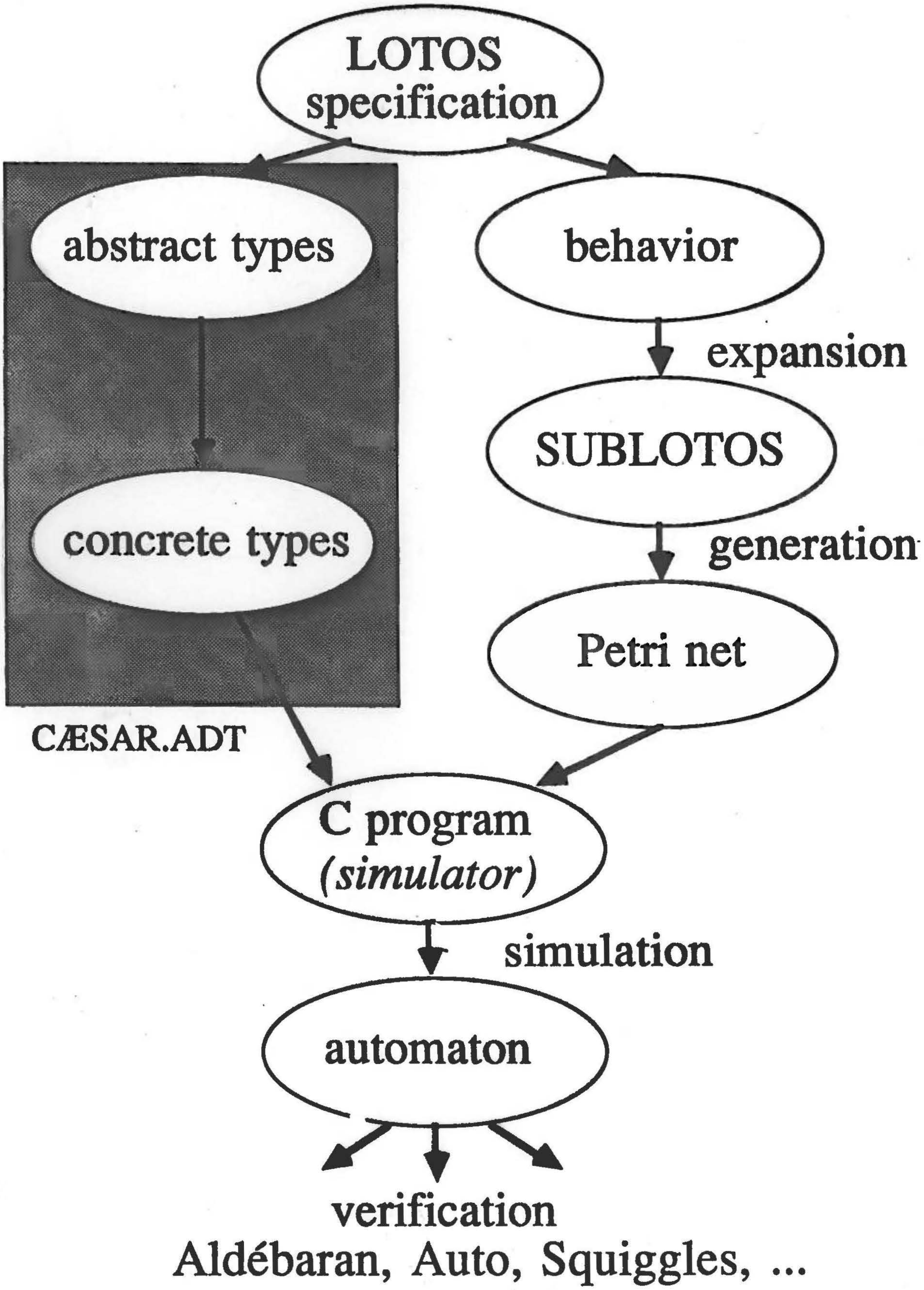
Compilation of LOTOS Abstract Data Types

Hubert Garavel

Laboratoire de Génie Informatique
Institut I.M.A.G
Grenoble, France

(I.N.R.I.A., VERILOG)

CÆSAR and CÆSAR.ADT



Compiling ADTs

Problem

Executing LOTOS abstract data type specifications

Existing solutions

1. dynamic term rewriting
2. code generation for rewriting machines [Wolz-Boehm]

Our approach

1′. static compilation

- performing computations at compile-time
- no pattern-matching, unification, backtracking, ... at run-time

2′. target language: C

Issues

- **data representation**
  LOTOS sorts → C týpes
- **translation of equations into deterministic code**
  LOTOS operations → C functions

[Schnoebelen, "Refined Compilation of Pattern-Matching for Functional Languages", SCP, 1988]

Example

- taken from the transport service [ISO-8072]
- history of requests
- transformations:
  - some operations removed: **NonEmpty**, eq, ne
  - one operation introduced: App

```
type TransportServiceBasicTSPRequestHistory is ...
sorts
  History
opns
  NoTReqs :   -> History
  App     :   TSP, History -> History
  Append  :   TSP, History -> History
  Empty   :   History -> Bool
eqns
  forall t, t1, t2 :  TSP,
        h, h1, h2 :  History
  ofsort History
    not (IsTReq (t)) => Append (t, h)  = h;
    IsTReq (t)       => Append (t, h)  = App (t, h);
  ofsort Bool
    Empty (NoTReqs)    = true;
    Empty (App (t, h)) = Empty (h) and not (IsTReq (t));
endtype
```

Implementing data

1. **apply flattening to the specification**

2. **treat each sort S in turn**

Here: $S =$ `History`

3. **consider the set of operations with result of sort S**

Here:

```
NoTReqs  :  -> History
App      :  TSP, History -> History
Append   :  TSP, History -> History
```

4. **divide this set in two parts**

- **constructors**: not completely defined by the equations

- **non-constructors** completely defined by the equations
  non-constructor operations can always be rewritten

Here:
⋆ constructors: `NoTReqs` and `App`
⋆ non-constructors. `Append`

```
not (IsTReq (t)) => Append (t, h) = h;
IsTReq(t)        => Append (t, h) = App (t, h);
```

Constructor identification can be done:

- by hand (as in CÆSAR.ADT)
- automatically [Comon]

5. choose an implementation for values

{ values of sort S } $\subseteq$ { terms made only of constructors }

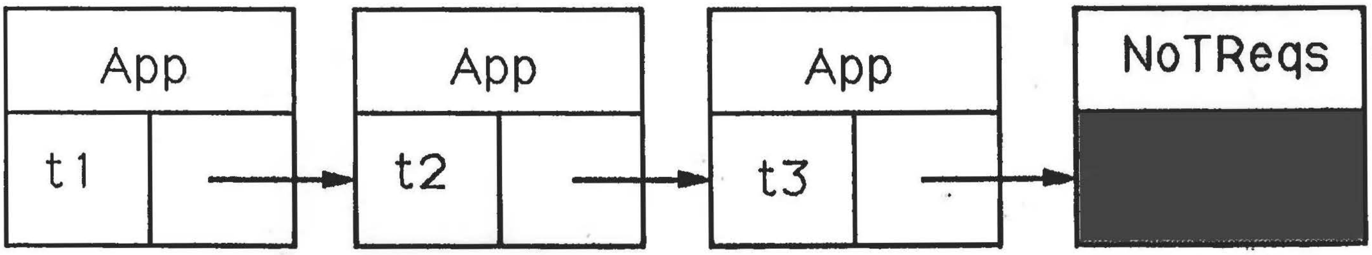
Here:

```
NoTReqs  :  -> History
App      :  TSP, History -> History
```

Syntactical definition:

```
<History> ::= NoTReqs | App (<TSP>, <History>)
```

Example: `App (t1, App (t2, App (t3, NoTReqs)))`



Representation with C data structures:

general: pointers and discriminated unions:

- `<App, t, h>`
- `<NoTReqs>`

optimized: no discriminant

- `<t, h>`
- `NULL`

Compiling operations

Implementation of constructors

- allocation and initialization of a memory cell

```
                ⎧ create a cell <App, t, h>
App (t, h) =    ⎨
                ⎩ return a pointer to it
```

Implementation of non-constructors

- pattern-matching algorithm
- generation by induction on the set of rules

```
Empty (NoTReqs)      = true;
Empty (App (t, h))   = Empty (h) and not (IsTReq (t));
```

⇓

```
                 ⎧ if h0 has the form <NoTReqs> then
                 ⎪   true
Empty (h0)   =   ⎨ else if h0 has the form <App, t, h> then
                 ⎩   Empty (h) and not (IsTReq (t))
```

```
not (IsTReq (t)) => Append (t, h) = h;
IsTReq (t)       => Append (t, h) = App (t, h);
```

⇓

```
                     ⎧ if not (IsTReq (t)) then
                     ⎪   h
Append (t, h)   =    ⎨ else if IsTReq (t) then
                     ⎩   App (t, h)
```

Theoretical issues

Restrictions

- equations are **oriented**
- equations must be **left-linear**

```
              f (t, h, h) = Append (t, h)
                            ⇓
        h = h' => f (t, h, h') = Append (t, h)
```

- **equations between constructors** must be removed

Termination

- What happens if the rewriting system does not terminate?
- The generated code loops (unfinite recursive calls).

```
          f (t, h) = Append (t, f (t, h))
                         ⇓

     f (t, h)  = { Append (t, f (t, h))
```

Confluence

- What happens if the rewriting system is not confluent?
- Call-by-value + decreasing priority is assumed.

```
              g (t, NoTReqs) = false;
              g (t, h) = IsTReq (t);
                        ⇓

                 ⎧ if h has the form <NoTReqs> then
                 ⎪   false
   g (t, h)  =   ⎨ else
                 ⎩   IsTReq (t)
```

Conclusion

- LOTOS ADTs can be translated into C libraries
- a prototype tool exists: CÆSAR.ADT
- translation is general
- translation is fast
- generated code is efficient, even optimal for:
  - integer numbers
  - enumerated types
  - tuples (records)
- other applications:
  - LOTOS → ASN.1
    SDL → C