CLAIRE: A High-Level Programming Language for Complex Algorithms

Yves Caseau & al.
Bouygues - Direction des Technologies Nouvelles
Ecole Normale Supérieure

Outline

- *Motivations*
- *CLAIRE (short overview)*
- *Rule-based Programming*
- *Set-based Programming*
- *Conclusions*

Background

- Combinatorial Optimization and Constraints
  - Competitive advantages and reciprocal benefits of CO and CP techniques
  - How to combine them: Hybrid Algorithms
- Systematic Approach
  - Resource Allocation, Matching
  - Scheduling (job-shop, cumulative)
  - Routing (TSP, VRP, VRPXX)
  - Time-Tabling
- Produce a Library of Hybrid Algorithm
  - Building Blocks for Optimization Applications
  - Research & Teaching

CLAIRE: Specifications

- Simple and Readable
  - executable pseudo-code (teaching CO algorithms)
  - few simple (well-understood) concepts
- Multi-paradigm
  - Objets , Functions
  - Rules
  - Versions (search tree exploration)
- C++ compatibility, Freeware
  - generate C++ objets
  - efficiency similar to hand-written C++
  - sources and binaries available
  - Next release will generate Java

CLAIRE at a glance

- *Object-Oriented functional language with parametric polymorphism*
  - *mixes compiled and interpreted code, static and dynamic typing*
- *Modeling language (sets, relations, ...)*
  - *high level of abstraction, ease of use*
- *Inference engine for object-based rules*
  - *first-order logic (propagation)*
- *Tools for tree search*
  - *choice points and backtracks*

CLAIRE: Industrial Motivations

- *Better Productivity*
  - *C++ code generator, DLL production (Windows NT)*
  - *Reduction factor approximately 5*
- *Simplify Maintenance*
  - *more readable means easier to maintain*
  - *high level of abstraction, "concise and elegant" programming*
- *Support Reuse*
  - *Black-box component approach is poorly suited to combinatorial optimization.*
  - *We build source-code libraries*

Objects

- *Single inheritance class hierarchy*

point <: object(x:integer, y:integer)

- *Parametric Classes*

stack[of] <: thing(of:type, contents:list)

- *Class hierarchy embedded in a multiple inheritance data type lattice*

(1 .. 10) $\subseteq$ (20 .. 30) $\subseteq$ (1 .. 30) $\subseteq$ integer

stack[of = integer] $\subseteq$ stack[of:subtype[integer]] $\subseteq$ stack $\subseteq$ object

tuple(integer,integer) $\subseteq$ list[integer] $\subseteq$ list[integer $\cup$ float] $\subseteq$ list

Polymorphism

- *Free Overloading (attach method to complex types)*

f(x:{0},y:(1 .. 12)) → 1

f(x:(0 .. 10), y:integer) → (x + y)

f(x:integer,y:({1,2} U (7 .. 10)) → (x - y)

- *optimization through code generation*

sum(s:subtype[integer]) : integer

→ let d := 0 in (for x in s d :+ x, d)

sum(1 .. 10) ...

let d := 0, m := 10, x := 1 in

(while (x <= m) (d :+ x, x :+ 1), d)

- *composition polymorphism*
  - *Takes advantage of rules of type f(g(x)) = h(x)*
  - *Example : det(A * B) = det(A) * det(B)*

Sets and Relations

- *Set-based Programming*
  - *data types are sets (e.g., extensions)*
  - *easy syntax for set expressions*

  {x in person | x.age ∈ (0 .. 17)}

  {x.father | x in person}

  list{salary(x) | x in {x in person | x.department = sales}}
- *Efficient Parametrization (operation/representation)*
- *Relations are first-class citizens (inverses…)*

dist[x:(0 .. 100),y:(0 .. 100)] : integer := 0

comment[c:class] : string := ""

- *Modeling Abilities*

meet[s:set[person]] : date := unknown

course[tuple(person,set[person])] : room := unknown

Hypothetical Reasoning

- *Worlds*
  - *world+() creates a choice point*
  - *world-() makes a backtrack*
  - *other derived operations*
- *tree search*
  - *Trailing stack, optimized for depth-first search*

```
solve() : boolean ->
 when q := pick() in
   exists( c in possible(q) |
              branch(  (column[q] := c, solve())))
 else true

branch(e) :: (     world+(),
                         ( (try e catch contradiction
   false) |
                         (world-(), false) ))
```

Logical Assertions

- *CLAIRE uses an "object-oriented" logic which is an extension of binary DATALOG ...*

edge(x,y) | exists(z, edge(x,z) & path(z,y))

⇒ path(x) :add y

- *with methods ...*

exists(z, z = salary(x) & z > 30000 & z < 50000 &

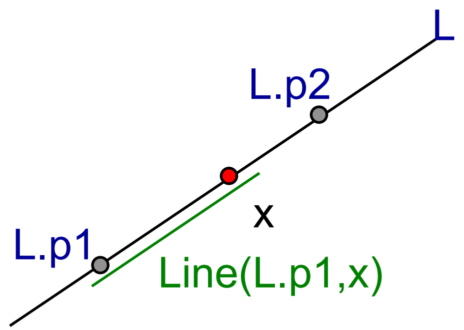
y = 3000 + (z - 30000) * 0.28) ⇒ x.tax := y

- *and the ability to create new objects and sets*

L.p2 ∈ Holds(line(L.p1,x)) ⇒ Holds(L) :add x

z = size({y in person | y ∈ x.children})

⇒ x.family_size := z + 2



Rules

- ## *Production Rules*

```
rule1(x:person[age:(18 .. 40)]) :: rule(
  x.salary < average({y in person | x.department = y.department}
 ⇒ increase_salary(x) )
```

- ## *Control*
  - *Rules are triggered by events (updates on slots & arrays)*
  - *priorities, inheritance*
  - *trigger: once/ exact / many*
  - *rule sets and finer control are easy to implement*
- ## *Queries*
  - *A query is a relation defined by a logical assertion*
  - *naive top-down resolution (no recursion)*

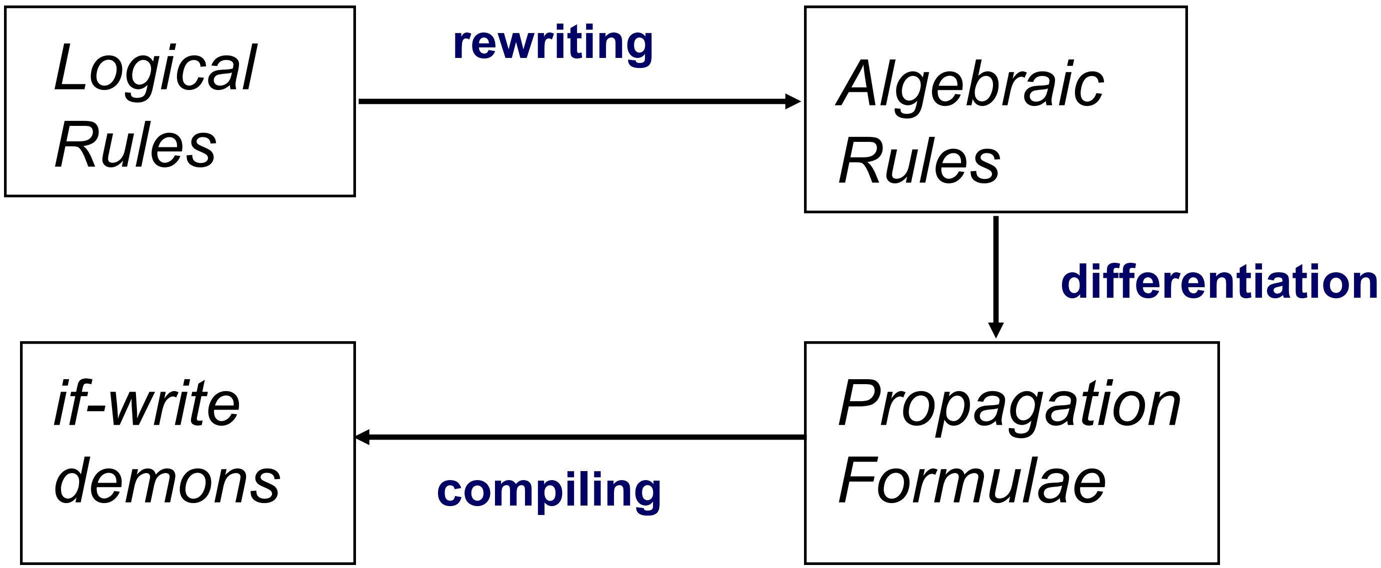
Goodies

- *Modules*
- *(Fast) Exceptions*
- *Memory Management*
- *Second-Order Types*

Why Compile Rules ?

- Performance
  - much faster than best RETE-based compiler
  - no comparison with RETE-base interpreter
- Autonomous objects
  - The generated code is purely procedural
  - no need for additional engine or data structures
  - Ideally suited for Java Beans concept
- Dynamic Rules
  - Dynamic Libraries
  - Consistency-checking at compile-time
  - through interpreter (50 times slower)

Principles for Compiling Rules



CLAIRE compiles a set of rules into a set of demons (procedural attachment). Java observable mechanism could be used.

Rules Compilation

- *Complete compilation*
  - *totally procedural C++ code (no auxiliary data structures)*
  - *Sets/Lists/Vectors handling is thoroughly optimized*
- *Benchmarks*
  - *Standard benchmarks are easy to translate (inverse is not true)*
  - *State-of-the-art results (600K to 3Mips on a P166)*
  - *but no inter-rule optimization is performed*
- *Empirical Results*
  - *10 to 50% penalty compared with hand-optimized procedural code*
  - *source code reduction factor from 4 to 30.*

Code Example

- *The Airline scheduler:*

```
city <: thing(
  time_zone:integer = 0,
  previous_request:set[request],
  starting:set[flight],
  arriving:set[flight],
  starting_plan:set[travel_plan],
  arriving_plan:set[travel_plan] )
```

7 == (9:(AF401):13, 8:(AF301, LU401):12)
8 == (9:(AF401):13, 8:(AF301, LU401):12)
9 == (9:(AF401):13)
10 == (12:(AF011, BA401):19,
10:(AF311, LU511, LU411):16)
11 == (12:(AF011, BA401):19)
12 == (12:(AF011, BA401):19)

```
complex_plan(r:request, f:flight) :: rule(
 (f.fromc = r.start & f.to != r.arrive &
  f.depart  >= r.depart & (f.depart + f.time) <= r.end_at )
 ⇒ let r2 := request!(f.to, r.arrive ,(f.depart + f.time), r.end_at) in
     (if not(r2 ∈ f.to.previous_request)
        (f.to.previous_request :add r2,
         r2.start := f.to,
         r2.arrive := r.arrive)) )
```

Rules Applications

- *Crane scheduling application*
  - *Propagation of resource constraint*
  - *Complex propagation patterns*
    - 1 week to implement 3 rules with methods
  - *minimal performance degradation*
- *Television Advertising Optimization*
  - *very large bin packing with constraints*
  - *Ad-hoc limited tree search with heuristics*
  - *Rules are used for defining heuristic control parameters*

(x.satisfaction < minsat & x.productline = … & y ∈ x.products
& y.satisfaction < (minsat / 2) …) => priority(y) :* 2

Set-Based Programming

Set-based Programming is the combination of:

- *sets are "first-class citizens" with a choice of multiple representations (abstraction)*
- *abstract sets may be used to capture "design patterns"*

```
for x in {p.father | p in person} ...

for y in {p in person | p.father.retired? = true} ...

sum({f(x) | x in {y in (1 .. 10) but 5 |  p(y) >0}})

min( i but t, <Atleast, TE)
```

Concrete and Abstract Sets

- *Concrete Sets*
  - *classes* `for x in person print(x), size(person)`
  - *sets, lists* `set(1,2,3,4), list(Peter, Paul, Mary)`
  - *data structure library (Bitvectors, Hset, ...)*
- *Data Types*
  - *Intervals* `1 .. n, "abc" .. "abd"`
  - *Union, Intersection* `array U property, list ^ subytpe[char]`
  - *Parameterized Types*

    `for x in person[age:(15 .. 18)] print(x)`
- *Set Expressions*
  - *image*

    `{age(x) | x in person}, list{i + 1 | i in (1 .. 10)}`
  - *selection*

    `{x in person | x.age > 0}, list{i in (1 .. n) | f(i) > 0}`

Extensibility

- *Classes*
  - *extending the data structure library*

```
Hset[of] <: set_class(of:type,content:list,index:integer)

add(s:Hset[X], y:X) : void
 -> let i := hash(s.content,y) in ....
set!(s:Hset) -> {x in s.content | known?(x)}
```

- *Patterns*
  - *a pattern is a function call that is dealt with lazily*

```
but(x:abstract_set,y:any) : set -> {z in x | z != y}

%(x:any, y:but[tuple(abstract_set,any)])
    => (x % y.args[1] & x != y.args(2))
```

Explicit Iteration

- *Lazy evaluation directed by set type*

```
for x in person print(x)
for y in (1 .. n) f(y)
```

- *Optimization through source code generation*

```
for c in person.descendent
   for x in c.instances print(x)
let y := 1 in
   (while (y <= n) (f(y), y :+ 1))
```

- *Iteration mechanism is extensible*

```
iterate(x:Hset,v:Variable,e:any)
   => for v in x.content (if known?(v) e)

iterate(x:but[tuple(abstract_set,any)],v:Variable,e:any)
   => for v in x.args[1] (if (v != x.args[2]) e)
```

Implicit Iteration

- *Image/selection expressions imply iterations*

```
{x in (1 .. 10) | f(x) > 0}
{length(c.subclass) | c in (class but class)}

let s := {}, x := 1 in
   (while (x <= 10) (if (f(x) > 0) s :add x, x :+ 1),
    s)
```

- *The iteration of such an expression is also lazy :*

```
for x in {x in (1 .. 10) | f(x) > 0} print(x)
for y in {c.slots | c in (class \ relation.ancestors)} print(y)

for c in class.instances
   if not(c % relation.ancestors) print(c.slots)
```

Abstraction: A Tradeoff

- *CLAIRE supports true abstraction:*
  - *One may substitute a representation by another and not have to change a line of code*
  - *On the other hand, re-compiling is necessary*
- *CLAIRE supports efficient generic methods*

```
sum(s:subtype[integer]) : integer
 => let d := 0 in (for x in s d :+ x, d)

count(s:subtype[integer]) : integer
 => let d := 0 in (for x in s d :+ 1, d)

min(s:abstract_set,p:property,default:any) ....
```

Application: Embedded Linked List

- *Embedding (using slots) is more efficient*

```
Task <: object( ....
               next:Task, prev:Task, ....)
```

- *Patterns may represent « virtual linked lists »*

```
chain(x:Task) : list[Task]
    -> let l := list(x), in
         (while known?(next,x) (x := x.next, l :add x),
          l)
insert(x:Task,y:list[Task]) => ...
iterate(x:chain[tuple(Task)],v:Variable,e:any) => …
```

- *These lists may be used as usual :*

```
count(chain(t1)), sum({x.weight | x in chain(t0)}), ...
```

Feature Combination

```
iterate(s:Interval, v:Variable, e:any)
  => for v in s.use.users (if SET(v)[v.index] e)

<atleast(x:Task, y:Task) => (x.atleast <= y.atleast)

min(s:any, f:property, default:any) : any
   => let x := default in
           (for y in s (if f(x,y) x := y),  x)
```

```
//  the task with the smallest earliest start date in i, different from t
min(i but t, <atleast, TEnd)

let x := TEnd in
    (for y in i.use.users
        (if SET(i)[y.index]
           (if (y != t)  (if (y.atleast <= x.atleast) x := y))),
            x)
```

Structure Iterators

- *Combining iterators and patterns is useful to iterate more complex data structures such as trees :*

```
Tree <: object(value:any,right:Tree,left:Tree)
TreeIterator <: object(tosee:list, status:boolean)
iterate(x:by[tuple(Tree,TreeIterator)], v:Variable, e:any)
  =>  let v := start(x.args[2], x.args[1]) in
           while (v != unknown)
                (e, v := next(x.args[2], x.args[1])

TreeIteratorDFS <: TreeIterator()
start(x:TreeIteratorDFS, y:Tree) -> ...
next(x:TreeIteratorDFS, y:Tree) -> ...
DFS :: TreeIteratorDFS()

TreeIteratorBFS <: TreeIterator() ...

for x in (myTree by DFS) print(x)
{y.weight | y in (myTree by BFS)}
```

A Real-Life Example (I)

```
// builds a maximum weight complete matching
match()
  ->  (...,        // initialization
       while (HN != N)   (if not(grow())  dual_change()))

// a step repeats until the forest is hungarian (return value is false)
// or the matching is improved (return value is true)
// explore is the stack of even nodes that have net been explored yet
grow() : boolean
  -> let i := pop(explore) in
      ( exists( j in {j in GpiSet(i,LastExplored[i] + 1,LastValid[i]) | not(odd?[j])} |
                (if  (sol-[j] != 0)
                     (//[SPEAK] grow: add (~S,~S) to forest// i,j,
                      odd?[j] := true, pushEven+(sol-[j]), tree[j] := i, false)
                 else (augment(i,j), true)))  |
        (if (explore[0] != 0) grow())     )
```

Hungarian Algorithm (II)

```
// change the dual feasible solution, throw a contradiction if there are no perfect matching
dual_change() : integer
  -> let e := Min( list{vclose[i] | i in {j in Dom | not(odd?[j])}}) in
       (//[SPEAK] DUAL CHANGE: we pick epsilon = ~S // e,
        if (e = NMAX) contradiction!(),
        for k in stack(even)     (pi+[k] :+ e, LastExplored[k] := LastValid[k]),
        for j in {j in Dom | odd?[j]}   (pi-[j] :- e, vclose[j] := NMAX)),
        clear(explore),
        for i in stack(even)
           let l := Gpi[i], k := size(l), toExplore := false in
             (while (LastValid[i] < k) (k, toExplore) := reduceStep(i,j,l,k,toexplore),
              if toExplore push(explore,i)))

// look at edges outside the valid set one at a time
reduceStep(i:Dom,j:Dom,l:list,k:integer,toExplore:boolean) : tuple(integer,boolean)
 -> let j := l[k], c := Cpi(i,j) in
          (if (c = 0)  (//[SPEAK] dual_change: Add edge ~S,~S // i,j,
                           Gpiadd(l,i,j,k), toexplore := true)
           else (vclose[j] :min c, k :- 1)),
           list(k,toexplore))
```

CLAIRE Strategy

- Phase I: 95 - 97
  - Free Academic Software
  - Support-as-you-use approach
  - Productivity gains for R&D projects
  - Algorithm libraries
- Phase II: 98 - 99
  - Deployed Applications
  - Industrial Users Club
  - Industrial Support for CLAIRE run-time library
  - libraries: ECLAIR, Schedule, LP, ...
- Phase III: 99 onwards
  - part of development tools suite
  - looking for collaborations

Conclusion

- CLAIRE is a powerful tool for processing business rules:
  - *State-of-the-art inference engine*
  - *Expressive Power*
- CLAIRE is a good tool to implement complex algorithms for decision aid
  - *multi-paradigm*
  - *high level of abstraction*
  - *readability*
- CLAIRE is available
  - *compiler, interpreter, tools for tracing and debugging*
  - *UNIX and Windows NT:* http://www.ens.fr/~laburthe/claire.html
  - *complete system: (20000 lines), compiler (executable) 1.5 Mb*
    - run-time: 5000 lines of code (100K) [Java: 1000 lines]

Industrial Applications

- 4 applications written with CLAIRE have been deployed
  - Construction Equipment Inventory
  - Crane Daily Planning
  - Call Center Scheduling
  - Advertising Resource Optimization
- A BOUYGUES library of reusable components is being developed (business objects)
- A public library (ENS) of classical algorithms is also being developed (forthcoming book)

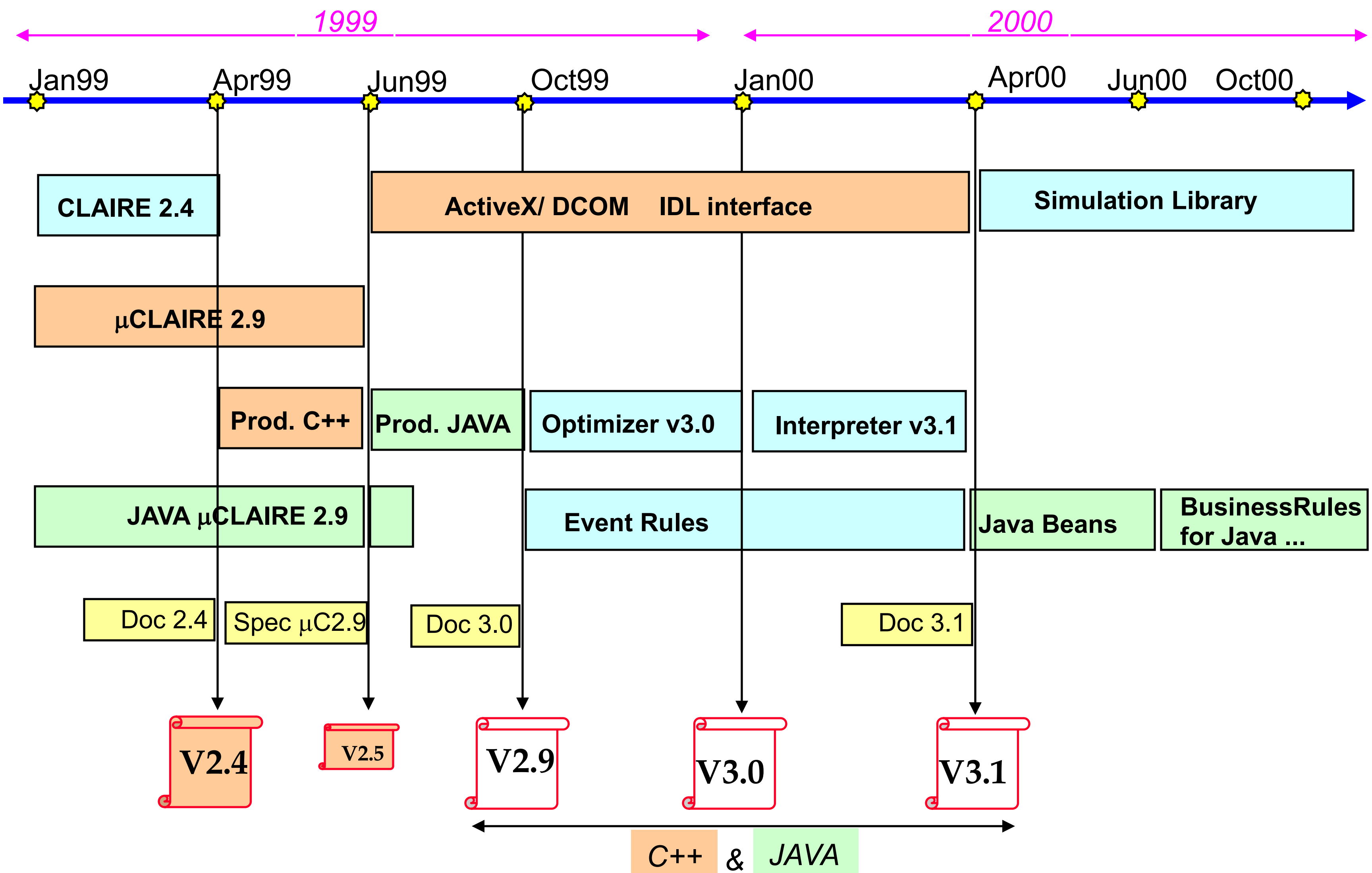
Future Directions

- Component generator
  - Stripped Objects (no reflective descriptions)
  - Message Interface (DCOM/ CORBA/ …)
- Java-based version of CLAIRE
  - Simpler Run-time (almost none)
  - Generate Human-maintenable code
  - Exit Strategy
- Simulation Environment
  - Finite Event simulation for stochastic optimization
  - Agent- based
- Optimization Software Platform
  - Do not extend CLAIRE but build software components
  - Business objects for call center scheduling or routing

Future Releases

- *v2.4*
  - *Arrays (dynamic but constant length)*
  - *Improved compiler for floats*
- *v2.5 (?)*
  - *Re-usable C++ Kernel (shorter -simpler -faster)*
  - *Simplified Type System (safer)*
- *v2.9*
  - *Java Compiler (beta version)*
- *v3.0*
  - *Component Compiler (COM ?)*
  - *Improved Code Optimizer*

RoadMap



1999
2000
Jan99
Apr99
Jun99
Oct99
Jan00
Apr00
Jun00
Oct00
CLAIRE 2.4
ActiveX/ DCOM IDL interface
Simulation Library
μCLAIRE 2.9
Prod. C++
Prod. JAVA
Optimizer v3.0
Interpreter v3.1
JAVA μCLAIRE 2.9
Event Rules
Java Beans
BusinessRules for Java ...
Doc 2.4
Spec μC2.9
Doc 3.0
Doc 3.1
V2.4
V2.5
V2.9
V3.0
V3.1
C++ & JAVA

μCLAIRE: what's up ?

- *C++ Kernel*
  - C++ Namespaces
  - C++ class components
  - Separated Reflective Description
- *Type System*
  - *Integer Intervals Only*
  - *list[t] replaced by list (LISP) and list<t> (C++, C<t> = C[of = t])*
    - lists as objects vs. lists as values
    - safer (uniform for lists, sets and arrays)
- *Optimization*
  - *Native Floats*
  - *Uniform C++ Object implementation (but integers)*
    - simpler debugging
    - faster generic methods
    - easier maintenance (takes more from C++: e.g., streams)