Integrating formal methods within a process calculi framework

Hubert Garavel

joint work with the VASY team

INRIA Grenoble Rhône-Alpes

http://www.inrialpes.fr/vasy

Outline

- Motivations
- A word about CADP
- Integration at a low level: semantic models
- Integration at a high level: user interfaces
- Integration at a high level: languages
- Concluding remarks

Motivations

Proliferation of formal methods

- There are so many formal methods!
  - see the Formal Methods Web page of J. Bowen
  - see Wikipedia
- Why?
  there are (at least) 4 possible causes
- Can we integrate them?
  (this is the theme of this IPA school)
- Warning! The talk might be biaised towards:
  - process calculi, especially LOTOS
  - verification, especially explicit-state verification, especially CADP

Cause #1: different concepts

- Complex systems exhibit different aspects
  - data: types, functions, equations…
  - concurrency: behavior, processes, communication, synchronization…
  - real-time: delays, deadline (urgency)
  - performance and probabilities

- Multiplicity of concepts in real systems is a philosophical problem

- Two schools:
  - The *rigoric* one
  - The *flexible* one

The "rigoric" school

- Scientists like to keep things simple (Occam's razor principle)
- They like if the world can be seen and described using one single formalism
- Wonderful result: any formalism with the expressiveness of a Turing machine can do the job — Yet, this is not always adequate

Counter-examples

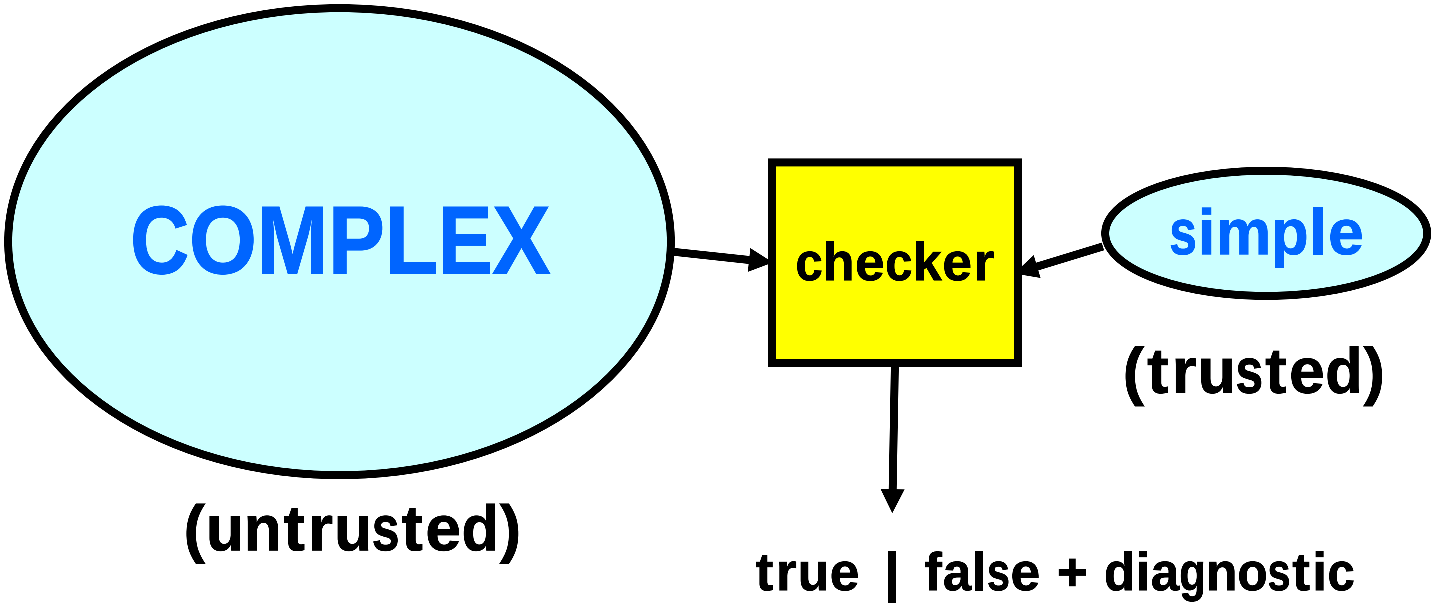
- Example 1: algebraic data types
  - no support to model concurrency
  - SOS semantics ends up being coded in the program!
- Example 2: "pure" process calculi
  - "pure CSP", "basic LOTOS", pi-calculus, etc.
  - FIFOs modeled by dynamic creation of processes!
- Example 3: real time
  - Continous delays modeled by discrete ticks!
  - Urgency (“must”) modeled by choice (“may”)

The "flexible" school

- Convenience first: do not hesitate to combine concepts if needed
- Issue #1: coherence
  - how to ensure a sound semantics?
- Issue #2: redundancy
  - means of expression can be duplicated (e.g. data vs processes)
  - requires guidelines for a preferred style

Cause #2: various verification approaches

Verification is essentially a comparison:



- Two options:
  - one-language
  - two-languages
- *Actually plenty of other options:*
  - *state-based vs action-based, linear-time vs branching-time, etc.*

The “one language” approach

- COMPLEX and simple are described in the same language

- Example 1: theorem proving
  COMPLEX and simple are formulas

- Example 2: equivalence checking
  COMPLEX and simple are automata (LTS, etc.)

The “two languages” approach

- COMPLEX and simple are described using two different languages
  - COMPLEX is often in an imperative language
  - simple is often in a declarative language
- Example 1: Hoare’s logic
  - COMPLEX is a sequential program
  - simple is a pre- and a post-condition
- Example 2: Model checking
  - COMPLEX is a concurrent program (or hardware)
  - simple is a temporal logic formula

Cause #3: different application domains

- Computer science is, in principle, unified
- But it has different applications fields:
  - Telecommunications
  - Avionics
  - Hardware architectures
  - Embedded systems
  - Web services
  - etc.
- Formal methods are often influenced by their potential users
- Tradeoff between a single universal formal method and several specialized ("domain specific") ones

Examples

- Avionics:
  - Many engineers have an electronics or control theory background
  - Graphical languages are appealing to them (LUSTRE/SCADE, …)
- Telecommunications:
  - Engineers are familiar with message queues
  - They like languages with built-in FIFO queues (which queues? bounded or unbounded? reliable or lossy? order-preserving or not…)
  - Estelle, SDL
- Hardware:
  - Designers want to model instantaneous communication (as electricity on a wire)
  - Rendezvous is sometimes too simple for hardware design

Cause #4: human factors

- Scientific creativity naturally leads to different variants

- Formal methods are, to a large extent, a matter of individual (subjective, aesthetic, philosophical) taste:
  - graphical vs textual
  - totally functional, totally algebraical, etc.
  - prohibit or require nondeterminism
  - …

Other personal reasons

- The weight of history: joining forces with a competitor may be perceived as a defeat
- A tactic to survive in the international competition:
  defining a different language is a way to protect oneself against comparisons
- National schools:
  - UK: CCS, CSP…
  - NL: ACP, mCRL…
- Even an international standard (LOTOS) based on CCS + CSP was not sufficient…

Summary

- 4 reasons for proliferation of formal methods
  - cause #1: different concepts
  - cause #2: different verification techniques
  - cause #3: different application domains
  - cause #4: human factors

- Is this proliferation suitable or not?
  - diversity (= positive)?
  - or fragmentation (= negative)?

We could accommodate...

- "Moral" arguments:
  - All formal methods are equal in dignity ☺
  - We should preserve the diversity of formal methods as we should preserve threatened species ☺
- "Economical" arguments:
  - Competition is suitable by essence
  - We already have several operating systems, graphical user interfaces, file systems, object oriented languages
    (but not as many as formal methods)

But…

- The global picture is confused
- Formal methods have a limited industrial acceptance
- Training is expensive, and industry wants to know in which method to invest
- Tool development is expensive and fragmentation prevents reaching a critical mass of investment

What we should do...

- Increase collaboration (rather than competition)
- Integrate/interconnect formal methods and tools from different origins
- Expected benefits:
  - reduce the complexity presented to end-users
  - factorize tool development
  - reuse tools developed for other languages

Several forms of integration

- Low-level integration: semantic models
  - code is shared and reused between tools
  - the user still perceives that it has different tools
  - common semantic models

- High-level integration:
  - more ambitious
  - common user interfaces
  - unified languages

A word about CADP

What is CADP?

A toolbox for verifying asynchronous systems

- At the crossroads between 2 branches of computer science:
  - Concurrency theory
  - Computer-aided verification

- Development started in 1986 …
  - Caesar: LOTOS compiler / state space generator
  - Aldebaran: bisimulation tool

… continuously enhanced for 20 years

CADP wrt other model checkers

- Parallel programs (rather than sequential programs)
- Message passing (rather than shared memory)
- Languages with a formal semantics (process calculi)
- Dynamic data structures (records, lists, trees...)
- Explicit-state (rather than symbolic)
- Action-based (rather than state-based)
- Branching-time logic (rather than linear-time logic)

CADP verification features

- Several paradigms:
  - Model checking (modal μ-calculus)
  - Equivalence checking (bisimulations)
  - Visual checking (graph drawing)
- Several techniques:
  - Reachability analysis
  - On-the-fly verification
  - Compositional verification
  - Distributed verification
  - Static analysis

Other CADP features

- Beyond mere verification:
  - Multiple input languages
  - Step-by-step simulation
  - Rapid prototyping
  - Test generation
  - Performance evaluation

- Generic software components for verification
- Modular, extensible architecture (APIs)

CADP today

- A comprehensive toolbox
  - 42 tools
  - 17 software libraries
- 5 computing platforms supported
  - Sparc/Solaris, Intel/Linux, Intel/Windows, PowerPC/MacOS X, Intel/MacOS X
- International dissemination
  - License agreements signed with 395 organizations
  - Licenses granted for 909 machines (in 2007-2008)
  - 104 case-studies accomplished using CADP
  - 32 research tools connected to CADP
  - 28 university lectures based on CADP (since 2002)

Three main uses of CADP

- Design of critical systems:
  - academic and industrial case-studies
- Teaching concurrency theory:
  - practical feedback of process calculi, LTS, behavioural equivalences, μ-calculus, etc.
  - lab exercises
- Research in verification:
  - new tools developed using CADP libraries
  - new tools interfaced with CADP tools

CADP and integration issues

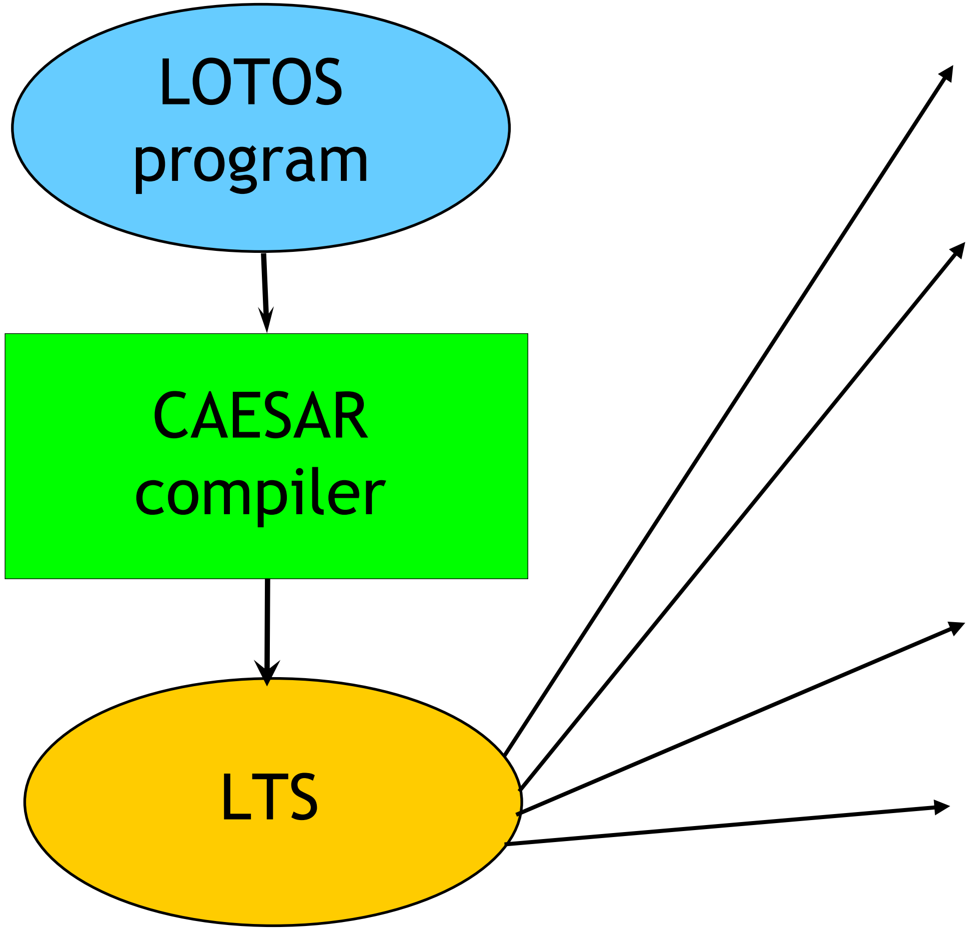
- CADP is the oldest software program implementing concurrency theory results that is still used and enhanced

- From the beginning, the architecture of CADP was designed
  - to be modular
  - to be interfaced with other tools

- In the sequel, we review the CADP approaches to integration

Integration at a low-level: semantic models

Step #1: Interconnection at LTS level



- temporal logics (80's)
  - XESAR
  - MEC
- bisimulation tools (80's)
  - ALDEBARAN
  - AUTO
  - PIPN
  - MEIJE
  - SQUIGGLES
  - SCAN
- bisimulation tools (90's)
  - CWB
  - FC2
- graph drawing tools (90's)
  - GML
  - VCG
  - VISCOPE

The BCG format

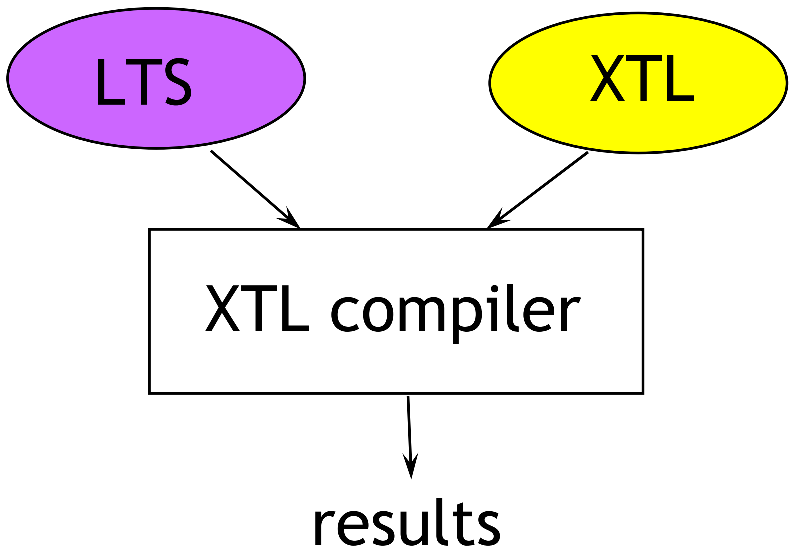
- Various problems:
  - each of these tools had its own LTS format
  - these formats were often poorly defined (ambiguous)
  - these formats were textual (verbose, loss of disk space)

- Idea: define a generic LTS format
  - a binary format with compression techniques
  - typed information attached to states and transitions

- BCG (Binary-Coded Graphs):
  - a compact file format for storing LTSs
  - a set of APIs
  - a set of software libraries (30,000 lines of code)
  - a set of tools (binary programs and scripts)
  - conversions between BCG and other formats

Step #2: XTL

- How to exploit the contents of BCG files?
- XTL is both:
  - a query language for LTSs encoded in BCG
  - a compiler for this language



XTL

- Main features of XTL
  - functional language with model checking features
  - special types: **states, state sets, transitions, transition sets, labels…**
  - access to the typed objects of the BCG file

- Applications of XTL
  - libraries: HML, CTL, ACTL, mu-calculus
  - rapid prototyping of temporal logics
  - temporal logics extended with value passing

XTL: An example

The $\langle A \rangle F$ modality of HML (Hennessy-Milner logic) can be expressed in XTL

$\langle A \rangle F$ denotes the set of states S that

- lead to states satisfying F
- following transitions satisfying A

```
def Diamond (A:labelset, F:stateset):stateset =
    { S:state where
      exists T:edge among out (S) in
        (label (T) among A) and (target (T) among F)
      end_exists }
end_def
```

Step #3: On-the-fly LTS exploration

- Motivations:
  - *Most model checkers are dedicated to one particular input language (Spin, SMV, ...)*
  - *They can't be reused easily for other languages*
  - *How can we "open" model-checkers to get access to their LTS on-the-fly?*
- Idea: introduce modularity by separating
  - **language-dependent aspects:**
    compilers from languages into an LTS model
  - **language-independent algorithms:**
    algorithms for LTS exploration

Implicit LTS: Open/Caesar

Another practical issue arising in the early 90's

How to combine:

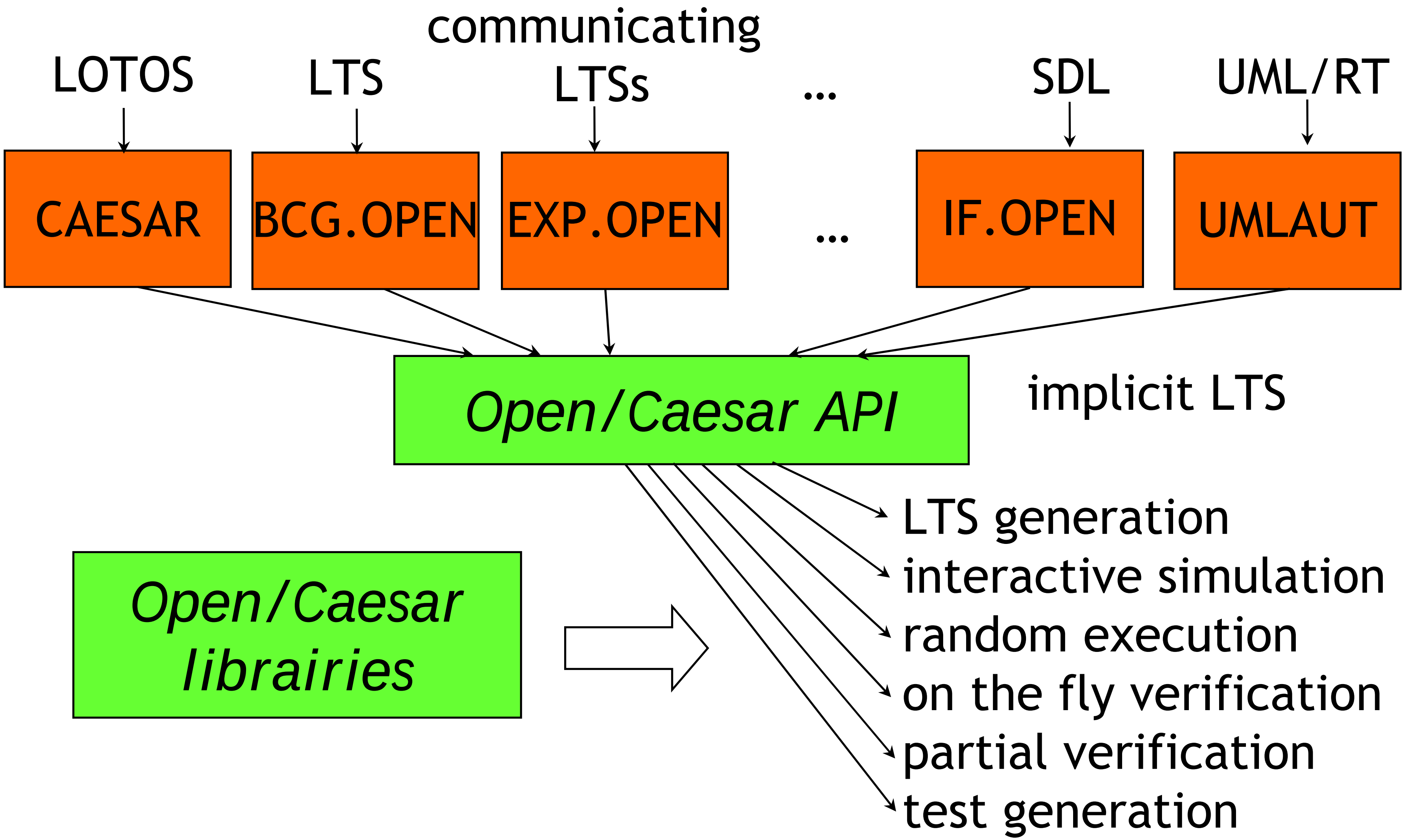
- a separation betwen LTS generation and LTS verification
- and the need for "on-the-fly" verification?

Both were needed, but seemed incompatible at first sight

Solution: the Open/Caesar architecture [Garavel-1998]

- A programming interface to separate language-dependent from language-independent aspects
- Many tools have been written above this interface: simulation, testing, verification, etc.
- Other languages than LOTOS have been connected to this interface
- An essential feature of CADP, often replicated in other papers/tools

OPEN/CAESAR architecture



OPEN/CAESAR libraries

A set of predefined data structures

- EDGE: list of transitions (e.g., successor lists)
- HASH: catalog of hash functions
- STACK_1: stacks of states and/or labels
- DIAGNOSTIC_1: set of execution paths
- TABLE_1: state tables
- BITMAP: Holzmann's "bit state" tables

Specific primitives for on the fly verification

- possibility to attach additional information to states
- stack or table overflow => backtracking
- etc.

An example: GENERATOR

```
#include "caesar_graph.h"
#include "caesar_edge.h"
#include "caesar_table_1.h"

TYPE_TABLE_1 t;     TYPE_STATE s1, s2;                TYPE_EDGE e1_en, e;
TYPE_LABEL l;       TYPE_INDEX_TABLE_1 n1, n2         TYPE_POINTER dummy;

INIT_GRAPH ();
INIT_EDGE (FALSE, TRUE, TRUE, 0, 0);
CREATE_TABLE_1 (&t, 0, 0, 0, 0, TRUE, NULL, NULL, NULL, NULL);
if (t == NULL) ERROR ("not enough memory for table");

START_STATE ((TYPE_STATE) PUT_BASE_TABLE_1 (t));
PUT_TABLE_1 (t);
while (!EXPLORED_TABLE_1 (t)) {
    s1 = (TYPE_STATE) GET_BASE_TABLE_1 (t);
    n1 = GET_INDEX_TABLE_1 (t);
    GET_TABLE_1 (t);

    CREATE_EDGE_LIST (s1, &e1_en, 1);
    if (TRUNCATION_EDGE_LIST () != 0) ERROR ("not enough memory for edge lists");

    ITERATE_LN_EDGE_LIST (e1_en, e, l, s2) {
        COPY_STATE ((TYPE_STATE) PUT_BASE_TABLE_1 (t), s2);
        (void) SEARCH_AND_PUT_TABLE_1 (t, &n2, &dummy);
        print_edge (n1, STRING_LABEL (l), n2);
    }
    DELETE_EDGE_LIST (&e1_en);
}
```

OPEN/CAESAR applications

- EXECUTOR: random walk
- SIMULATOR: interactive simulation (textual)
- XSIMULATOR: interactive simulation (graphical)
- GENERATOR: exhaustive LTS generation
- REDUCTOR: LTS generation with safety reduction
- PROJECTOR: LTS generation with constraints
- TERMINATOR: Holzmann's bit-space algorithm
- EXHIBITOR: search paths defined by reg. expr.
- TGV: test sequence generation

and more…

Step #4: On-the-fly verification

- Motivation:
  - The Open/Caesar architecture allows LTS exploration in a modular, generic way
  - Can we get further, with extra software components especially dedicated to LTS verification?
- Approch followed in CADP:
  - additional software layer on top of OPEN/CAESAR
  - BES (*Boolean Equation Systems*) represented internally as boolean graphs
  - BES: a unified formalism for model checking and equivalence checking

Support for BES in CADP

- CAESAR_SOLVE_1:
  - a library for solving (alternation-free) BES on the fly
  - 7 solving algorithms implemented so far
  - based on top of the OPEN/CAESAR API
- 4 applications of CAESAR_SOLVE_1:
  - BES_SOLVE: solver for an explicit (alternation free) BES contained in a gzipped text file
  - EVALUATOR3: evaluation of mu-calculus formulas (extended with regular expressions)
  - REDUCTOR: on-the-fly minimization of an LTS (several equivalences: strong, branching, weak, etc.)
  - BISIMULATOR: on-the-fly comparison of two LTS (an implicit one in OPEN/CAESAR and an explicit one in BCG)

Step #5: Model checking with data

- Introducing data computation in formulas
- Approach:
  - A richer formula language:

```
[ {RECV ?l:NatList} ]
let n:Nat := sum (l) in
   < {DELIVER !n} > < {ACK !n} > true
end let
```

  - Parameterized Boolean Equation Systems (PBES) [Mateescu's PhD thesis]
  - Evaluator 4 model checker (under testing)
- The concept of PBES is now reused in other tools

Summary

In CADP, integration at the level of semantic models was achieved in 5 successive steps:

- #1: BCG (format for explicit LTS)
- #2: XTL (exploration of explicit LTS)
- #3: Open/Caesar (exploration of implicit LTS)
- #4: BES (model- and equivalence-checking on implicit LTS)
- #5: PBES (BES extended with data computations)

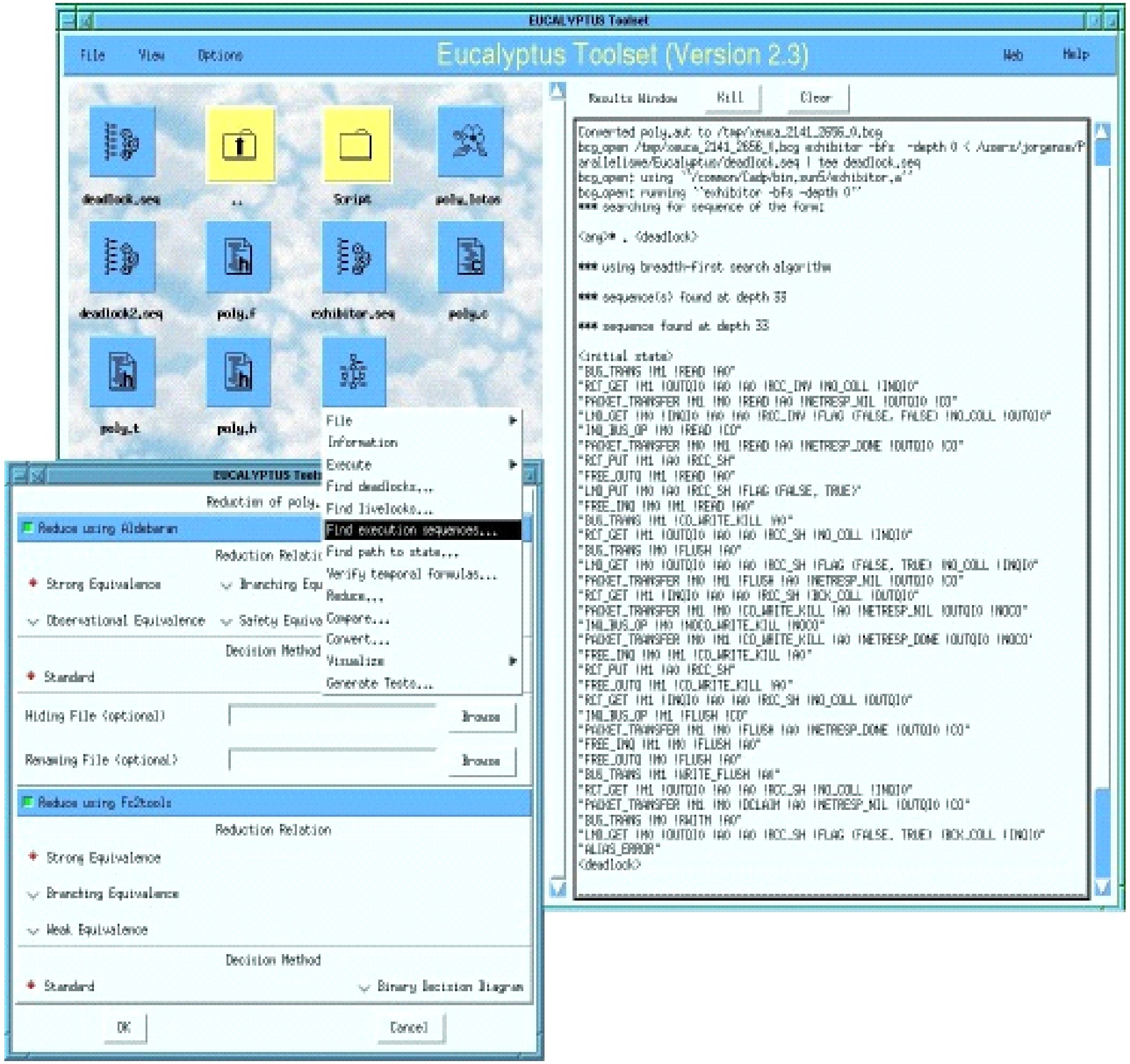
Integration at a high-level: user-interfaces

Interfaces: A key feature for industry

- Early verification tools only had simple command-line interfaces:
  - ad hoc command interpreters (QUASAR, CWB)
  - LISP or Tcl/Tk commands (Meije, FcTools)
- More elaborate interfaces have been developed for CADP
- Two lines of work:
  - a graphical user interface (EUCALYPTUS)
  - a scripting language for verification (SVL)

EUCALYPTUS graphical-user interface

- Version 1 (1994)
- Version 2 (1996-now)
- Main features:
  - file types
  - user-friendly contextual menus
  - support all the CADP tools



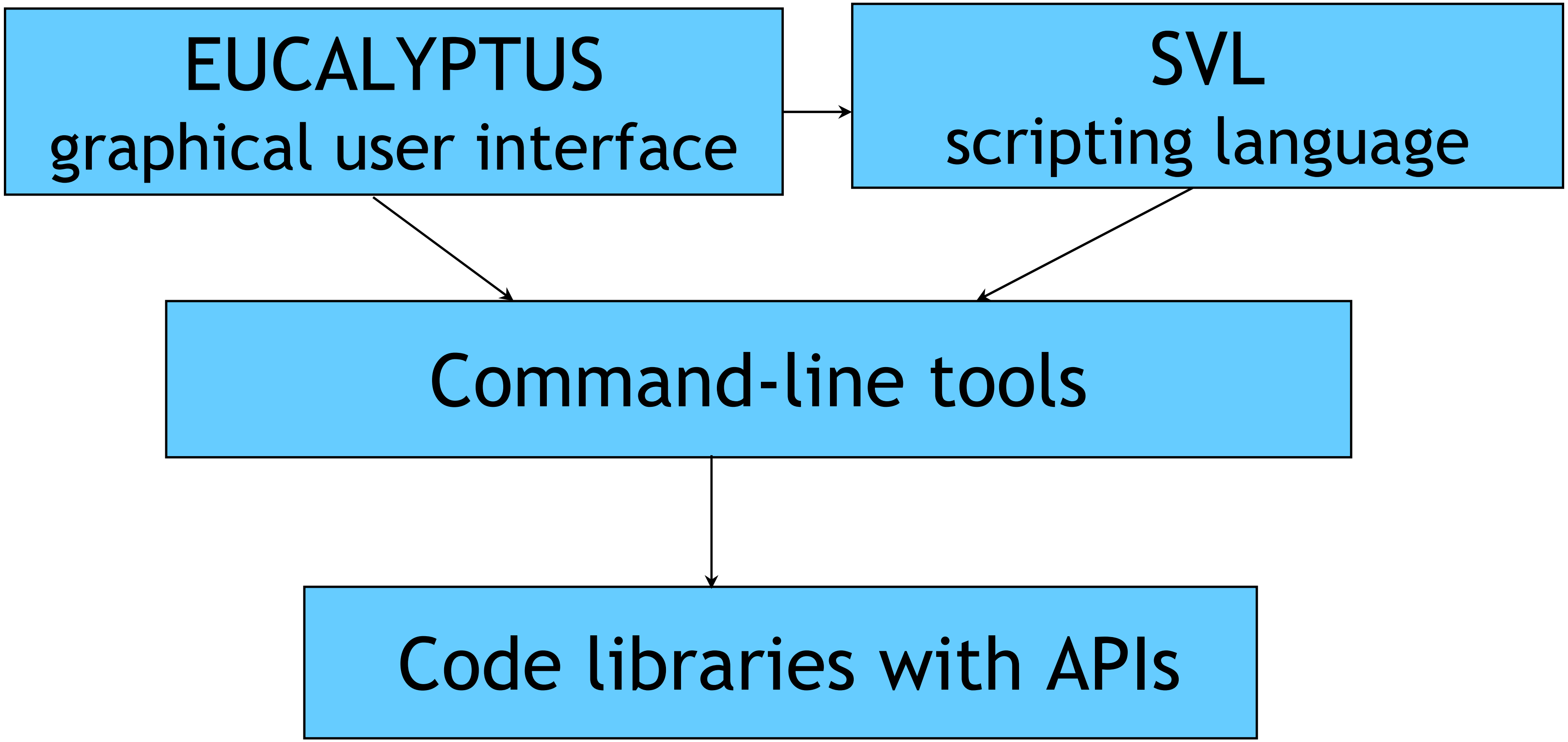
SVL (*Script Verification Language*)

- Scripting language for verification scenarios
- Special constructs for:
  - equivalence checking
  - model checking
  - compositional verification
- "Semantics-aware"

```
"F.exp" = leaf branching reduction of
  hide G in
    (
    "spec.lotos":P1 [A, B, G]
    |[G]|
    "spec.lotos":P2 [C, G]
    ) ;
"D.seq" = deadlock of  "F.exp";
"L.seq" = livelock of "F.exp";
```

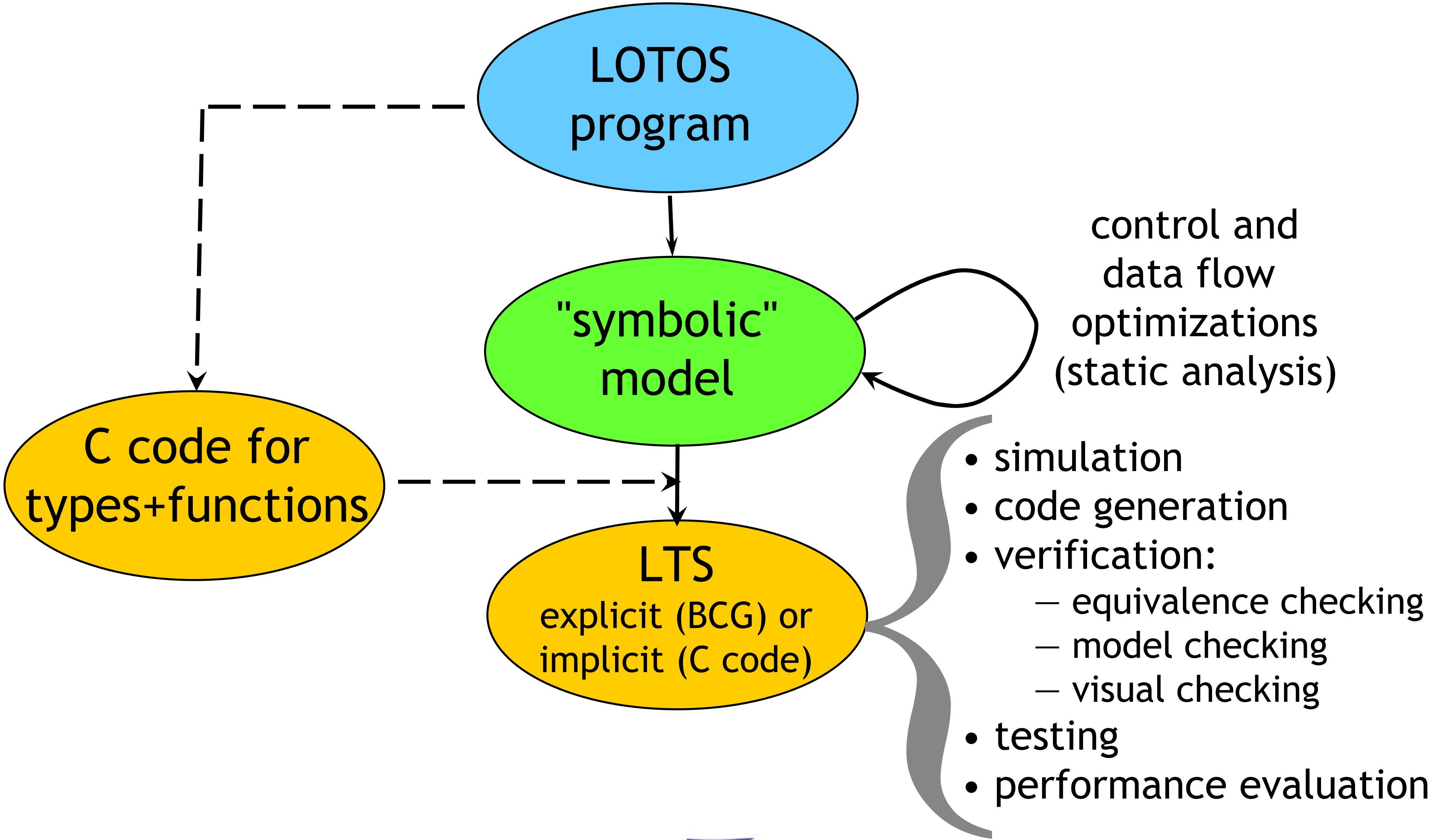
an SVL script

A layered software architecture



3. Integration at a high level: languages

The LOTOS compilers available in CADP



How can we reuse these compilers?

- Academic and industrial users:
  - In general, users dislike learning new languages
  - They want to continue using their favorite languages

- CADP developers:
  - The LOTOS tool chain is a huge work
  - Developing tools for a new language is costly
  - Can we reuse this tool chain for other languages?

- Idea: translate new languages to LOTOS to reuse the LOTOS compilers

Attempt #1: LOTOS vs mCRL

- This was a desirable goal (VASY-CWI collaboration)
- But there are several incompatitilities that make the tranlation cumbersome
- The most annoying one was the order of algebraic equations in data types
  - LOTOS (as handles by CADP) enforces decreasing priority between equations (rewrite system with priorities)

    ```
    forall X, Y: T
    X eq X = true;
    X eq Y = false; (* lower precedence *)
    ```

  - mCRL has no priority at all (a random selection is made)
- We stopped considering this translation

Attempt #2: From CSP to LOTOS

- CSPm (machine-readable CSP): a version of CSP supported by the FDR model checker
- CSPm and LOTOS are close (both derive from CSP)
- But translation from CSPm to LOTOS is difficult:
  - CSPm has higher-order functions (λ-expressions)
  - CSPm allows lazy computations and list comprehensions, whereas CADP relies upon a strict rewrite strategy
  - the choice operator “[]” of CSPm does not translate easily to LOTOS
- We stopped considering this translation

Attempt #3: From CHP to LOTOS

- CHP (Communicating Hardware Processes):
  - a process calculus to describe asynchronous circuits [Martin-86]
  - inspired by guarded commands and CSP
- TAST synthesis tool (TIMA Lab., Grenoble)
  - compiles CHP specifications to VLSI circuits
- But no model checker available for CHP

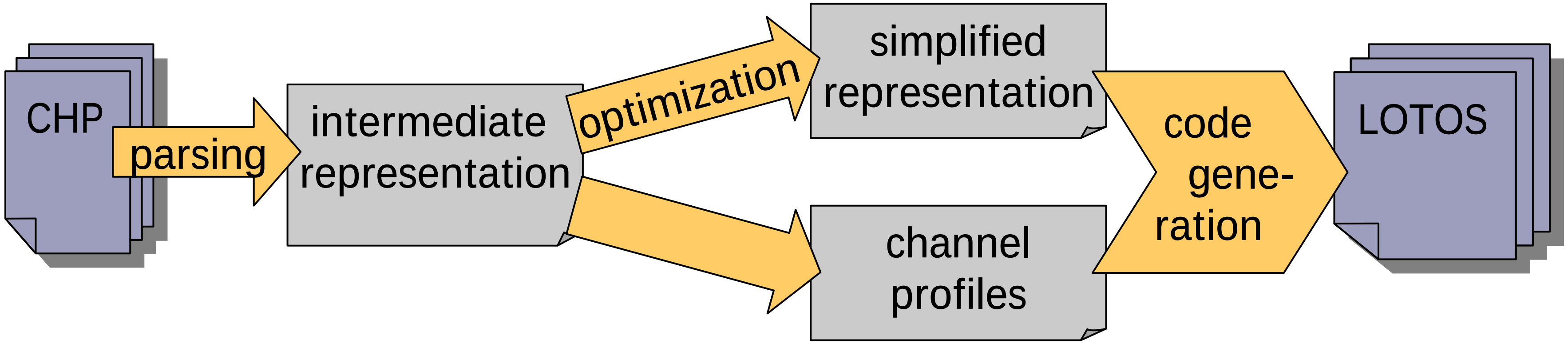
CHP vs LOTOS (1/2)

- CHP has hardware-oriented data types
  - bit arrays
  - machine words, etc.
- CHP has an imperative syntax:
  - variable assignment
  - symmetric sequential composition
  - loop statement
- CHP has two different parallel operators:
  - collateral composition (inside processes)
  - parallel composition to combine processes

CHP vs LOTOS (2/2)

- Main difference: interprocess communications
  - CHP communication reflect the low-level aspects of hardware implementation
  - communication channels are shared variables
  - rendezvous is achieved using special protocols
- In CHP, communication is:
  - oriented (an emitter and a receiver)
  - dissymetric (an active side and a passive side)
  - not atomic (it may takes several steps)
- CHP has a specific "probe" operator:
  - before rendezvous, the receiver can check the value that the emitter is ready to send

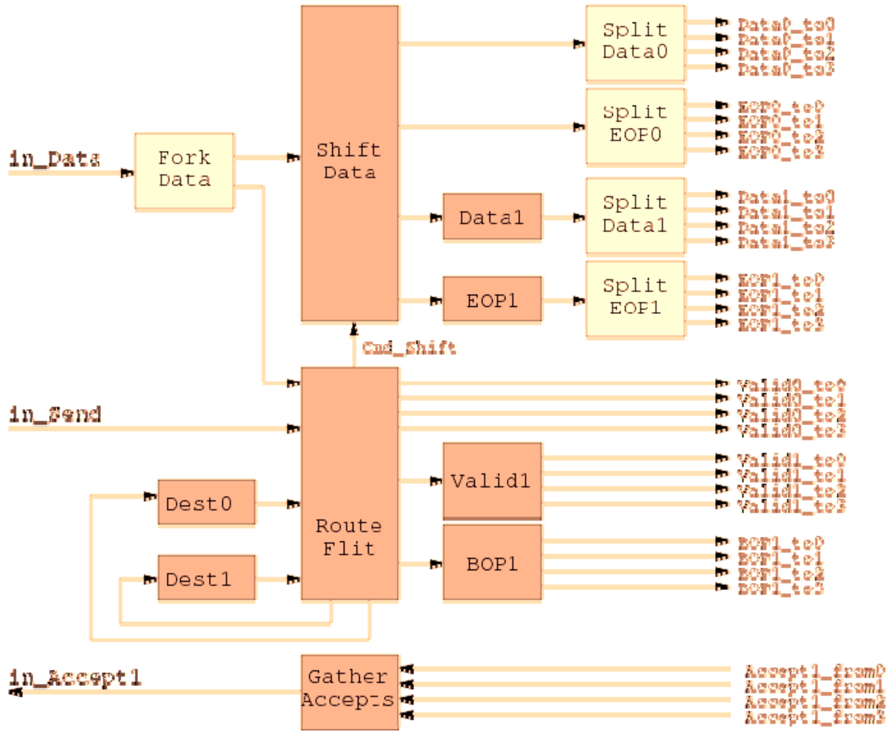
Translator from CHP to LOTOS



- chp2lotos: 19,300 lines of code
- code specialization for different kinds of probes (reduction up to a factor of 156)
- validated on 500 CHP specifications

Application to asychronous circuits

- Three case-studies (joint work between VASY and CEA-LETI)
- DES (*Data Encryption Standard*) chip
- ANOC (*Asynchronous Network on Chip*) communication node
- FAUST network on chip



ANOC node input controller (complex arrangement of 14 asynchronous processes)

Attempt #4: From FSP to LOTOS

Work inspired by this book:

Jeff Magee and Jeff Kramer (Imperial College)
Concurrency: State Models and Java Programs
Wiley, 2006

FSP: a simple, popular process algebra

- concise, expressive, user-friendly
- supported by the LTSA too (animation and LTL property checking)

Joint work undertaken to connect FSP and CADP, so as to verify larger FSP models

Translation from FSP to LOTOS

- Some features of FSP are missing in LOTOS:
  - priority operator
  - label renaming

- Fortunately, these features are handled by the EXP.OPEN and SVL tools of CADP

- So, an FSP specification can be translated into a set of LOTOS, EXP, SVL files

  10,500 lines of FSP produce

  72,000 l. LOTOS, 8,000 l. EXP, 2,000 l. SVL

Translator from FSP to LOTOS

- fsp2lotos: 25,500 lines of code

- Validated on 574 FSP specifications (the LTSs produced by LTSA and CADP are checked to be strongly equivalent)

- fsp2lotos will be shipped with the next version of CADP

Enhancements to LOTOS

- 1988: Ed Brinksma's PhD thesis on Extended LOTOS
- 1993-2001: ISO project to standardize an enhanced version of LOTOS
- Initial goal: a simple revision of LOTOS
- Final result: E-LOTOS
  - complete rewrite of LOTOS
  - abstract data types replaced by functional types
  - process operators replaced by equivalent functional / imperative constructs
  - new features: time, exceptions, modules

E-LOTOS: A mitigated result

- Positive aspects of E-LOTOS:
  - better than LOTOS in most respects
  - simpler syntax (away from the "algebraic" mania)
  - formal semantics (timed LTS, SOS rules)
  - industrial users tend to prefer E-LOTOS to LOTOS
- Negative aspects of E-LOTOS:
  - semantics too complex, irregular at places
  - lack of funding for E-LOTOS
  - never implemented entirely

LOTOS NT

- A "reasonable subset" of E-LOTOS proposed by the VASY team (1995-now)
- Main idea: getting closer to programming languages, still retaining the formal aspects
- Three parts:
  - types
  - functions
  - processes
- Language uniformity: functions are a particular case of processes
- (no support for time at the moment)

LOTOS NT types

- Inductives types:
  - set of constructors with named typed parameters
  - special cases: enumerated types, records, unions, lists, trees, etc.
  - shorthand notations for lists and sets
- Notations for constants:
  - natural numbers: 123, 0xAD, 0o746, 0b1011
  - integer numbers: -421, -0xFD, -0o76, -0b110
  - characters: 'a', '0', '\n' , '\\', '\"'
- Standard functions ("==", "<=", "<", ">=", ">" , field selectors and updaters) are defined automatically

Sample LOTOS NT types

```
type DAY is (* enumerated type *)
        MON, TUE, WED, THU, FRI, SAT, SUN
        with "==", "<=", "<", ">=", ">"
end type
type DATE is (* record type *)
        DATE (D : DAY, N : NAT, M : NAT, Y : NAT)
        with "get", "set"
        (* for selectors X.D, ... and updaters X.{D => E}*)
end type
type NAT_LIST is (* inductive type *)
        NIL,
        CONS (HEAD : NAT, TAIL : NAT_LIST)
end type
```

LOTOS NT functions

- Three kinds of parameters: "in" (call by value), "out" and "inout" (call by reference)
- Function overloading allowed
- Functions defined using standard algorithmic statements:
  - Local variable declarations and assignments
  - Sequential composition
  - Breakable loops
  - If-then-else conditionals
  - Case statements
  - (Uncatchable) exceptions
- Type checking and variable initialization analysis ensure a clean imperative style

Sample LOTOS NT functions (1/2)

```
function GET_HEAD (L : NAT_LIST) : NAT
   raises EMPTY_LIST : NONE is
   case L in
      var HEAD : NAT in
      NIL -> raise EMPTY_LIST
   | CONS (HEAD, any NAT_LIST) -> return HEAD
   end case
end function
```

Sample LOTOS NT functions (2/2)

```
function COUNT (L : NAT_LIST, out EVENS, out ODDS : NAT) : NAT is
   EVENS := 0; ODDS := 0;
   loop SCAN_L in
        case L in
        var HEAD : NAT, TAIL : NAT_LIST in
          NIL -> break SCAN_L
        | CONS (HEAD, TAIL) ->
                if IS_EVEN (HEAD)
                then EVENS := EVENS + 1
                else ODDS := ODDS + 1
                end if;
                L := TAIL
        end case
   end loop;
   return ODDS + EVENS
end function
```

LOTOS NT processes

- Processes are a superset of functions:
  - variable assignment
  - if-then-else, case, loops, etc.
  - symmetric sequential composition (as in ACP)
- Additional operators:
  - action
  - choice
  - parallel composition
  - gate hiding, etc.
- A safer language than LOTOS:
  - bracketed syntax
  - typed channels (overloading allowed)
  - static semantics constraints (variable initialization, etc.)

Sample LOTOS NT process

```
channel C is
  (N : Nat)
end channel
process ELEVATOR [CALL, GO, UP, DOWN: C] (CURRENT, TARGET: FLR) is
  loop
    if TARGET > CURRENT then
      CURRENT := CURRENT + 1; UP (CURRENT)
    elsif TARGET < CURRENT then
      CURRENT := CURRENT - 1; DOWN (CURRENT)
    else (* TARGET == CURRENT *)
      select
        CALL (?TARGET)
        []
        GO (?TARGET)
      end select
    end if
  end loop
end process
```

Attempt #5: TRAIAN and LNT2LOTOS

- TRAIAN (1996-now):
  - a LOTOS NT → C compiler
  - so far, only LOTOS NT data types are compiled
  - intensively used to build VASY compilers
  - http://www.inrialpes.fr/vasy/traian
- LNT2LOTOS (2005-now):
  - a LOTOS NT → LOTOS translator
  - translation for types and functions finished
  - translation for processes being implemented
  - currently 22,300 lines of code
  - already in use by Bull

Summary

- Translations that do not work:
  - mCRL to LOTOS
  - CSPm to LOTOS
- Translations that work:
  - CHP to LOTOS
  - FSP to LOTOS
  - LOTOS NT to LOTOS
- Translations under study:
  - System C/TLM to LOTOS

Concluding remarks

Conclusion

- Diversity of formal methods: a fact
  plenty of reasons for it
- Integration of formal methods:
  - economically suitable
  - scientifically interesting
- 3 different approaches used for CADP:
  - integration at a low-level: semantic models
    - BCG, XTL, Open/Caesar, BES, PBES
  - integration at a high level: user-interfaces
    - graphical user interfaces, script languages
  - integration at a high level: languages
    - translation of CHP, FSP, LOTOS NT to LOTOS

More information…

http://vasy.inrialpes.fr