The Pivot framework: Design and Implementation

B. Stroustrup

G. Dos Reis

Department of Computer Science

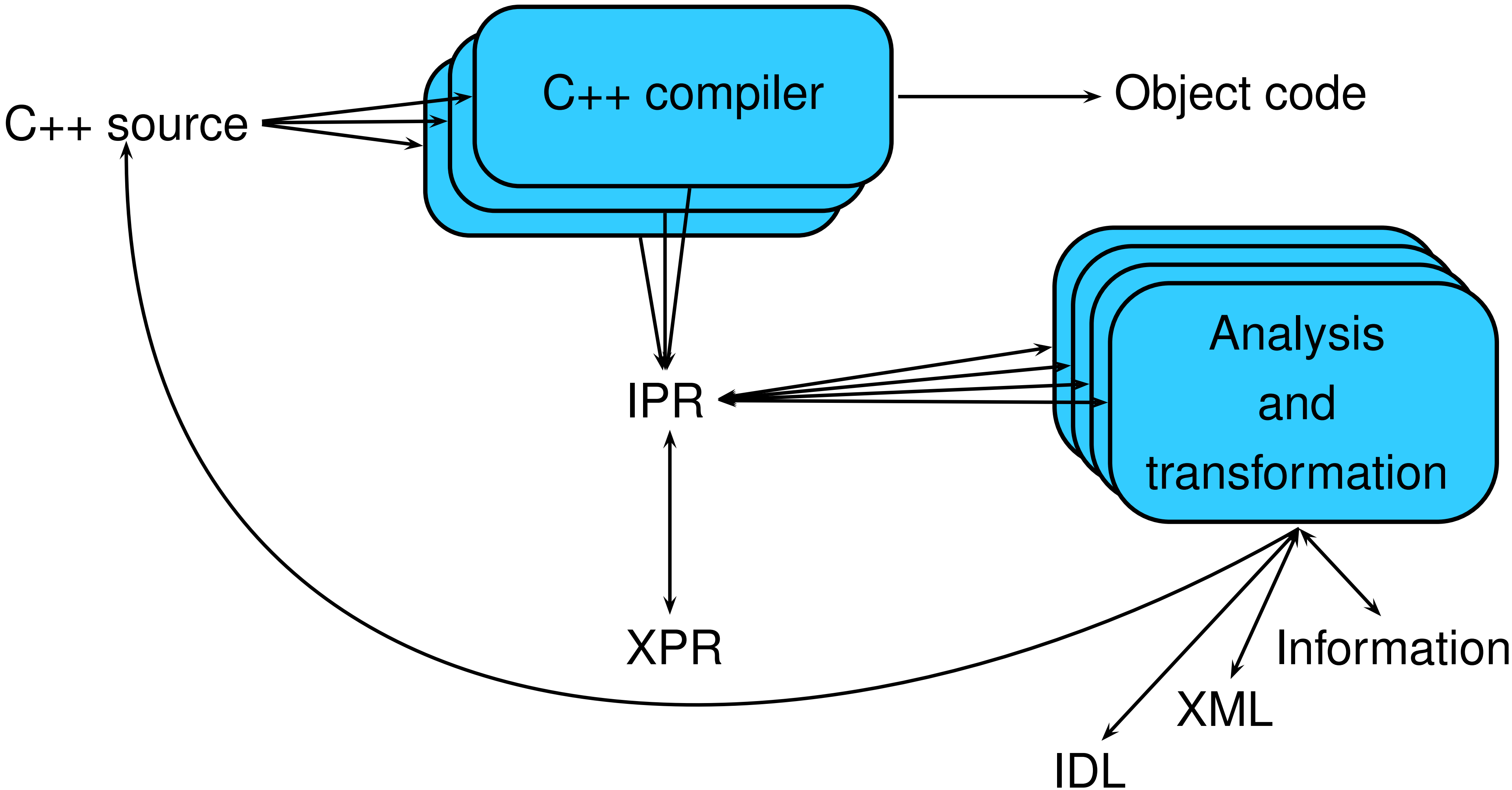
Texas A&M University

The Problem

- The original problem (inspiration)
  - Poor support for CORBA and for high-level parallel and distributed programming techniques
- No **widely-available** and general static analysis and transformation for C++
- There are many incomplete tools
- The community is fractured
  - Few dare to rely on other groups tools
- None of the existing tools deal with the **higher levels of C++** (templates, specialization, concepts)
  - Those are the aspects of C++ that are crucial for advanced optimization, validation of safety, enforcement of dialects, support of advanced libraries.

The Pivot

A framework for static analysis and transformation of C++



The Pivot parts

- IPR (Internal Program Representation)
  - a **fully general** typed abstract syntax tree representation of **all C++** (with the exception of macros)
  - has unified type system
  - is **prepared for C++0x facilities**, notably concepts
  - Potentially standard
- XPR (eXternal Program Representation)
  - Compact, persistent, user-readable, portable representation of IPR
- IPR $\Longleftrightarrow$ XPR parsers
- Traversal and transformation tools
- Specific tools
  - E.g. IPR $\Longleftrightarrow$ XPR, IPR $\Longleftrightarrow$ IDL, style checker, ...

IPR: Design rules

IPR should:

- be **complete** — represent all Standard C++ constructs
- be **general** — not targeted to a very small area of applications; must be useful to the wide C++ community
- be **regular** — must contain C++ but not mimic its irregularities; prefer general rule to long list of special cases
- put **emphasis on types** — those are verifiable comments; IPR nodes may be thought of as fully typed abstract syntax tree
- be **compiler neutral** — NOT tied to any particular compiler details or implementations.
- be **efficient and elegant**

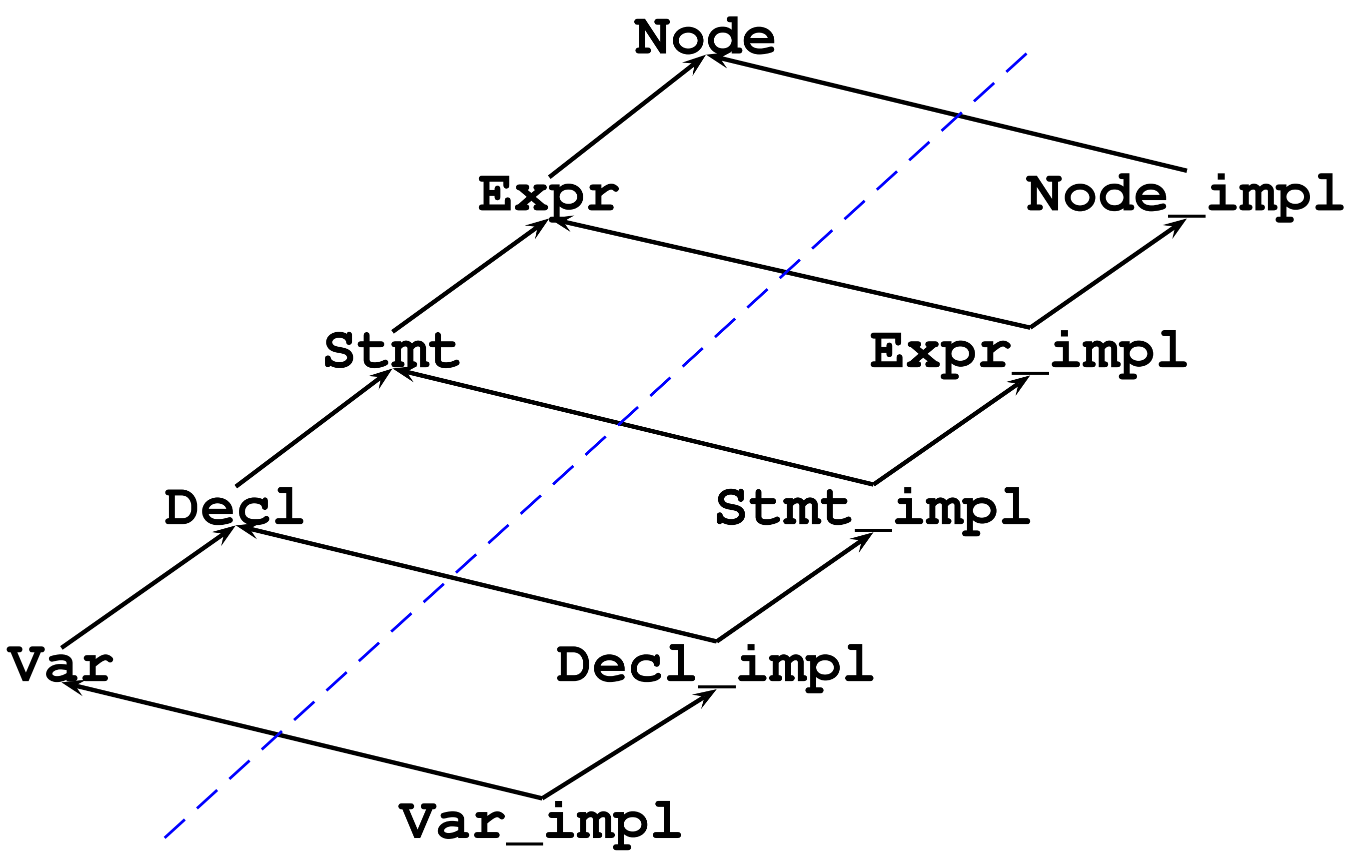
IPR interfaces

- The modeled language is expression-based
  - E.g. statements, declarations are expressions too
- Simple, can represent incomplete or erroneous programs
- Two interfaces: Properly encapsulate implementation details from users:
  1. Purely “functional”, abstract classes, for most users
     - No mutation operation on abstract classes
     - Users don’t get pointers directly
  2. Mutating (operates on “concrete” classes)
     - Users get to use pointers for in-place transformation
- Library (not users) deals with memory management
- Traversal or “climbing” is based on the Visitor Design Pattern

IPR implementation

Earlier attempts (including XTI):

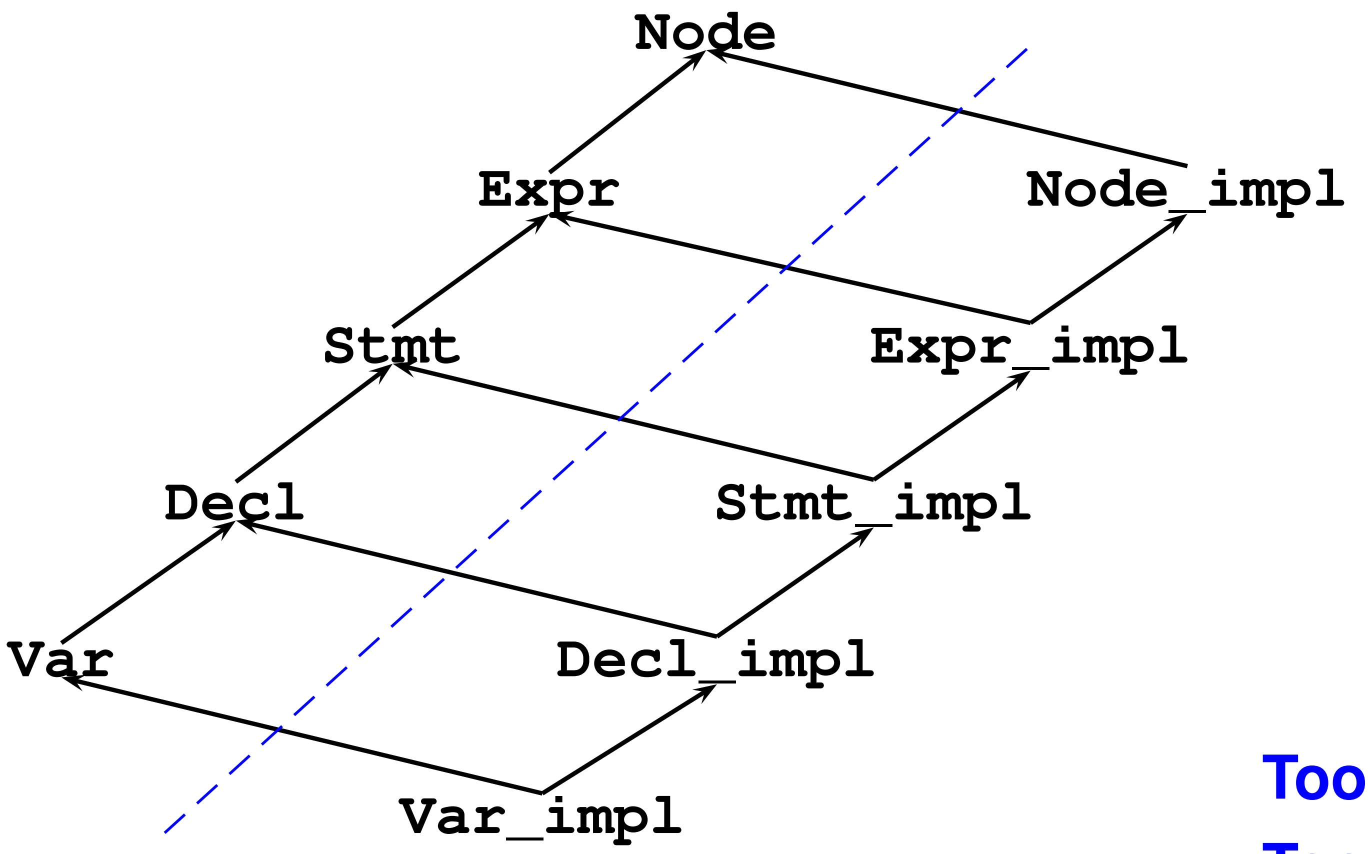
Every interface class **`Xyz`** should have a corresponding implementation class **`Xyz_impl`**.



IPR implementation

Earlier attempts (including XTI):

Every interface class **`Xyz`** should have a corresponding implementation class **`Xyz_impl`**.



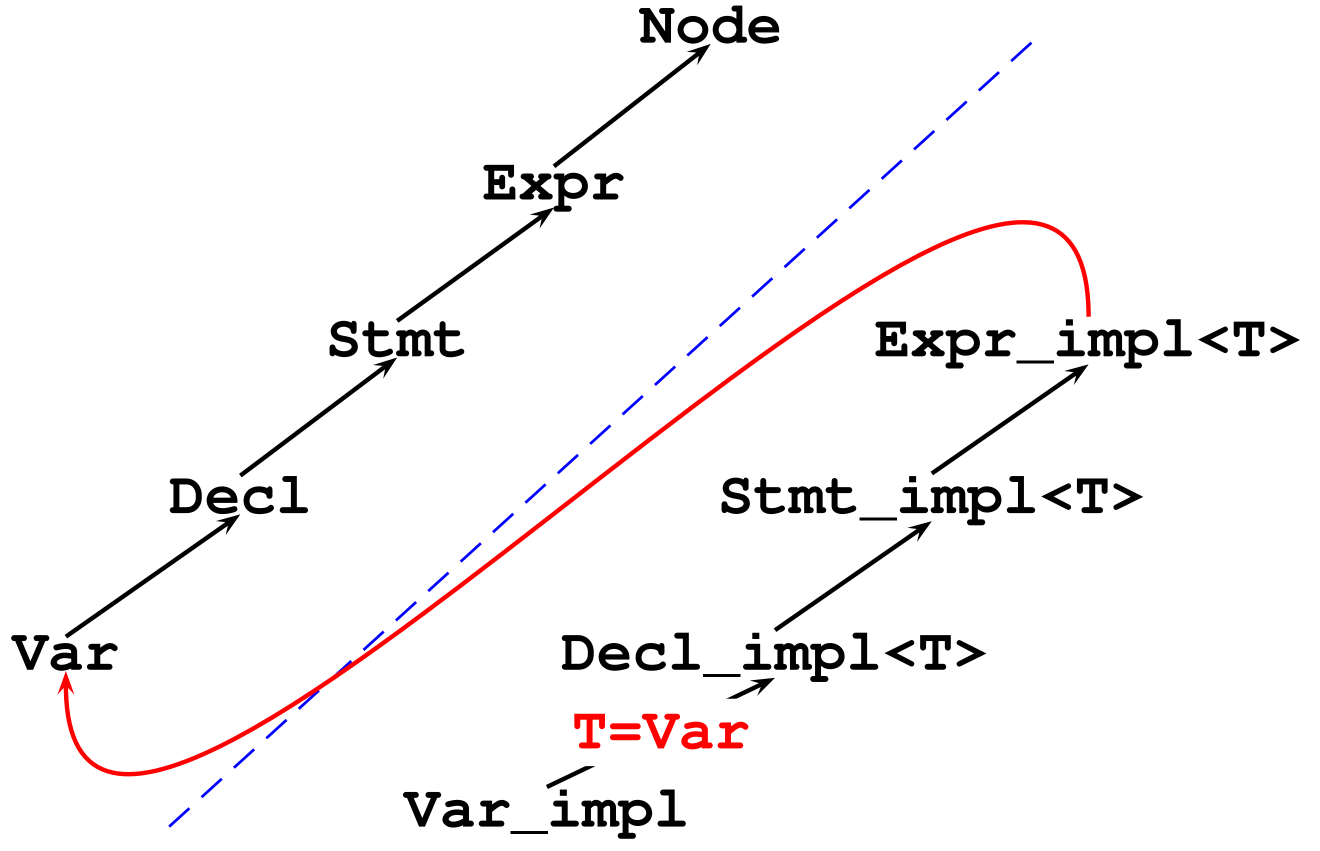
Too Complicated

Too slow

IPR implementation cont'd

Linearization:

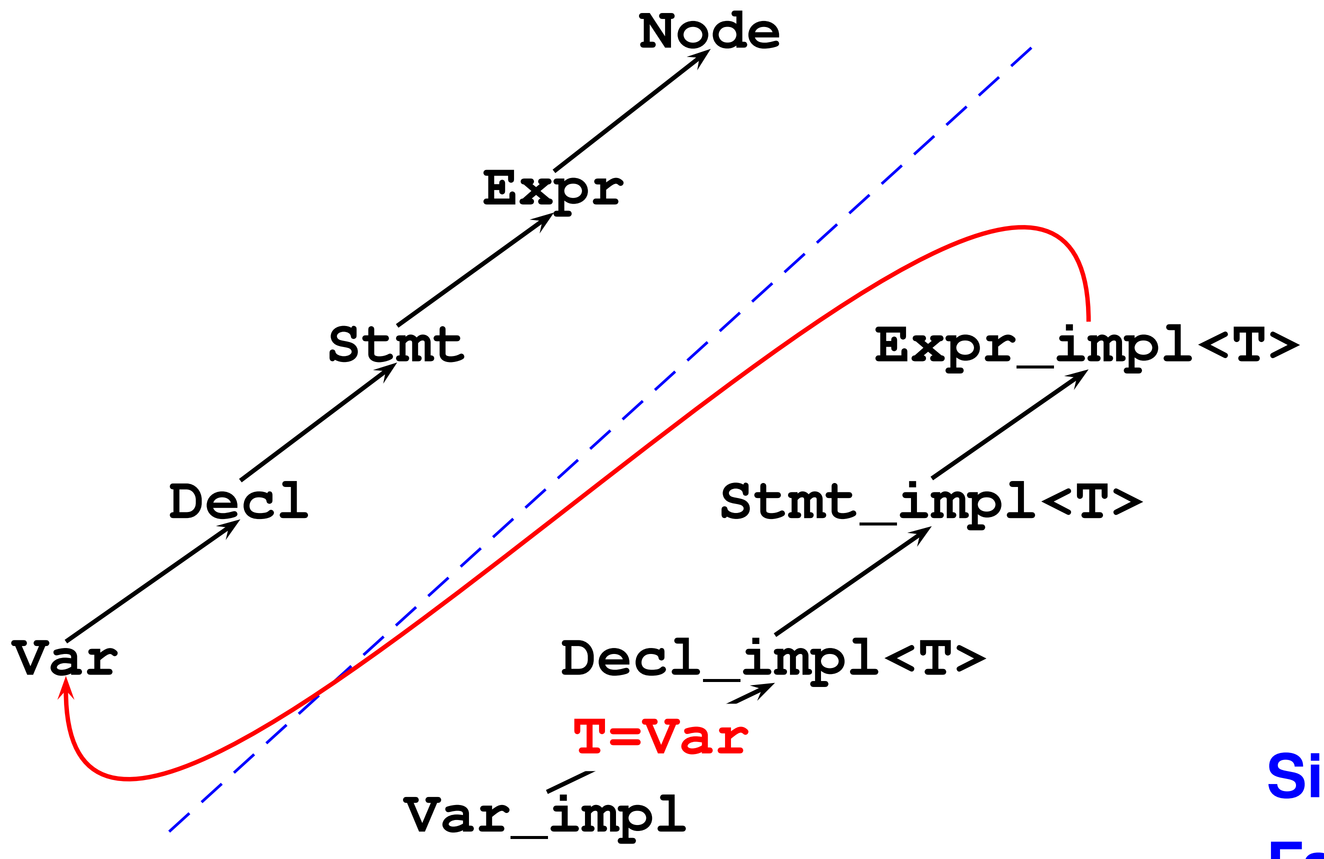
Parameterize implementations by interfaces



IPR implementation cont'd

Linearization:

Parameterize implementations by interfaces



Simpler

Faster

XPR: Persistent IPR

- be **simple** to process
  - XPR parsers should not duplicate work already done in C++ compilers;
- be **fast** to process
  - Ideally, close to Unix `cat` efficiency
- be **compact**, human readable
- reflect the **inner syntax** of Standard C++
- have parsers **easy to implement** with traditional tools
  - generated parsers (bottom-up, top-down), hand-written recursive-descent

Vec (C++)

```
template<class T>
struct Vec {
  Vec(int);
  T& operator[](int);
  const T& operator[](int) const;
  int size() const;
  // ...
private:
  T* data;
  int length;
};

template<class T>
Vec<T> operator+(const Vec<T>& u, const Vec<T>& v)
{
  Vec<T> w(u.size());
  for (int i = 0; i < u.size(); ++i)
    w[i] = u[i] + v[i];
  return w;
}
```

Vec (XPR)

```
Vec :<T :class> :class {
  #ctor :(this :*Vec<T>, n :int) throw(...) Vec<T> public;
  operator[] :(this :*Vec<T>, n :int) throw(...) &T public;
  operator[] :(this :*const Vec<T>, n :int) throw(...) &const T public;
  size :(this :*const Vec<T>) throw(...) int public;

  data :*T private;
  length :int private;
};

operator+ :<T :class> (u :&const Vec<T>, v :&const Vec<T>) throw(...)  Vec<T>
{
   w :Vec<T> = { u.size() };
   for (i :int = 0; i < u.size(); ++i)
     w[i] = u[i] + v[i];

   return w;
}
```

Connection with compilers

Currently, IPR generators are being developped with two compilers

- EDG front-end : aim full integration
  - (+) complete C++, well-documented, relatively easy to modify, can be compiled with a C++ compiler, high-level IR;
  - (-) clever "optimizations" built into the high-level IR $\Rightarrow$ missing some information contained in the input source
- GCC (debug info): initial proof of concept
  - (+) freely available;
  - (-) incomplete C++, undocumented (changing) formats, too much compiler low-level details, too incomplete (high-level) information contained in input source.

`Vec` *through EDG*

[P. Pirkelbauer, `operator+` member]

```
Vec :<T :class> class {
   #ctor :(this :*Vec<T>,  :int) Vec<T>;
   operator[] :(this :*Vec<T>,  :int) throw(...) &T;
   operator[] :(this :*const Vec<T>,  :int) throw(...) &const volatile T;
   size : (this :*Vec<T>) throw(...) int;
   operator+ : (this :*Vec<T>, v :&const volatile Vec<T>) throw(...) Vec<T>
   {
      w :public Vec<T> = size(this);
      for (i :public int = 0; i < size(this); ++i)
         w[i] = (*this)[i] + v[i];
      return w;
   };
   data :private *T;
   count :private int;
};
```

`Vec` through GCC debug info

```
operator+ :<T :class> (u :&const Vec<T>, v :&const Vec<T>) Vec<T> throw(...)
{
    {
        w :Vec<T> = { u.size() };
        for (i :int = 0; i < u.size(); ++i)
          {
              w[i] = u[i] + v[i];
          }
        return w;
    }
}
Vec :<T :class> :class {
  #ctor :(this :*Vec<T>, n :int) throw(...) Vec<T> public;
  operator[] :(this :*Vec<T>, n :int) throw(...) &T public;
  operator[] :(this :*const Vec<T>, n :int) throw(...) &const T& public;
  size :(this :*const Vec<T>) throw(...) int public;
  data :*T private;
  length :int private;
};
```

Future work

- Complete infrastructure
  - Represent header files directly in IPR/XPR
- Integrate “concepts”
- Style analysis
  - including type safety and security
- Analysis and transformation of STAPL programs

- Build alliances