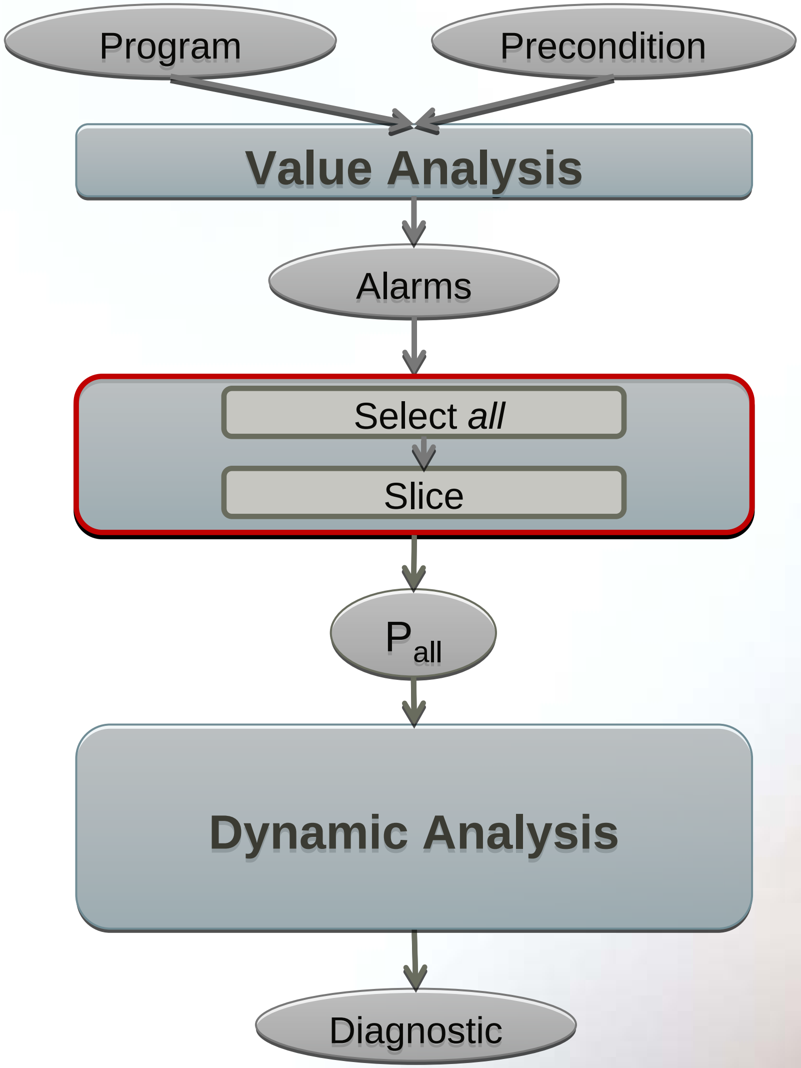
```
//@ assert( \valid(str+i) );
checksum[i]=str[i];
```

```
if( i < 0 || i >= length(str) ) error(); else
  checksum[i]=str[i];
```

```
void f ( int a ) {
 if(a == 0){
   x = 0;   y = 5;
 } else {
   x = 5;   y = 0;
 }
 tmp = x + y;
 if( tmp == 0 ) error(); else
 res = 10 / tmp;
}
```



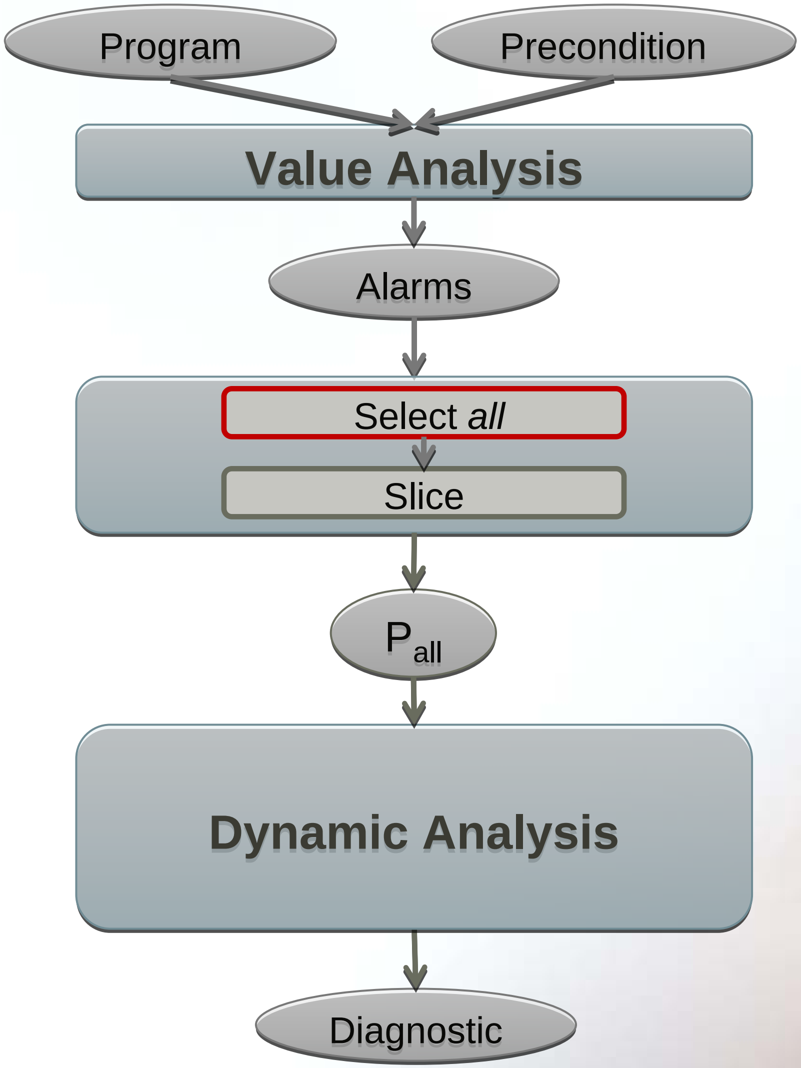
SANTE: option *all*



Slice w.r.t. the set of all alarms

```
1   int eurocheck(char *str){
2     unsigned char sum;
3     char c[9][3]={"ZQ","YP","XO",
      "WN","VM","UL","TK","SJ","RI"};
4     unsigned char checksum[12];
5     int i = 0, len = 0;
6_0   //@ assert(\valid(str+0));
6     if(str[0]>=97 && str[0]<=122)
7       str[0]-=32; //capitalize
8     if(str[0]<'I' || str[0]>'Z')
9       return 2; //invalid char
10    if(strlen(str) != 12)
11      return 3; //wrong length
12    len = strlen(str);
13_0  //@ assert(\valid(str+i));
13    checksum[i]=str[i];
14    for(i=1;i<len;i++){
15_0    //@ assert(\valid(str+i));
15      if(str[i]<48 || str[i]>57)
16        return 4; //not a digit
17_0    //@ assert(0<=i && i<12);
17      checksum[i] = str[i]-48;}
18    sum=0;
19    for(i=1;i<len;i++)
20_0    //@ assert(0<=i && i<12);
20      sum+=checksum[i];
21    while(sum>9)
22      sum=((sum/10)+(sum%10));
23    for(i=0;i<9;i++)
24      if(checksum[0]==c[i][0])
25        break;
26    if(sum!=i)
27      return 5; //wrong checksum
28    return 0;} //OK
```

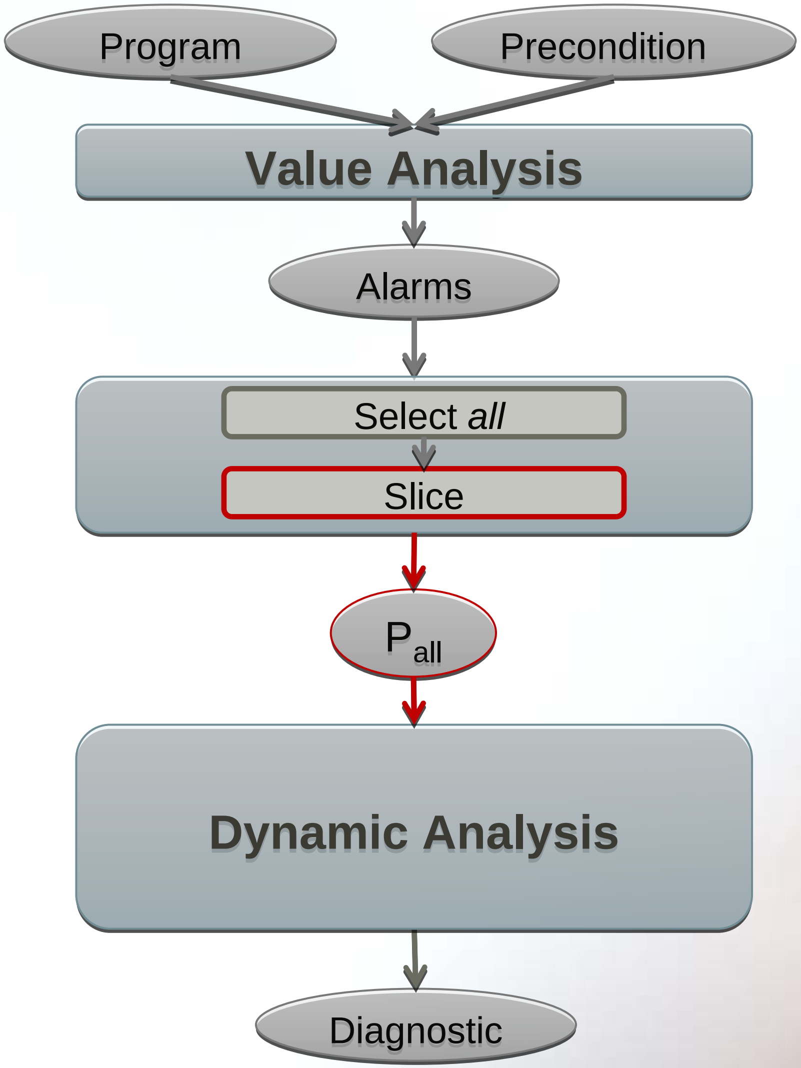
SANTE: option *all*



```
1 int eurocheck(char *str){
2    unsigned char sum;
3    char c[9][3]={"ZQ","YP","XO",
     "WN","VM","UL","TK","SJ","RI"};
4    unsigned char checksum[12];
5    int i = 0, len = 0;
6_0  //@ assert(\valid(str+0));
6    if(str[0]>=97 && str[0]<=122)
7      str[0]-=32; //capitalize
8    if(str[0]<'I' || str[0]>'Z')
9      return 2; //invalid char
10   if(strlen(str) != 12)
11     return 3; //wrong length
12   len = strlen(str);
13_0  //@ assert(\valid(str+i));
13   checksum[i]=str[i];
14   for(i=1;i<len;i++){
15_0    //@ assert(\valid(str+i));
15     if(str[i]<48 || str[i]>57)
16       return 4; //not a digit
17_0    //@ assert(0<=i && i<12);
17     checksum[i] = str[i]-48;}
18   sum=0;
19   for(i=1;i<len;i++)
20_0    //@ assert(0<=i && i<12);
20     sum+=checksum[i];
21   while(sum>9)
22     sum=((sum/10)+(sum%10));
23   for(i=0;i<9;i++)
24     if(checksum[0]==c[i][0])
25       break;
26   if(sum!=i)
27     return 5; //wrong checksum
28   return 0;} //OK
```

Slice w.r.t. the set of all alarms

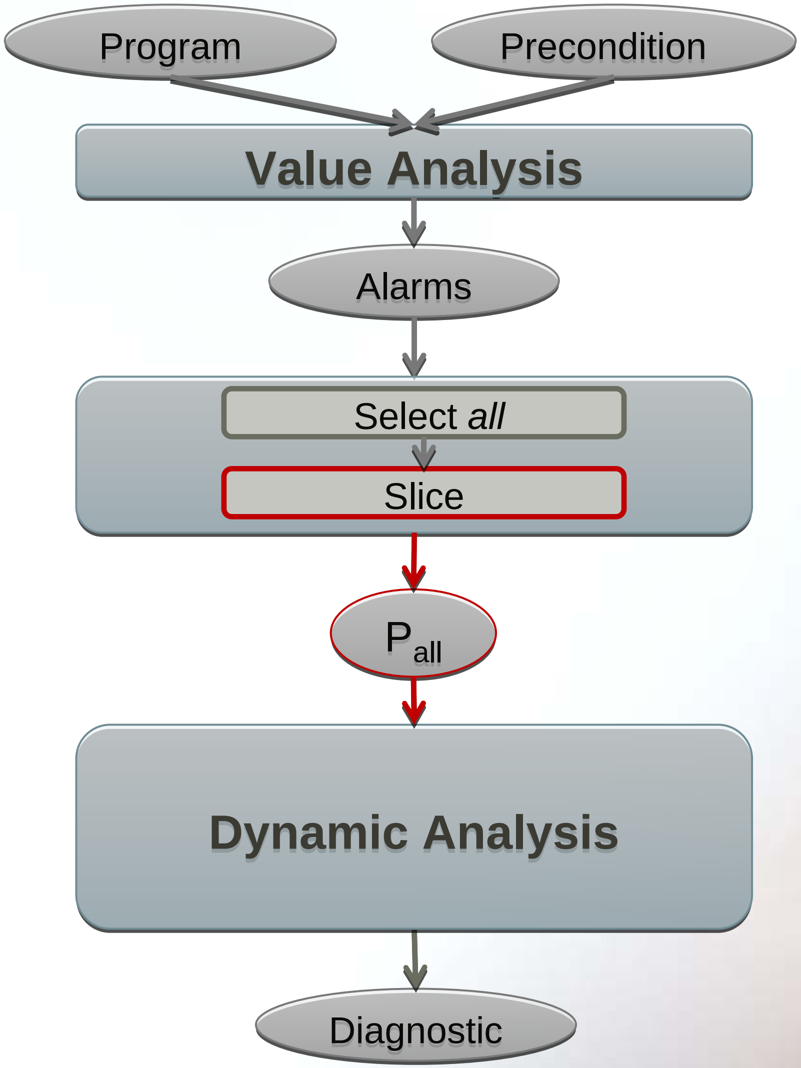
SANTE: option *all*



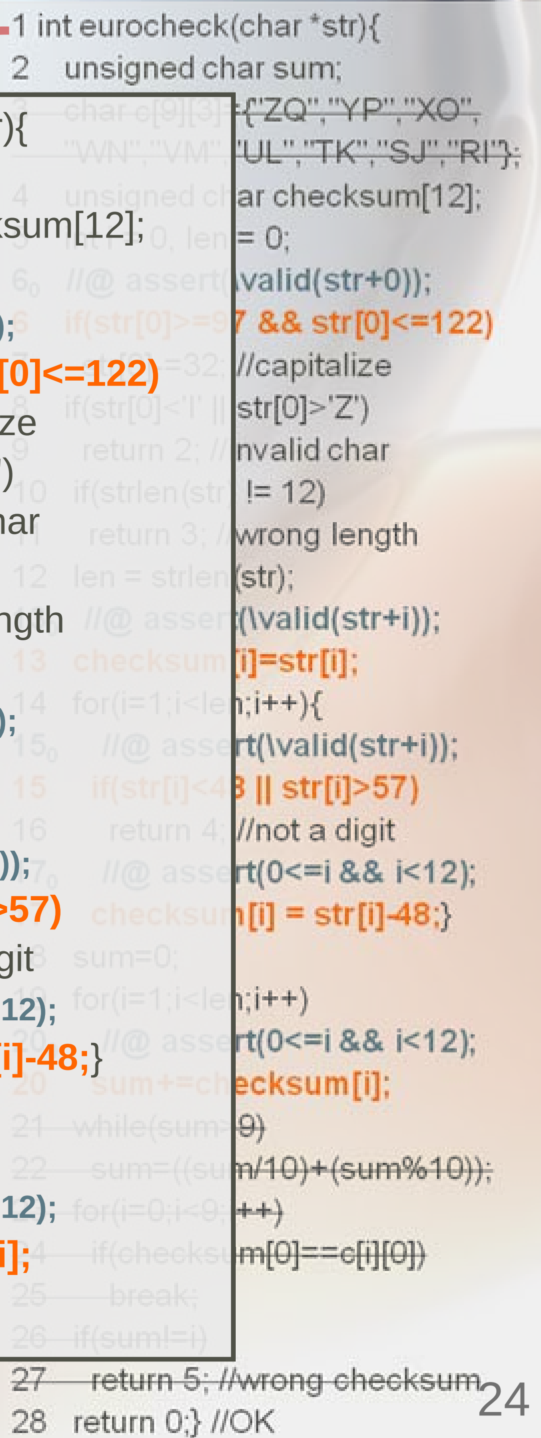
Slice w.r.t. the set of all alarms

```
int eurocheck(char *str){
  unsigned char sum;
  ~~char c[9][3]={"ZQ","YP","XO",~~
  ~~"WN","VM","UL","TK","SJ","RI"};~~
  unsigned char checksum[12];
  int i = 0, len = 0;
  //@ assert(\valid(str+0));
  if(str[0]>=97 && str[0]<=122)
    str[0]-=32; //capitalize
  if(str[0]<'I' || str[0]>'Z')
    return 2; //invalid char
  if(strlen(str) != 12)
    return 3; //wrong length
  len = strlen(str);
  //@ assert(\valid(str+i));
  checksum[i]=str[i];
  for(i=1;i<len;i++){
    //@ assert(\valid(str+i));
    if(str[i]<48 || str[i]>57)
      return 4; //not a digit
    //@ assert(0<=i && i<12);
    checksum[i] = str[i]-48;}
  sum=0;
  for(i=1;i<len;i++)
    //@ assert(0<=i && i<12);
    sum+=checksum[i];
  ~~while(sum>9)~~
    ~~sum=((sum/10)+(sum%10));~~
  ~~for(i=0;i<9;i++)~~
    ~~if(checksum[0]==c[i][0])~~
      ~~break;~~
  ~~if(sum!=i)~~
    ~~return 5; //wrong checksum~~
  return 0;} //OK
```

SANTE: option *all*



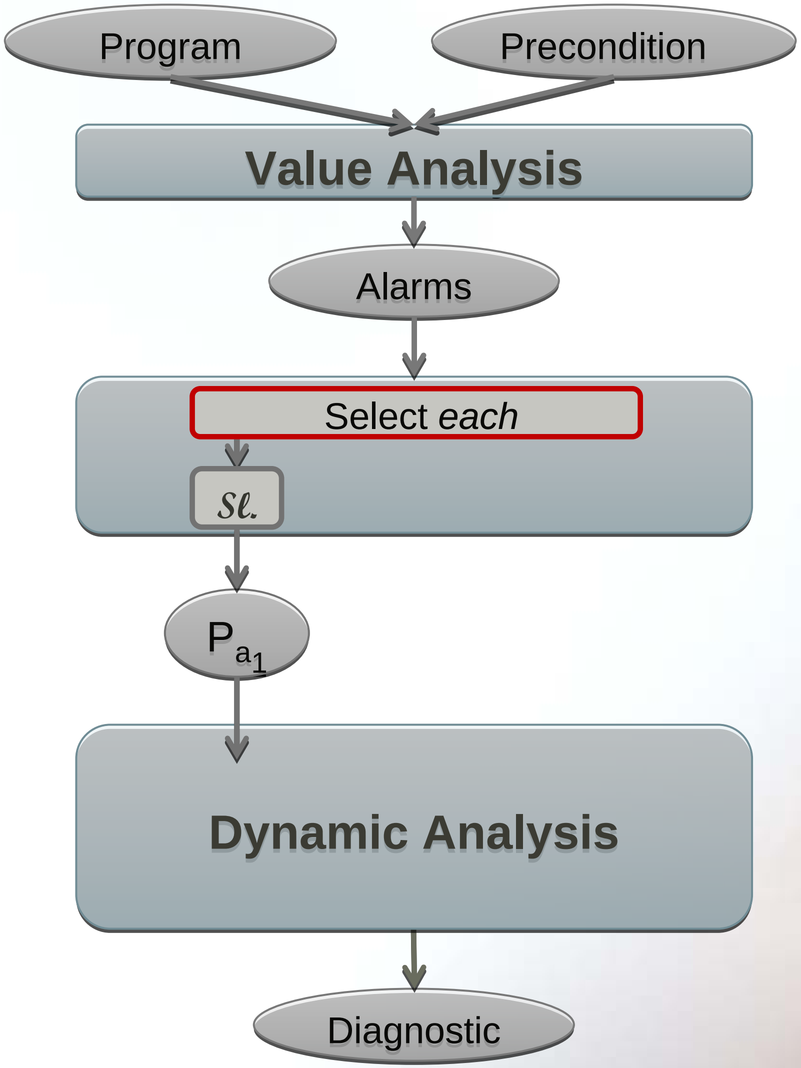
```
int eurocheck(char *str){
  unsigned char sum;
  unsigned char checksum[12];
  int i = 0, len;
 //@ assert(\valid(str+0));
  if(str[0]>=97 && str[0]<=122)
    str[0]-=32; //capitalize
  if(str[0]<'I' || str[0]>'Z')
    return 2; //invalid char
  if(strlen(str) != 12)
    return 3; //wrong length
  len = strlen(str);
  //@ assert(\valid(str+i));
  checksum[i]=str[i];
  for(i=1;i<len;i++){
   //@ assert(\valid(str+i));
    if(str[i]<48 || str[i]>57)
      return 4; //not a digit
    //@ assert(0<=i && i<12);
    checksum[i] = str[i]-48;}
  sum=0;
  for(i=1;i<len;i++)
    //@ assert(0<=i && i<12);
    sum+=checksum[i];
  return 0;} //OK
```



```
int eurocheck(char *str){
  unsigned char sum;
  ~~char c[9][3]={"ZQ","YP","XO",~~
  ~~"WN","VM","UL","TK","SJ","RI"};~~
  unsigned char checksum[12];
  int i = 0, len = 0;
  //@ assert(\valid(str+0));
  if(str[0]>=97 && str[0]<=122)
    str[0]-=32; //capitalize
  if(str[0]<'I' || str[0]>'Z')
    return 2; //invalid char
  if(strlen(str) != 12)
    return 3; //wrong length
  len = strlen(str);
  //@ assert(\valid(str+i));
  checksum[i]=str[i];
  for(i=1;i<len;i++){
    //@ assert(\valid(str+i));
    if(str[i]<48 || str[i]>57)
      return 4; //not a digit
    //@ assert(0<=i && i<12);
    checksum[i] = str[i]-48;}
  sum=0;
  for(i=1;i<len;i++)
    //@ assert(0<=i && i<12);
    sum+=checksum[i];
  ~~while(sum>9)~~
    ~~sum=((sum/10)+(sum%10));~~
  ~~for(i=0;i<9;i++)~~
    ~~if(checksum[0]==c[i][0])~~
      ~~break;~~
  ~~if(sum!=i)~~
    ~~return 5; //wrong checksum~~
  return 0;} //OK
```

Slice w.r.t. the set of all alarms

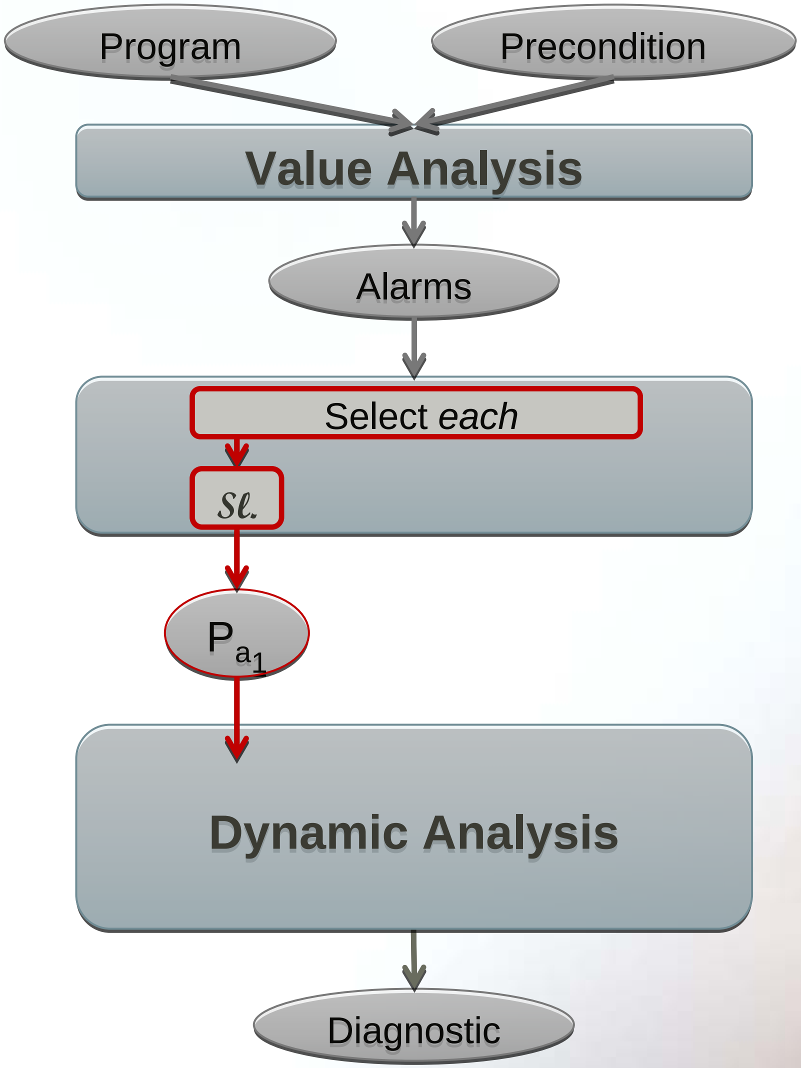
SANTE: option *each*



Slice w.r.t. each alarm

```
int eurocheck(char *str){
  unsigned char sum;
  char c[9][3]={"ZQ","YP","XO",
  "WN","VM","UL","TK","SJ","RI"};
  unsigned char checksum[12];
  int i = 0, len = 0;
  //@ assert(\valid(str+0));
  if(str[0]>=97 && str[0]<=122)
    str[0]-=32; //capitalize
  if(str[0]<'I' || str[0]>'Z')
    return 2; //invalid char
  if(strlen(str) != 12)
    return 3; //wrong length
  len = strlen(str);
  //@ assert(\valid(str+i));
  checksum[i]=str[i];
  for(i=1;i<len;i++){
    //@ assert(\valid(str+i));
    if(str[i]<48 || str[i]>57)
      return 4; //not a digit
    //@ assert(0<=i && i<12);
    checksum[i] = str[i]-48;}
  sum=0;
  for(i=1;i<len;i++)
    //@ assert(0<=i && i<12);
    sum+=checksum[i];
  while(sum>9)
    sum=((sum/10)+(sum%10));
  for(i=0;i<9;i++)
    if(checksum[0]==c[i][0])
      break;
  if(sum!=i)
    return 5; //wrong checksum
  return 0;} //OK
```

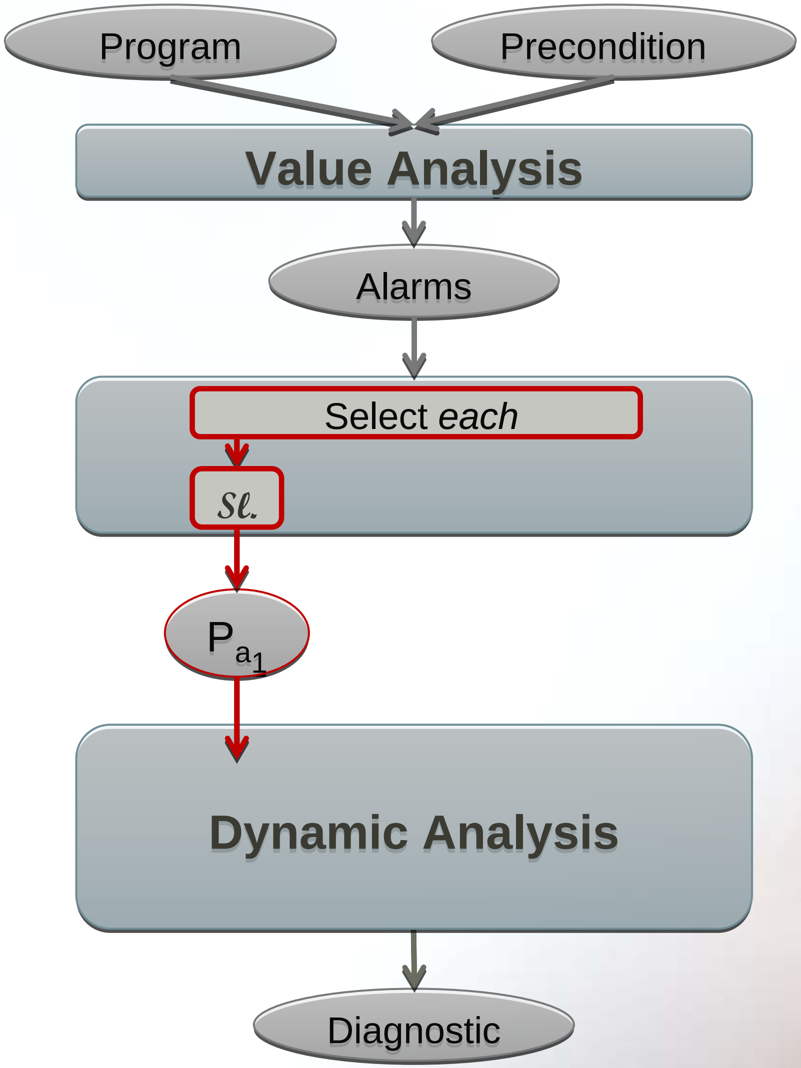
SANTE: option *each*



```
1 int eurocheck(char *str){
~~2   unsigned char sum;~~
~~3   char c[9][3]={"ZQ","YP","XO",~~
~~      "WN","VM","UL","TK","SJ","RI"};~~
~~4   unsigned char checksum[12];~~
~~5   int i = 0, len = 0;~~
6₀   //@ assert(\valid(str+0));
6    if(str[0]>=97 && str[0]<=122)
7      str[0]-=32; //capitalize
~~8   if(str[0]<'I' || str[0]>'Z')~~
~~9     return 2; //invalid char~~
~~10  if(strlen(str) != 12)~~
~~11    return 3; //wrong length~~
~~12  len = strlen(str);~~
~~13₀ //@ assert(\valid(str+i));~~
~~13  checksum[i]=str[i];~~
~~14  for(i=1;i<len;i++){~~
~~15₀   //@ assert(\valid(str+i));~~
~~15    if(str[i]<48 || str[i]>57)~~
~~16      return 4; //not a digit~~
~~17₀   //@ assert(0<=i && i<12);~~
~~17    checksum[i] = str[i]-48;}~~
~~18  sum=0;~~
~~19  for(i=1;i<len;i++)~~
~~20₀   //@ assert(0<=i && i<12);~~
~~20    sum+=checksum[i];~~
~~21  while(sum>9)~~
~~22    sum=((sum/10)+(sum%10));~~
~~23  for(i=0;i<9;i++)~~
~~24    if(checksum[0]==c[i][0])~~
~~25      break;~~
~~26  if(sum!=i)~~
~~27    return 5; //wrong checksum~~
28  return 0;} //OK
```

Slice w.r.t. each alarm

SANTE: option *each*

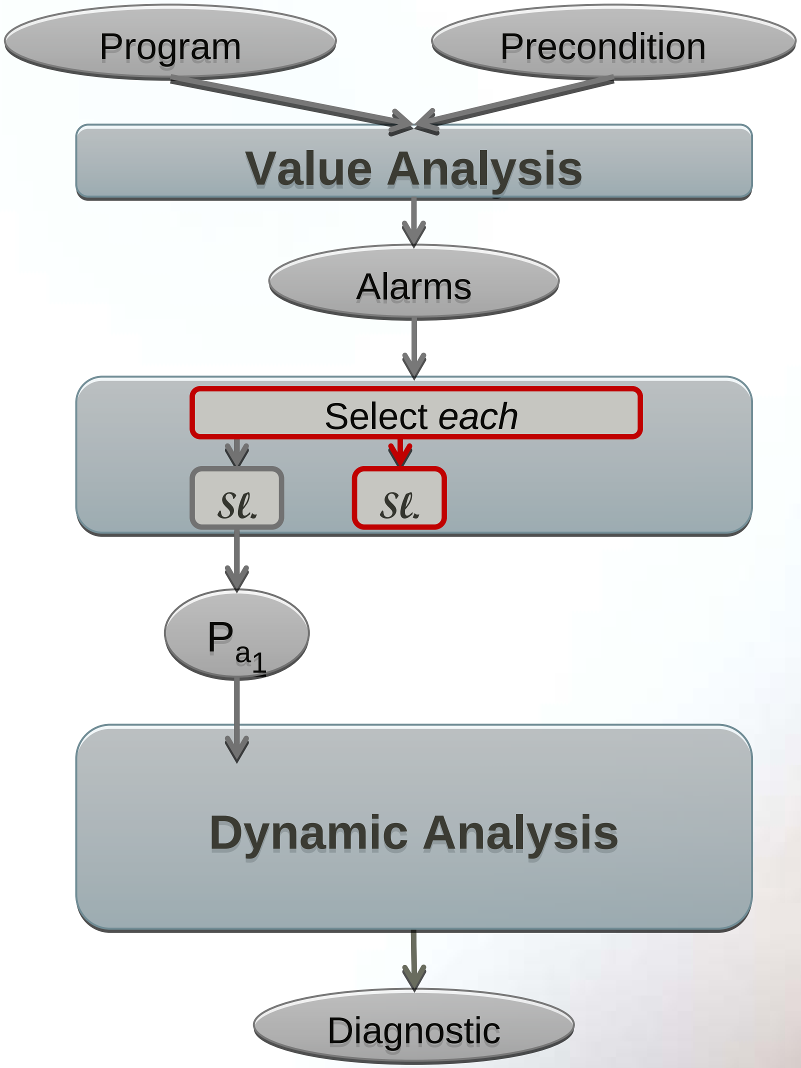


```
int eurocheck(char *str){
  if(str[0]>=97 && str[0]<=122)
    str[0]-=32; //capitalize
return 0;}
```

```
int eurocheck(char *str){
~~unsigned char sum;~~
~~char c[9][3]={"ZQ","YP","XO",~~
~~"WN","VM","UL","TK","SJ","RI"};~~
~~unsigned char checksum[12];~~
~~int i = 0, len = 0;~~
//@ assert(\valid(str+0));
if(str[0]>=97 && str[0]<=122)
  str[0]-=32; //capitalize
~~if(str[0]<'A' || str[0]>'Z')~~
~~  return 2; //invalid char~~
~~if(strlen(str) != 12)~~
~~  return 3; //wrong length~~
~~len = strlen(str);~~
~~//@ assert(\valid(str+i));~~
~~checksum[i]=str[i];~~
~~for(i=1;i<len;i++){~~
~~  //@ assert(\valid(str+i));~~
~~  if(str[i]<48 || str[i]>57)~~
~~    return 4; //not a digit~~
~~  //@ assert(0<=i && i<12);~~
~~  checksum[i] = str[i]-48;}~~
~~sum=0;~~
~~for(i=1;i<len;i++)~~
~~  //@ assert(0<=i && i<12);~~
~~  sum+=checksum[i];~~
~~while(sum>9)~~
~~  sum=((sum/10)+(sum%10));~~
~~for(i=0;i<9;i++)~~
~~  if(checksum[0]==c[i][0])~~
~~    break;~~
~~if(sum!=i)~~
~~  return 5; //wrong checksum~~
return 0;} //OK
```

Slice w.r.t. each alarm

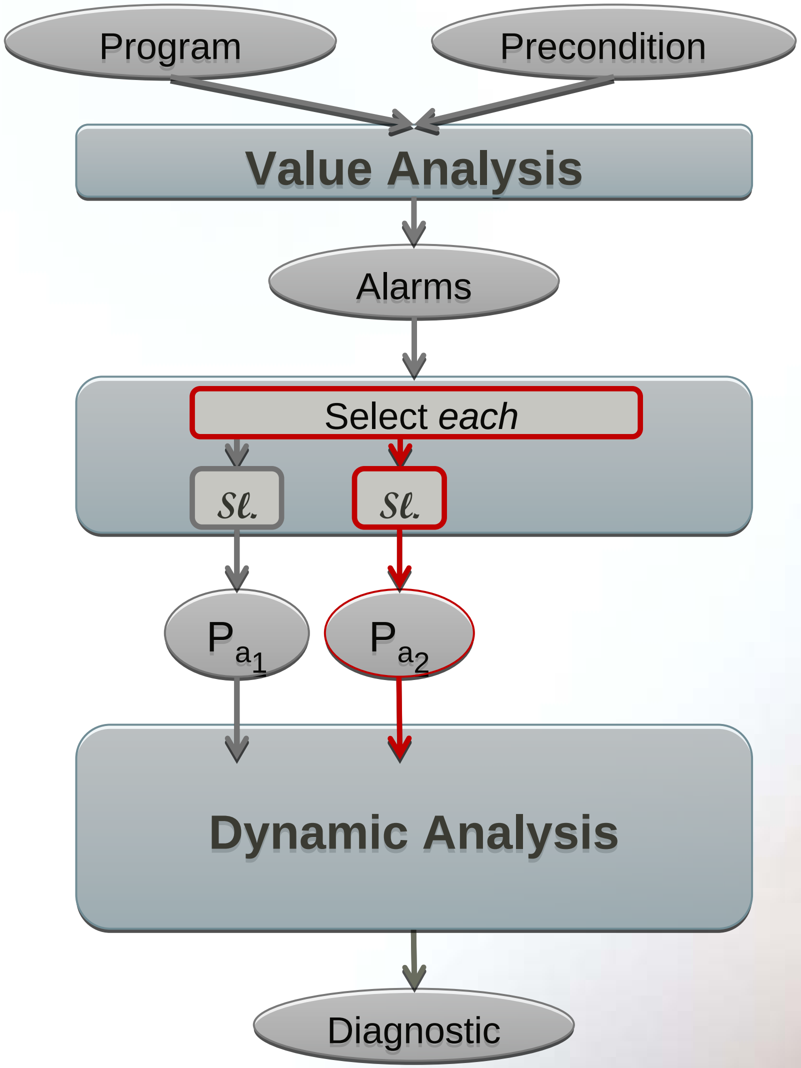
SANTE: option *each*



Slice w.r.t. each alarm

```
1  int eurocheck(char *str){
2    unsigned char sum;
3    char c[9][3]={"ZQ","YP","XO",
     "WN","VM","UL","TK","SJ","RI"};
4    unsigned char checksum[12];
5    int i = 0, len = 0;
6_0  //@ assert(\valid(str+0));
6    if(str[0]>=97 && str[0]<=122)
7      str[0]-=32; //capitalize
8    if(str[0]<'I' || str[0]>'Z')
9      return 2; //invalid char
10   if(strlen(str) != 12)
11     return 3; //wrong length
12   len = strlen(str);
13_0 //@ assert(\valid(str+i));
13   checksum[i]=str[i];
14   for(i=1;i<len;i++){
15_0   //@ assert(\valid(str+i));
15     if(str[i]<48 || str[i]>57)
16       return 4; //not a digit
17_0   //@ assert(0<=i && i<12);
17     checksum[i] = str[i]-48;}
18   sum=0;
19   for(i=1;i<len;i++)
20_0   //@ assert(0<=i && i<12);
20     sum+=checksum[i];
21   while(sum>9)
22     sum=((sum/10)+(sum%10));
23   for(i=0;i<9;i++)
24     if(checksum[0]==c[i][0])
25       break;
26   if(sum!=i)
27     return 5; //wrong checksum
28   return 0;} //OK
```

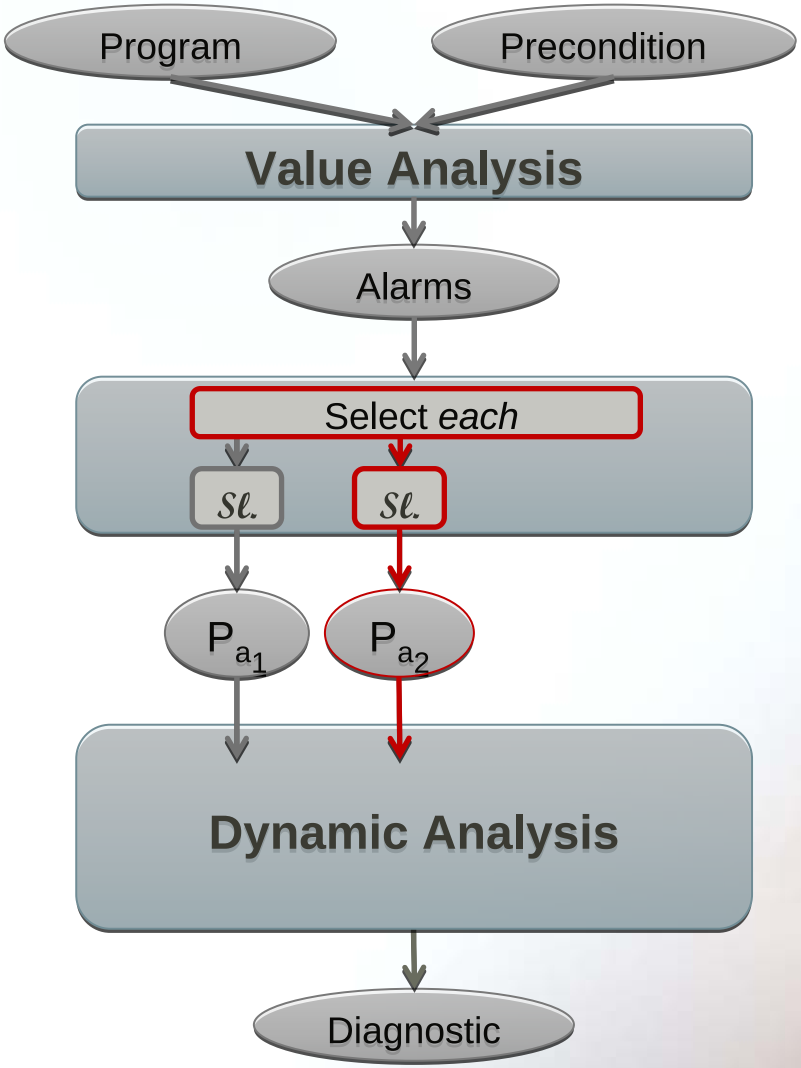
SANTE: option *each*



```
int eurocheck(char *str){
  ~~unsigned char sum;~~
  ~~char c[9][3]={"ZQ","YP","XO",~~
  ~~"WN","VM","UL","TK","SJ","RI"};~~
  unsigned char checksum[12];
  int i = 0, len = 0;
  //@ assert(\valid(str+0));
  if(str[0]>=97 && str[0]<=122)
    str[0]-=32; //capitalize
  if(str[0]<'I' || str[0]>'Z')
    return 2; //invalid char
  if(strlen(str) != 12)
    return 3; //wrong length
  ~~len = strlen(str);~~
  //@ assert(\valid(str+i));
  checksum[i]=str[i];
  ~~for(i=1;i<len;i++){~~
    ~~//@ assert(\valid(str+i));~~
    ~~if(str[i]<48 || str[i]>57)~~
      ~~return 4; //not a digit~~
    ~~//@ assert(0<=i && i<12);~~
    ~~checksum[i] = str[i]-48;}~~
  ~~sum=0;~~
  ~~for(i=1;i<len;i++)~~
    ~~//@ assert(0<=i && i<12);~~
    ~~sum+=checksum[i];~~
  ~~while(sum>9)~~
    ~~sum=((sum/10)+(sum%10));~~
  ~~for(i=0;i<9;i++)~~
    ~~if(checksum[0]==c[i][0])~~
      ~~break;~~
  ~~if(sum!=i)~~
    ~~return 5; //wrong checksum~~
  return 0;} //OK
```

Slice w.r.t. each alarm

SANTE: option *each*

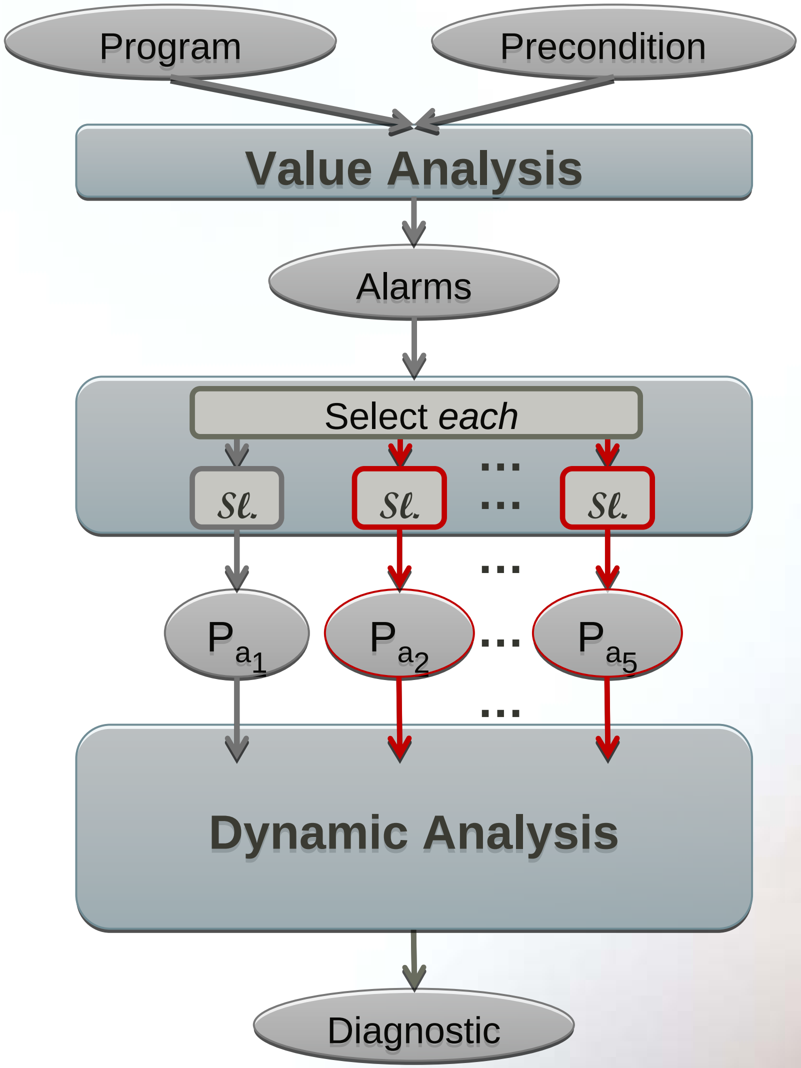


```
int eurocheck(char *str){
  unsigned char checksum[12];
  int i = 0;
 if(str[0]>=97 && str[0]<=122)
   str[0]-=32;
  if(str[0]<'I' || str[0]>'Z')
   return 2;
  if(strlen(str) != 12)
   return 3;
 checksum[i]=str[i];
  return 0;} //OK
```

```
1 int eurocheck(char *str){
~~2   unsigned char sum;~~
~~3   char c[9][3]={"ZQ","YP","XO",~~
~~      "WN","VM","UL","TK","SJ","RI"};~~
4   unsigned char checksum[12];
5   int i = 0, len = 0;
6_0  //@ assert(\valid(str+0));
6   if(str[0]>=97 && str[0]<=122)
7      str[0]-=32; //capitalize
8   if(str[0]<'I' || str[0]>'Z')
9      return 2; //invalid char
10  if(strlen(str) != 12)
11     return 3; //wrong length
~~12  len = strlen(str);~~
13_0   //@ assert(\valid(str+i));
13  checksum[i]=str[i];
~~14  for(i=1;i<len;i++){~~
~~15_0    //@ assert(\valid(str+i));~~
~~15    if(str[i]<48 || str[i]>57)~~
~~16      return 4; //not a digit~~
~~17_0    //@ assert(0<=i && i<12);~~
~~17    checksum[i] = str[i]-48;}~~
~~18  sum=0;~~
~~19  for(i=1;i<len;i++)~~
~~20_0    //@ assert(0<=i && i<12);~~
~~20    sum+=checksum[i];~~
~~21  while(sum>9)~~
~~22    sum=((sum/10)+(sum%10));~~
~~23  for(i=0;i<9;i++)~~
~~24    if(checksum[0]==c[i][0])~~
~~25      break;~~
~~26  if(sum!=i)~~
~~27    return 5; //wrong checksum~~
28  return 0;} //OK
```

Slice w.r.t. each alarm

SANTE: option *each*



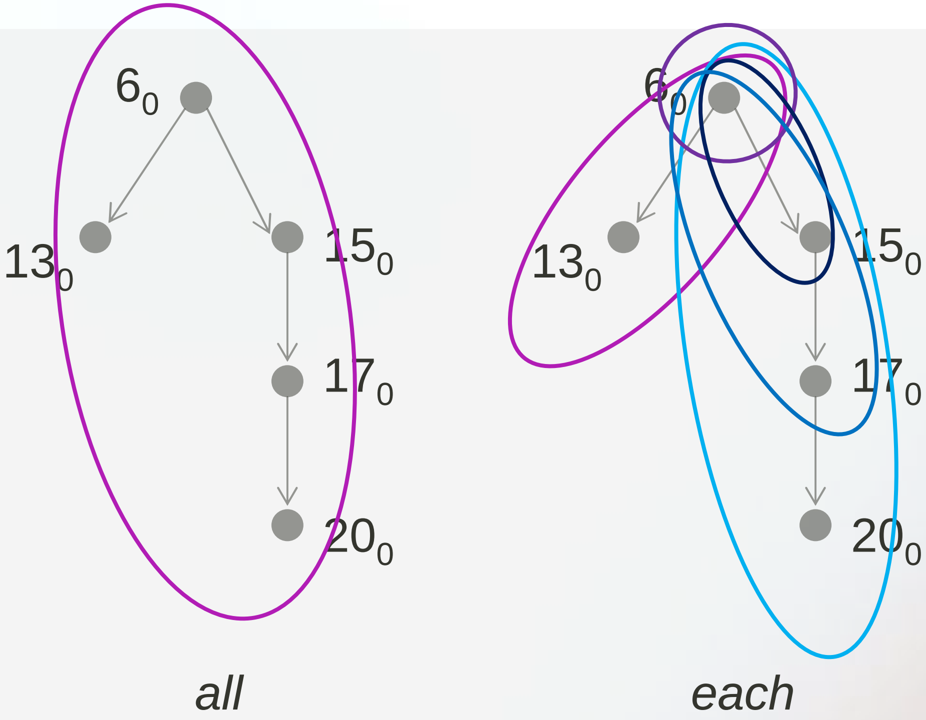
Slice w.r.t. each alarm

```
int eurocheck(char *str){
  unsigned char sum;
  char c[9][3]={"ZQ","YP","XO",
  "WN","VM","UL","TK","SJ","RI"};
  unsigned char checksum[12];
  int i = 0, len = 0;
  //@ assert(\valid(str+0));
  if(str[0]>=97 && str[0]<=122)
    str[0]-=32; //capitalize
  if(str[0]<'I' || str[0]>'Z')
    return 2; //invalid char
  if(strlen(str) != 12)
    return 3; //wrong length
  len = strlen(str);
  //@ assert(\valid(str+i));
  checksum[i]=str[i];
  for(i=1;i<len;i++){
    //@ assert(\valid(str+i));
    if(str[i]<48 || str[i]>57)
      return 4; //not a digit
    //@ assert(0<=i && i<12);
    checksum[i] = str[i]-48;}
  sum=0;
  for(i=1;i<len;i++)
    //@ assert(0<=i && i<12);
    sum+=checksum[i];
  while(sum>9)
    sum=((sum/10)+(sum%10));
  for(i=0;i<9;i++)
    if(checksum[0]==c[i][0])
      break;
  if(sum!=i)
    return 5; //wrong checksum
  return 0;} //OK
```

Outline

- Context and objectives
- SANTE: Basic options
- **SANTE: Advanced options**
- Experiments
- Conclusion & Perspectives

Alarms dependencies



17_0 *//@* **assert(0<=i && i<12);**

17 **checksum[i] = str[i]-48;**}

18 sum=0;

19 for(i=1;i<len;i++)

20_0 *//@* **assert(0<=i && i<12);**

20 **sum+=checksum[i];**

The alarms being dependent, it is redundant to consider them one by one

Alarm dependencies

- $a \rightsquigarrow a'$ (a' depends de a)
- $a \rightsquigarrow a'$ => a is preserved in the slice $p_{a'}$

- e is an end alarm:

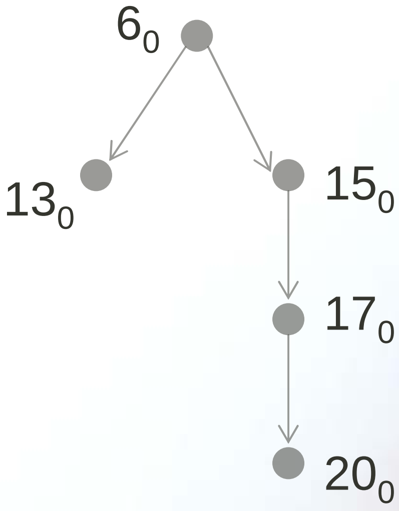
  $\forall s \in A, \qquad e \rightsquigarrow s \;\Rightarrow\; s \rightsquigarrow e$

- A' defines a slicing-induced cover of A if

  $$A = \bigcup_{a \in A'} labels(p_a) \cap A$$

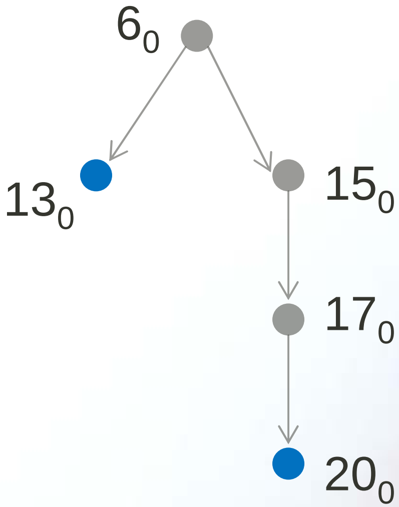
- There is a unique minimal slicing-induced cover. Each covering set contains a representative end alarm in each equivalence class of end alarms

Alarm dependencies



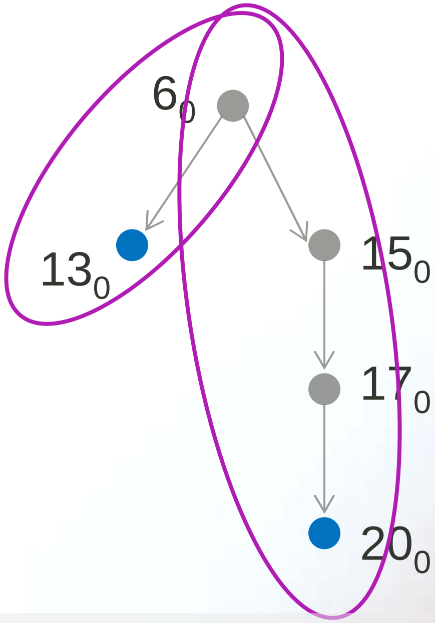
- 13_0 and 20_0 are end alarms
- $\{13_0, 20_0\}$ defines a minimal-slicing induced cover
- $\{6_0, 13_0\}$ and $\{6_0, 15_0, 17_0, 20_0\}$ are the covering sets

Alarm dependencies



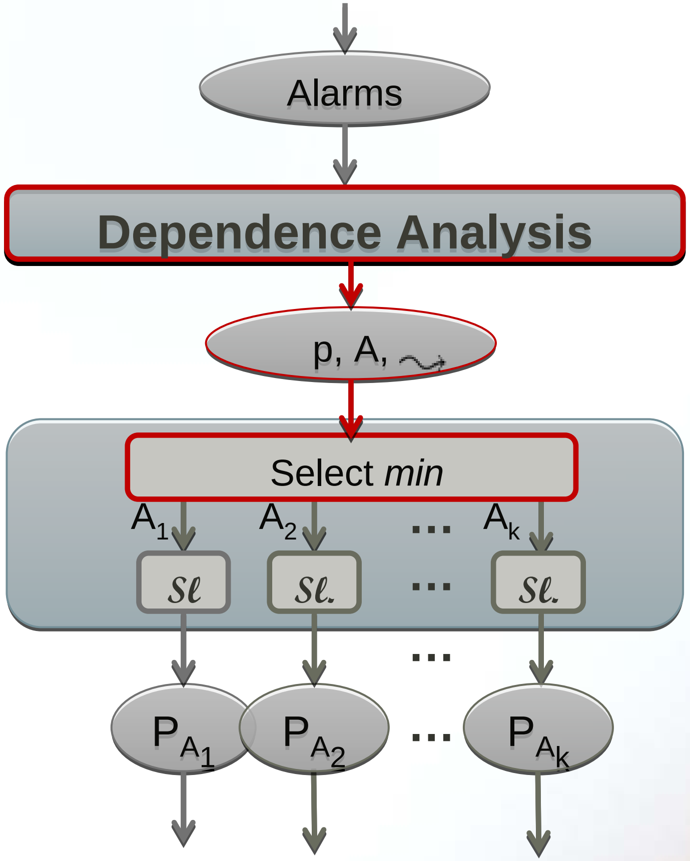
- 13_0 and 20_0 are end alarms
- $\{13_0, 20_0\}$ defines a minimal-slicing induced cover
- $\{6_0, 13_0\}$ and $\{6_0, 15_0, 17_0, 20_0\}$ are the covering sets

Alarm dependencies



- 13_0 and 20_0 are end alarms
- $\{13_0, 20_0\}$ defines a minimal-slicing induced cover
- $\{6_0, 13_0\}$ and $\{6_0, 15_0, 17_0, 20_0\}$ are the covering sets

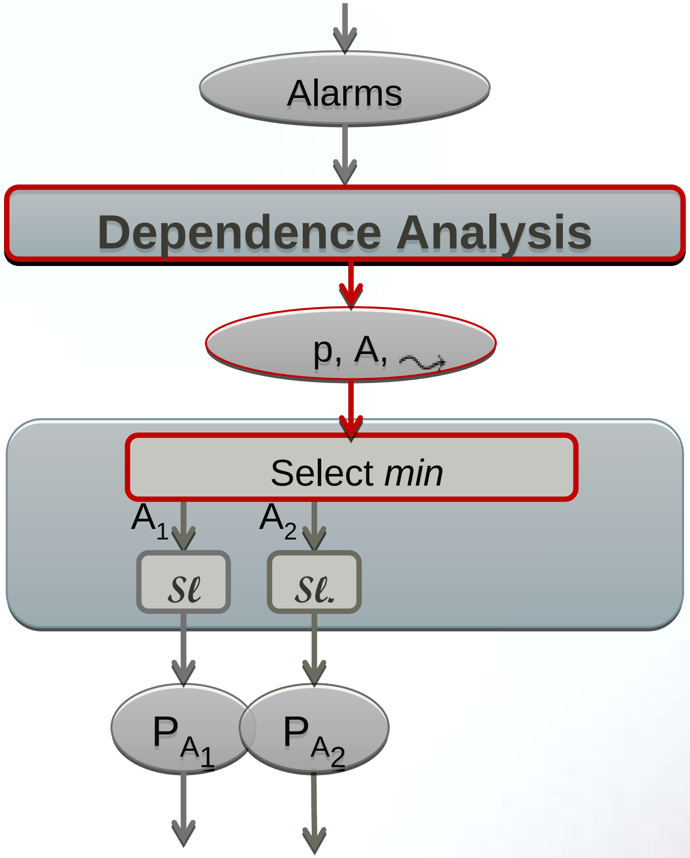
SANTE: option *min*

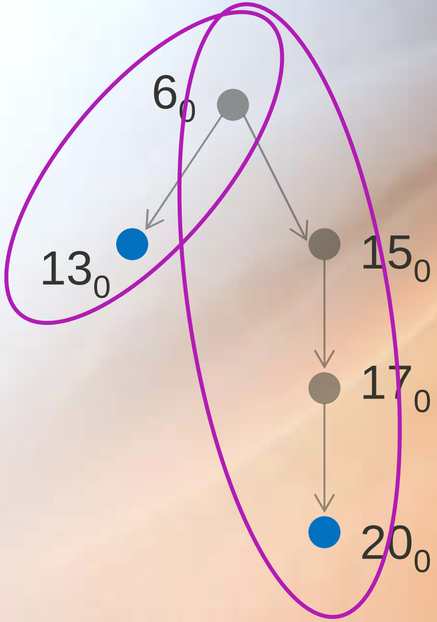


SANTE: option *min*

$A_1 = \{6_0, 13_0\}$

$A_2 = \{6_0, 15_0, 17_0, 20_0\}$

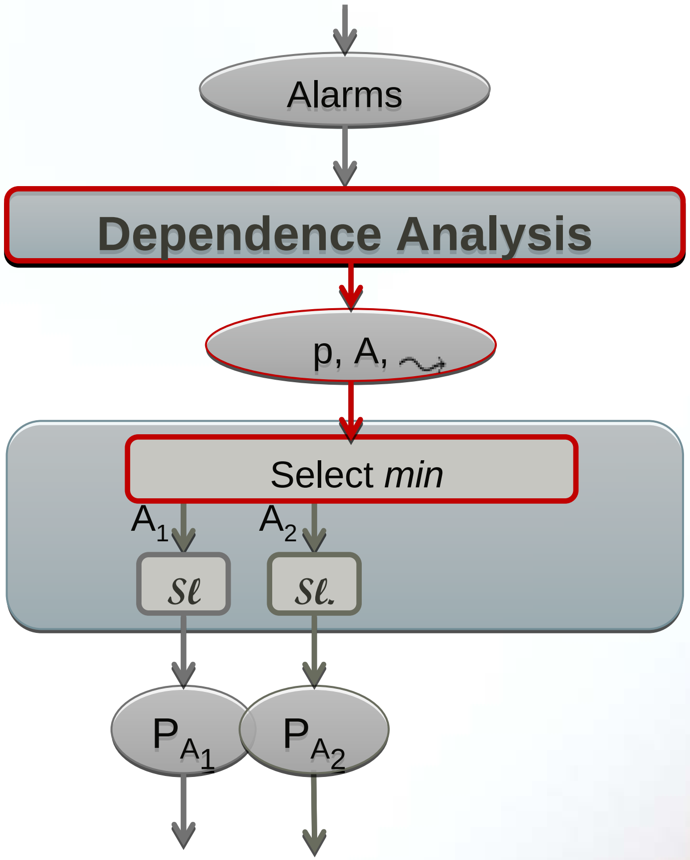


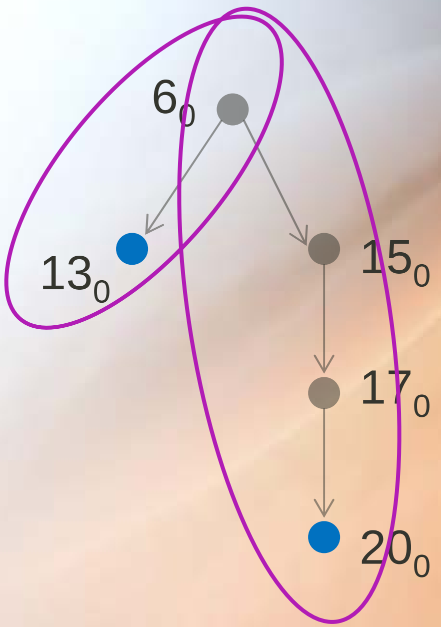


SANTE: option *min*

$A_1 = \{6_0, 13_0\}$

$A_2 = \{6_0, 15_0, 17_0, 20_0\}$





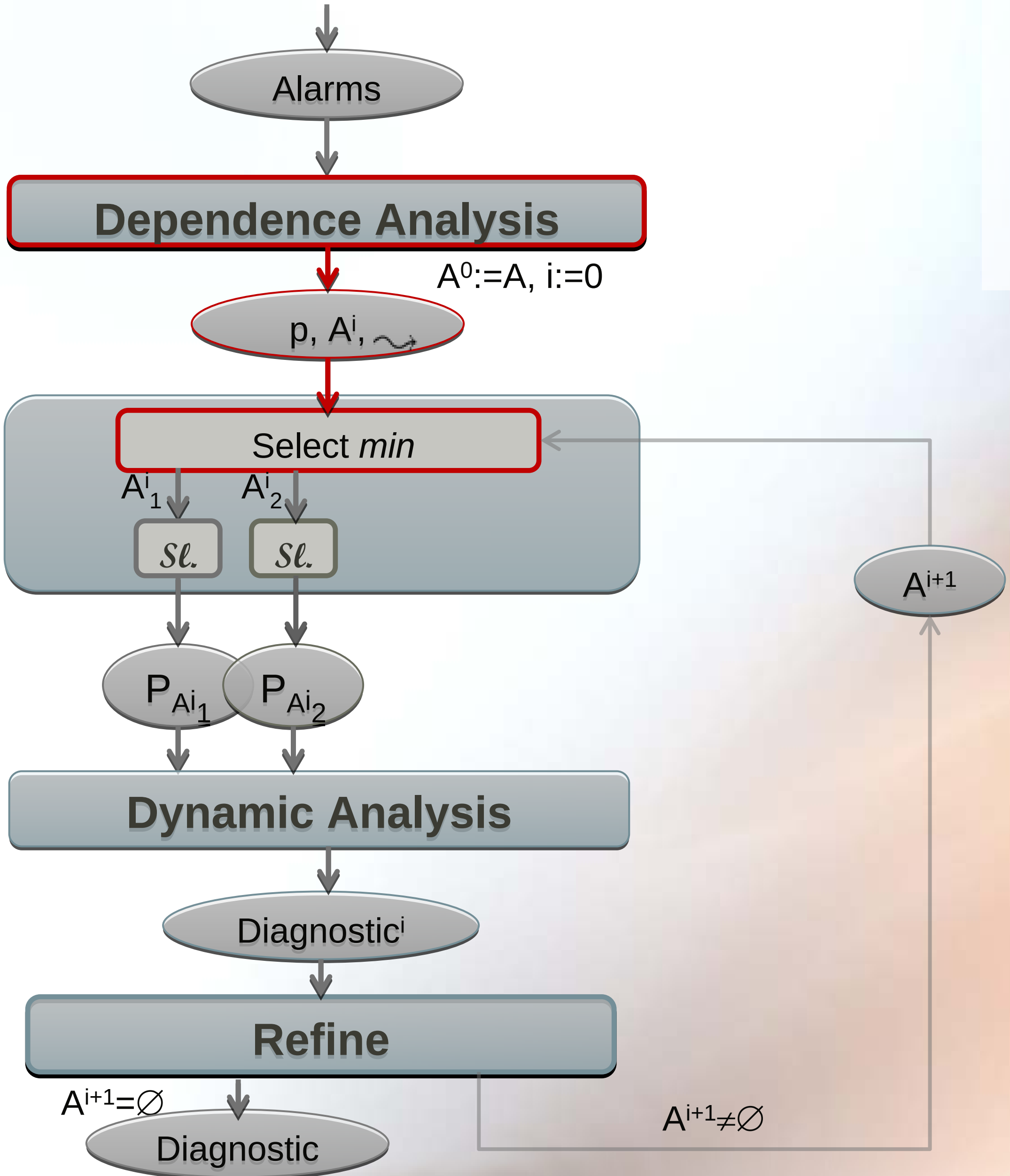
We can improve the classification considering smaller slices, removing alarms one by one starting from the root.

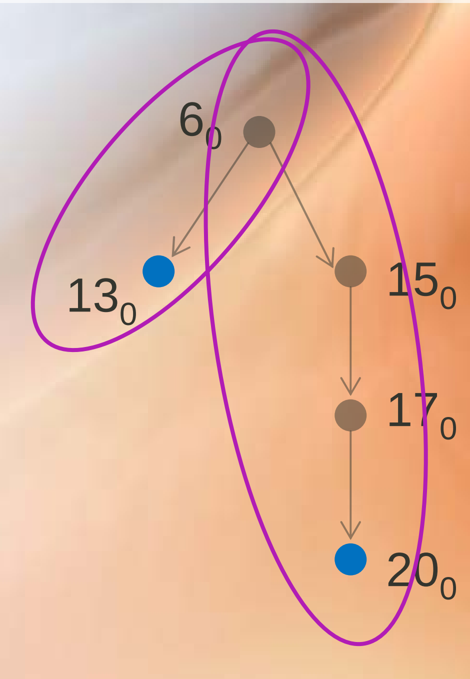
SANTE: option *smart*

$A^0 = \{6_0, 13_0, 15_0, 17_0, 20_0\}$

$A^0_1 = \{6_0, 13_0\}$

$A^0_2 = \{6_0, 15_0, 17_0, 20_0\}$





SANTE: option *smart*

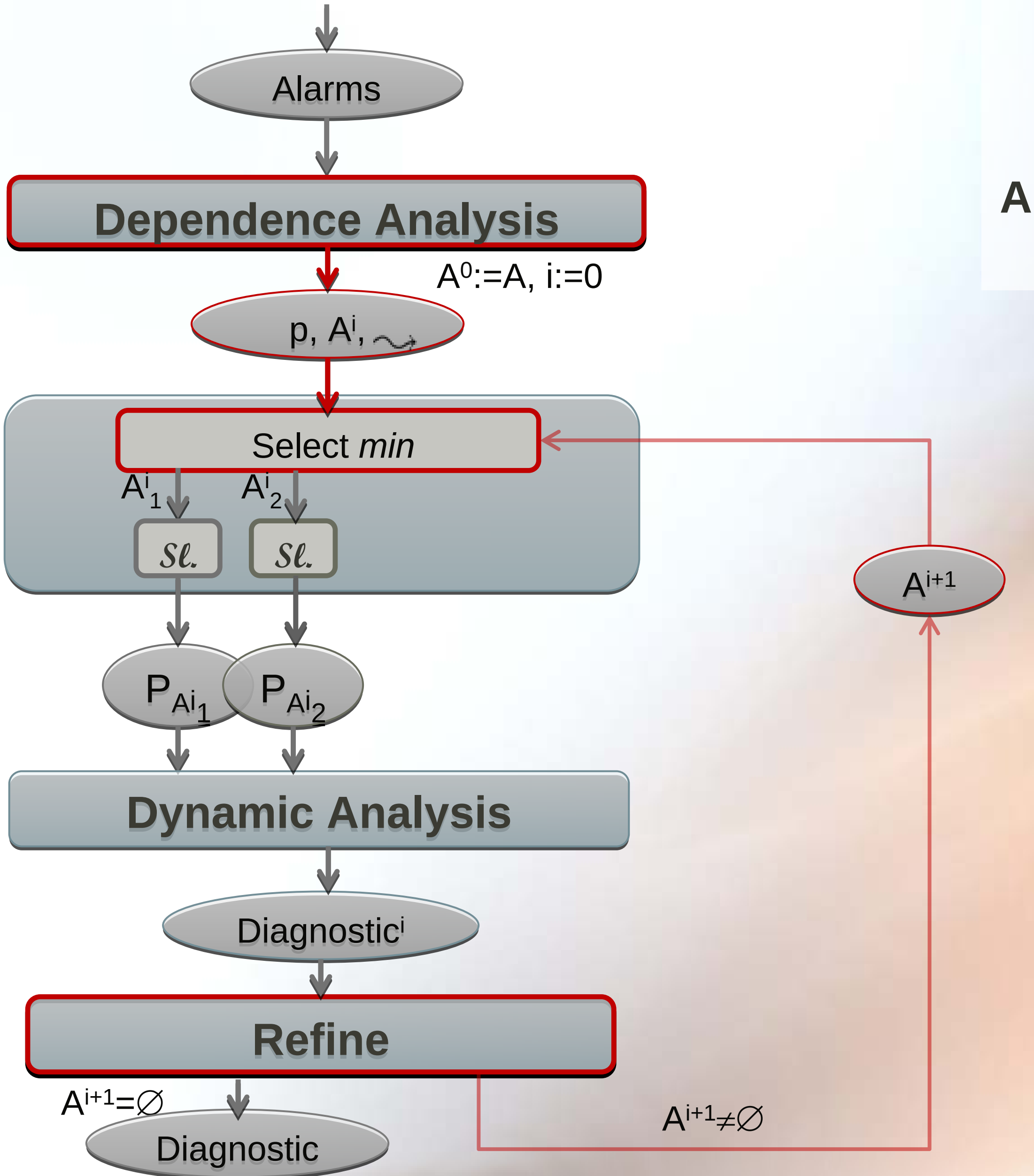
$A^0 = \{6_0, 13_0, 15_0, 17_0, 20_0\}$

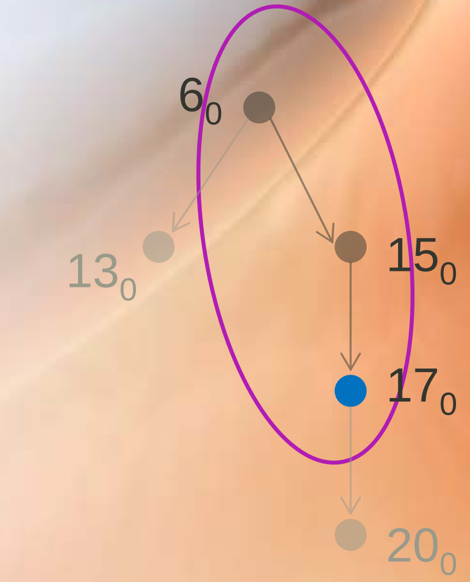
$A^0_1 = \{6_0, 13_0\}$

$A^0_2 = \{6_0, 15_0, 17_0, 20_0\}$

$A^1 = \{6_0, 15_0, 17_0\}$

$A^1_1 = \{6_0, 15_0, 17_0\}$





Outline

- Context and objectives
- SANTE: Basic options
- SANTE: Advanced options
- **Experiments**
- Conclusion & Perspectives

Experimental condition

- Experiments on real-life programs
- Timeout of dynamic analysis of 1 slice = 10 minutes
- 85 alarms in total

	Origin	Function	Size	Alarms	Bugs
1	libgd	gdImageStringFTEx	705	15	1
2	Apache	get_tag	696	12	3
3	polygon	main	202	29	2
4	rawcaudio	adpcm decoder	365	10	0
5	eurocheck	main	154	19	1

Classification of alarms

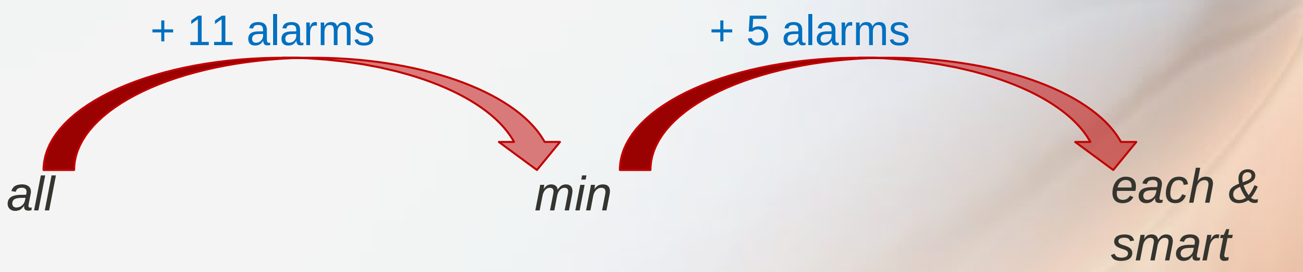
- The combined SANTE method gives better results than each technique used alone

SANTE vs *VA*	**SANTE vs *DA***
+ 19 alarms classified	+ 11 alarms classified
• Confirms some alarms as real bugs • Provides input states leading to the errors	• Terminates in some cases where DA times-out

- In the worst case, SANTE classifies as much alarms as each method

Classification of alarms

- Different usages of program slicing enhance classification



+ 11 alarms

+ 5 alarms

all

min

each & smart

  - with *each* and *smart* 6 alarms remain unclassified over 85
  - All known bugs are detected
  - SANTE *each* and SANTE *smart* give the best classification results

Analysis time

- *With slicing*
  - SANTE detects the same number of bugs in less time
  - The time savings can avoid the timeout in some cases
- *smart* is the best on these examples

	none	*all*	*each*	*min*	*smart*
1	TO	TO	1h 32min 52s	32min 16s	32min 16s
2	TO	TO	**3min 24s + 5 TO**	1 TO	1TO +54s
3	1min 31s	1min 20s	7s	7s	7s
5	18s	7s	13s	6s	6s

Best classification

Best classification and time

Simpler counter-examples

- Errors and alarms are reported with more precise information.
  - o path length of counter-examples

	none	all	each	min	smart
3	526	525	153	153	153
5	6	6	6	6	6

  - o The path length in counter examples diminishes on average by 24%. This rate goes up to 71%.

Program reduction

- Errors and alarms are reported with more precise information.
  - Average rate (t_{avg}) of program reduction:

	none **(en lignes)**	***all*** **(en lignes)**	***each*** **(en lignes)**	***min*** **(en lignes)**	***smart*** **(en lignes)**
3	179	96	10 slices 20 --- 34	10 slices 20 --- 34	10 slices 20 --- 34
5	124	74	5 slices 20 --- 62	20, 62	20, 62
$\mathbf{t_{avg}}$		-24%	-51%	-51%	-51%

  - This rate goes up to 97% for some alarms.

Outline

- Context and objectives
- SANTE: Basic options
- SANTE: Advanced options
- Experiments
- **Conclusion & Perspectives**

Related work

- Synergy / Dash - BLAST: verification of properties by static analysis and test generation
- Daikon: uses the test generation to generate invariants candidates
- Check ’n’ Crash, DSD-Crasher: combines 3 steps: dynamic inference, static analysis and dynamic verification
- EXE / Active Property Checking: tests all potential threats
- DyTa: combines static and dynamic analysis, removes irrelevant branches before test generation
- …

No other methods combine value analysis, program slicing and test generation

Conclusion

- **SANTE: Combines static analysis, program slicing and structural testing**
  - **More** precise **than a static analyzer**
  - **More** efficient **than a concolic testing tool**
  - Automatic**: human interference not needed**

- **Four uses of program slicing:**
  - Faster
  - Less **unclassified alarms**
  - Simplified counter-examples

Perspectives

- **Test other configurations of analysis techniques**
  - **Precision of value analysis**
  - **Selection of slices**
  - **All-branch test generation**
- **Compare SANTE with other tools**
- **Handle other class of alarms: invalid pointer, overflow, shifting, etc.**

References

- O. Chebaro, N. Kosmatov, A. Giorgetti, J. Julliand, **Combining static analysis and test generation for C program debugging** *Proc. of the 4th International Conference on Tests & Proofs* (TAP 2010).
- O. Chebaro, N. Kosmatov, A. Giorgetti, J. Julliand, **The SANTE Tool: Value Analysis, Program Slicing and Test Generation for C program debugging** Proc. of the 5th International Conference on Tests & Proofs (TAP 2011).
- O. Chebaro, N. Kosmatov, A. Giorgetti, J. Julliand, **How to Integrate Program Slicing into a Verification Technique Combining Static and Dynamic Analysis** Proc. of the 27th Symposium on Applied Computing (SAC 2012).