Unified Accelerator Libraries

Make your own way

ROOT

An Object-Oriented
Data Analysis Framework

N.Malitsky and R.Fliller III

Outline

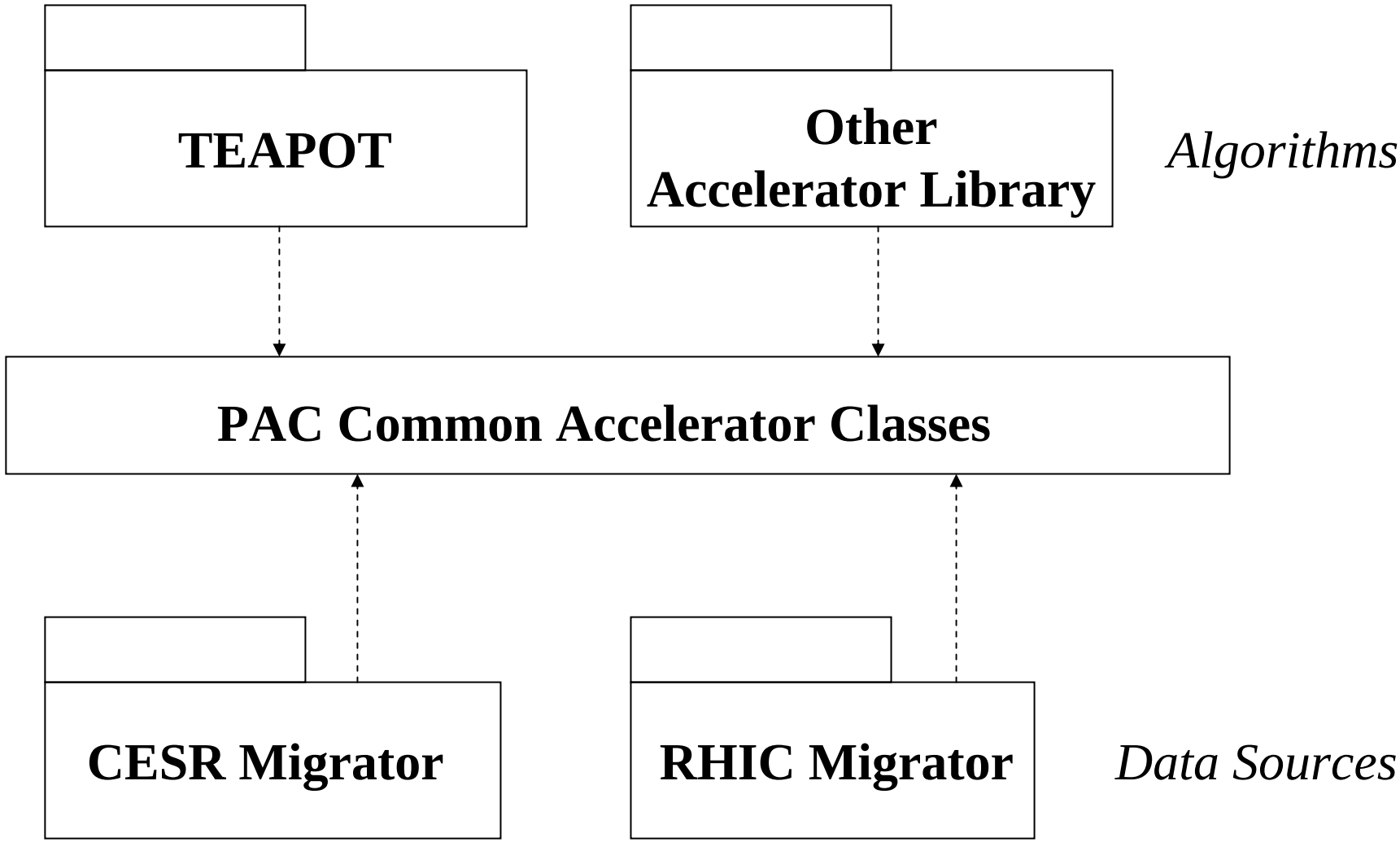
- Unified Accelerator Libraries
- CINT/UAL API interface
- Next Step

UAL Objectives (1995)

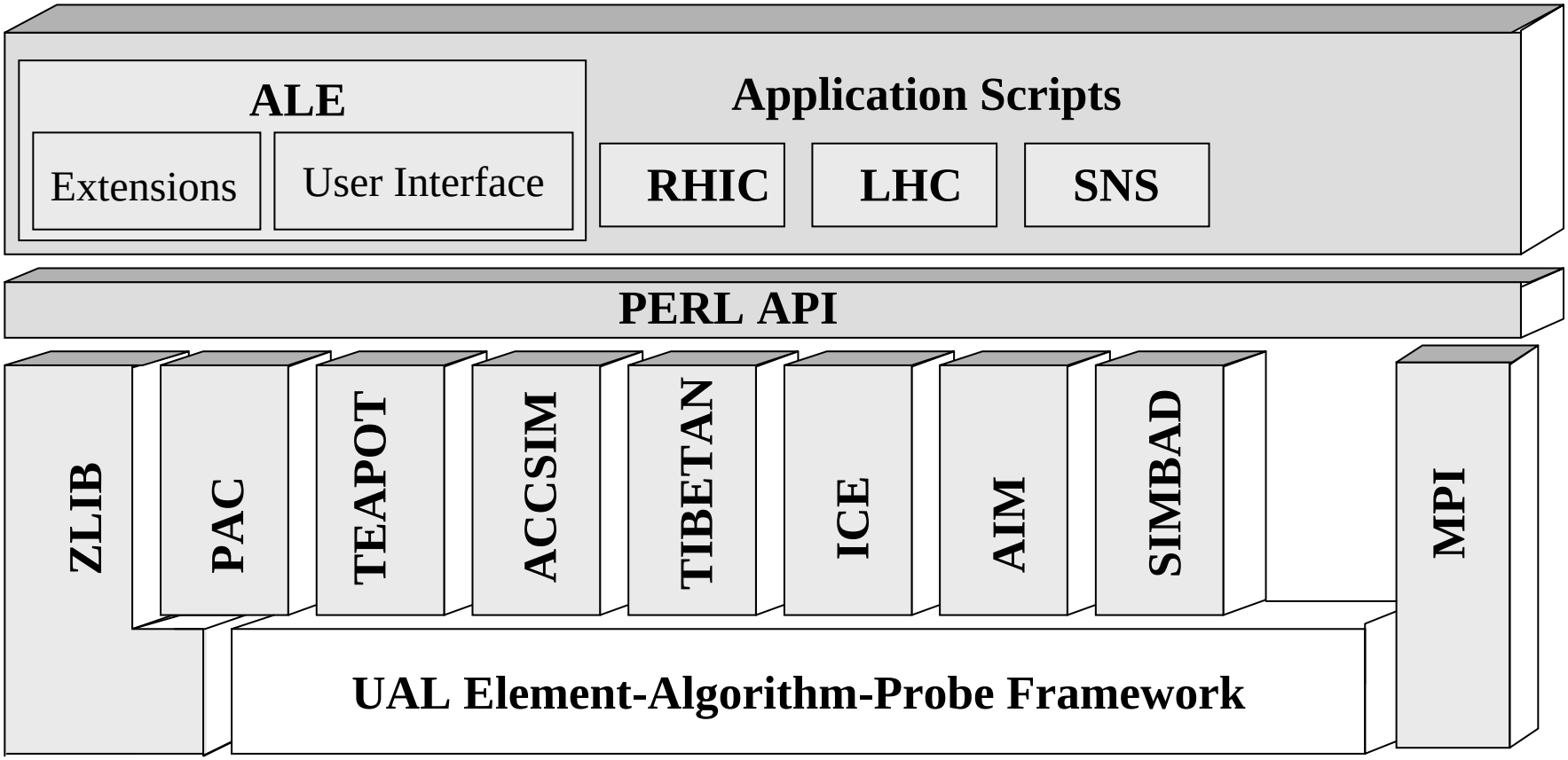
N. Malitsky and R.Talman

- **Form a customizable and extendable environment adaptable to new accelerator applications and conceptual models**
- **Facilitate development, deployment and reuse of diverse independently developed accelerator programs**
- **Integrate accelerator conceptual models and analysis patterns with modern technologies and software**

UAL Conceptual Idea (1995)



UAL Environment

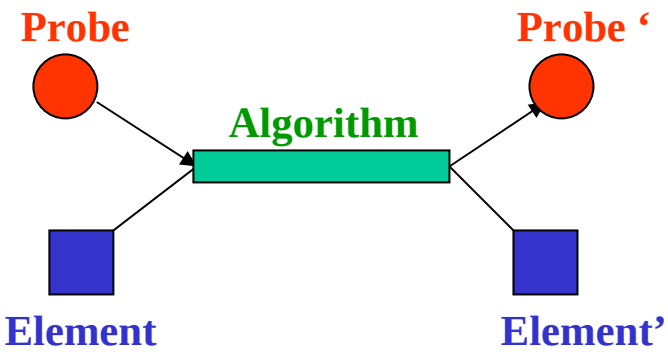


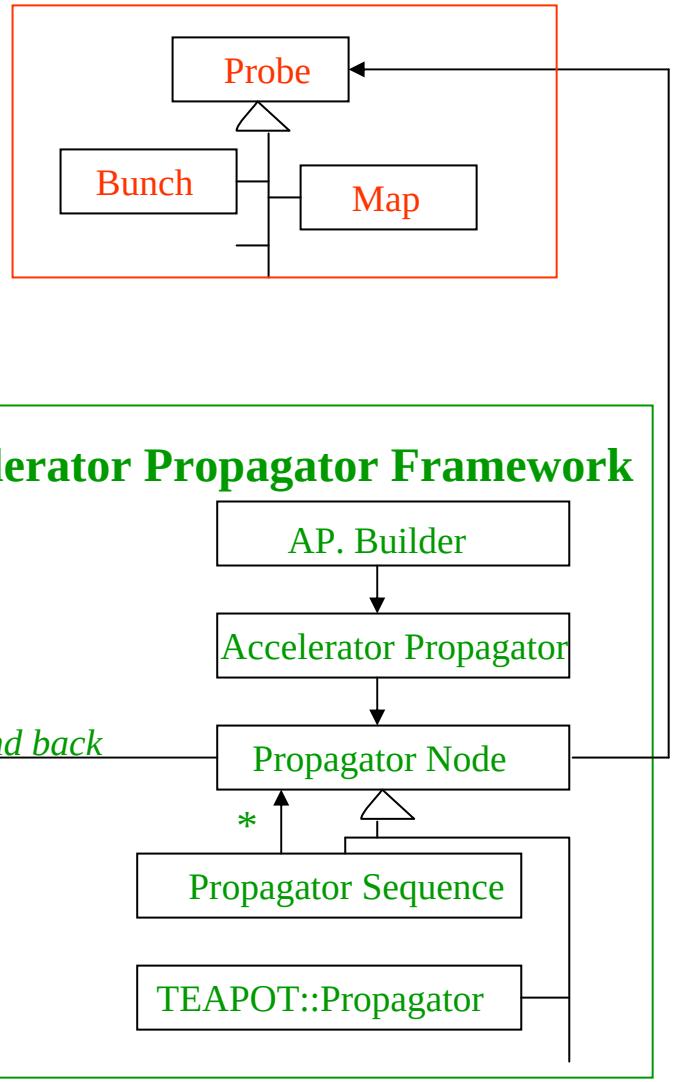
Libraries

http://www.ual.bnl.gov

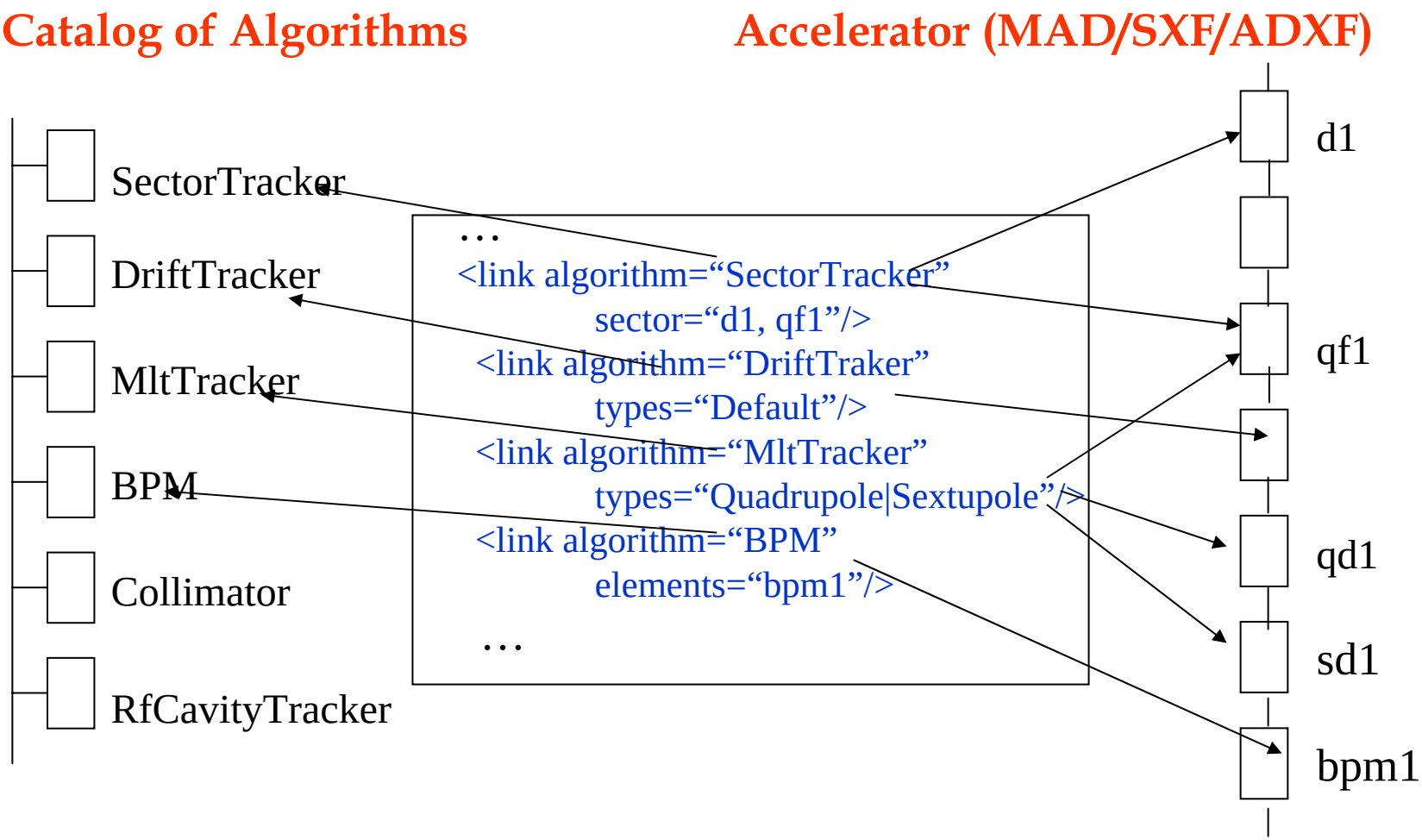
ACCSIM	*Accumulator Simulator (bunch distributions and diagnostics, collimator), author: Fred Jones, home page*
AIM	*Accelerator Instrumentation Models, authors: P.Cameron, M.Blaskiewicz*
ICE	*Incoherent and Coherent Effects, author: M.Blaskiewicz*
PAC	*Platform for Accelerator Codes (collection of Common Accelerator Objects), authors: G.Bourianoff, N.Malitsky, A.Reshetov, R.Talman (SMF)*
SIMBAD	*Simulation of Beam Advanced Dynamics, authors: Alfredo Luccio and Nick D'Imperio,*
SPINK	*Tracking code for polarized particles in a circular accelerator, author: A.Luccio*
SXF	*The SXF represents a flat, complete, and independent description of the current accelerator state, authors: H.Grote, J.Holt, N.Malitsky, F.Pilat, R.Talman, C.G.Trahern, W.Fischer*
TEAPOT	*Thin Element Accelerator Program for Optics and Tracking (collection of symplectic trackers and mappers, collection of correction algorithms), authors: Richard Talman and Lindsay Schachinger*
TIBETAN	*A longitudinal phase space tracking program, author: J.Wei*
UAL	*UAL **Element-Algorithm-Probe** framework, authors: N.Malitsky, R.Talman*
ZLIB	*Numerical library for differential algebra, author: Yiton Yan*

Element-Algorithm-Probe Framework (1998)





Accelerator Propagator Description Format



Evolution of UAL API

1. Accelerator + Propagator files (SXF, XML)
2. C++/CINT Shell with a dynamic collection of arguments: **shell.run(Args << Arg("bunch", bunch) << …**

UI has been divided into **1998**

1. Accelerator file (SXF, XML)
2. Perl User Shell with a dynamic collection of arguments: **$shell->run("bunch" => $bunch, …);**

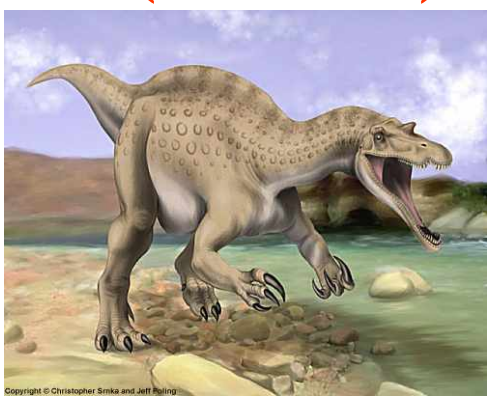
Perl API **1996**

$element->set ("angle" => 0.2, …

C++ API **1994**

element.set(0.2*ANGLE + …

B.C. (Before C++)



Programming Languages (PL)

Feature \ PL	C++	CINT	Perl	Python
Standard PL	★★★★★	★★★★	★★★★	★★★★
Debugging	★★★★★	not tested	★★	★★
C++ from PL	★★★★★	★★★★	★★★	★★★
PL from C++	★★★★★	not tested	not tested	not tested
Maintainable	★★★★★	not tested	★★	★★
Human-friendly (in general)	★★★	★★★	★★★	★★★
Human-friendly containers	★★★	★★★	★★★★★	★★★★★
Light scripting		★★★★★	★★★★★	★★★★★

Human-friendly containers

Perl built-in data types:

- *array*: dynamic collection of Perl variables, e.g.
  @theArray = (2,3,4);
- *hash*: associative array of string and Perl variables, e.g.
  %theHash = (“version” => 5.6, “web site” => “http://www.perl.com”);

C++ STL containers:

- *vector*: fixed-size collection of C++ variable
- *list:* dynamic collection of C++ variables
- *map*: associative array of key/value pairs

Conclusion: C++ STL containers implement the Perl built-in data types

Question: Are they appropriate for the user-oriented API with the variable number of function arguments?

User-Oriented API

Perl style:

```
$shell->run("turns" => 100,
            "bunch" => $bunch,
            "print" => "./results");
```

C++-based proposed variants:

1. overloaded stream operator: Args& operator << (const Arg& arg)

```
shell.run(Args()  << Arg("turns", 100),
                  << Arg("bunch", bunch),
                  << Arg("print", "./results"));
```

2. overloaded comma operator: Args& operator, (Args& args, const Arg& arg)

```
shell.run(( Arg("turns", 100),
            Arg("bunch", bunch),
            Arg("print", "./results") ));
```

Rapid prototyping with light scripting

Rapid prototyping scenario:

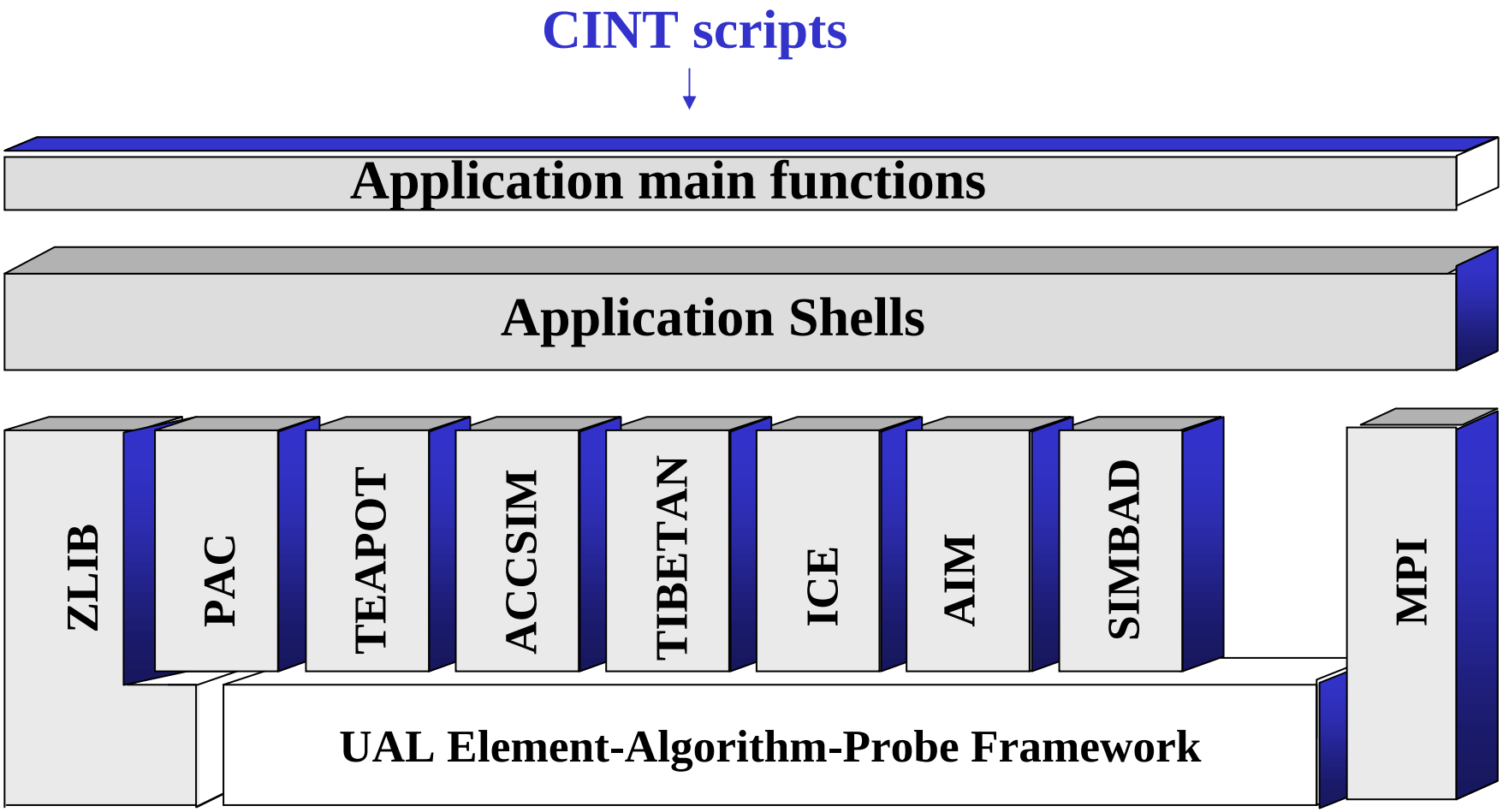
Perl/C++

1. Design of Perl application-specific packages
2. Implementation of these packages
3. Partial debugging
4. Test
5. Use with **light scripting** of the main script
6. Rewriting into C++

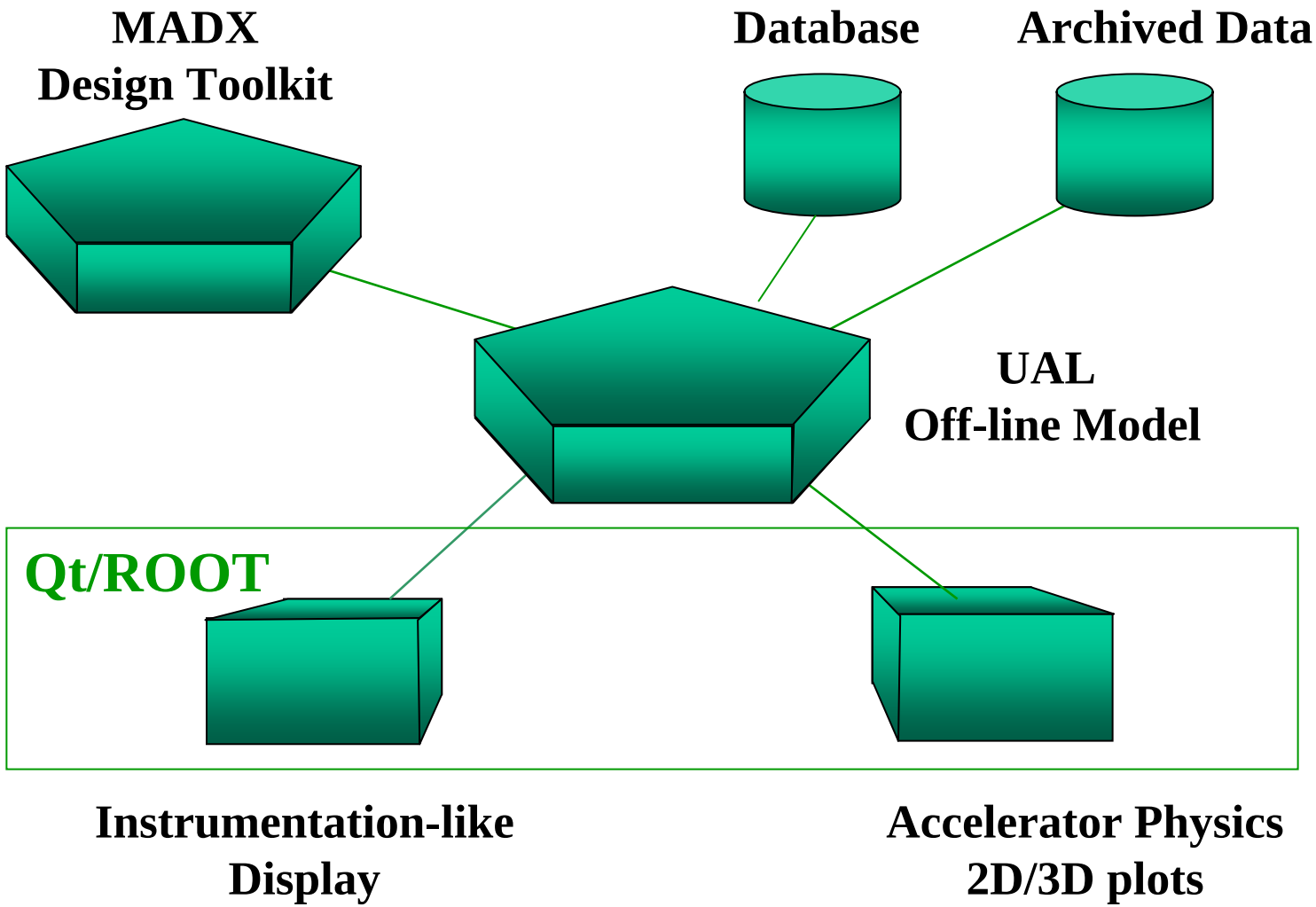
CINT/C++

1. Design of C++ application-specific classes
2. Implementation of shared library
3. Debugging
4. Test
5. Use with **light scripting** of the main function.

UAL Environment with CINT-based interface



Module File Structure

```
SIMBAD
├── Makefile
├── api
│   └── ROOT
│       ├── Makefile
│       ├── LinkDef.h
│       └── Headers
├── doc
├── include
├── lib
│   └── linux
│       ├── libRootSimbad.so
│       └── libSimbad.so
└── src
```

MADX-UAL-ROOT Off-line Facility