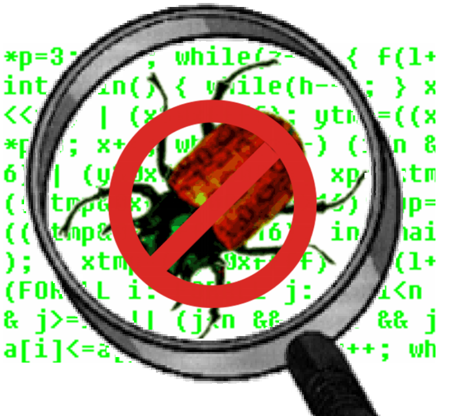
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Model Checking C++

Daniel Kroening



26.8.2004

Warning!

- No new research in this talk
- Talk is about doing <u>existing stuff</u> for different language only

➡ <u>You might consider this trivial and nod off</u>

Example from NASA JPL

```
class RCPNAM{
 public:
  RCPNAM();                                //!< DND for STRICT protocol.
  ~RCPNAM();                               //!< Decr cntr, delete if zero.
  RCPNAM(RCPNOD* p);                       //!< New ptr at given object.
  RCPNOD* operator->();
  RCPNOD& operator*();
  RCPNAM(const RCPNAM& rhs);               //!< Copy of existing RCPNAM.
  […]
  RCPNAM& operator=(const RCPNAM& rhs);    //!< Assign to existing RCPNAM.
  int nullp(){return ref_ == 0;};          //!< DND for STRICT.
  […]
 public:
  RCPNOD* ref_;
  static RCPNOD* copyAndInc(const RCPNOD*const* p,int RCPNOD::*cntr);
};
```

Example from NASA JPL

- Want to verify the code that goes into space
- Container class with reference counting
- Concurrent
- Mostly low-level, performance-oriented C and C++
- Uses assembly-language constructs for atomic accesses to pointers and counters

Example from NASA JPL

- Verification:
  - Extensive testing
  - SPIN failed
- Main challenge: pointers and references
- How many threads are enough?

Outline

1. Frontend
    - What’s so hard about C++?
    - Parsing, Type Checking
    - STL
2. Backend
    - Verification Backends
    - Dynamic Objects

What’s so hard about C++?

- **Existing model checkers: compile, and work on binary**
- **Infrastructure is difficult**
  - **Parsing is complicated (LR_k)**
  - **Complex name resolution rules (namespaces, templates, class hierarchy, overloading)**
- **But: there are tools that flatten C++ to C**

Why C++?

- Main advantage of C++:

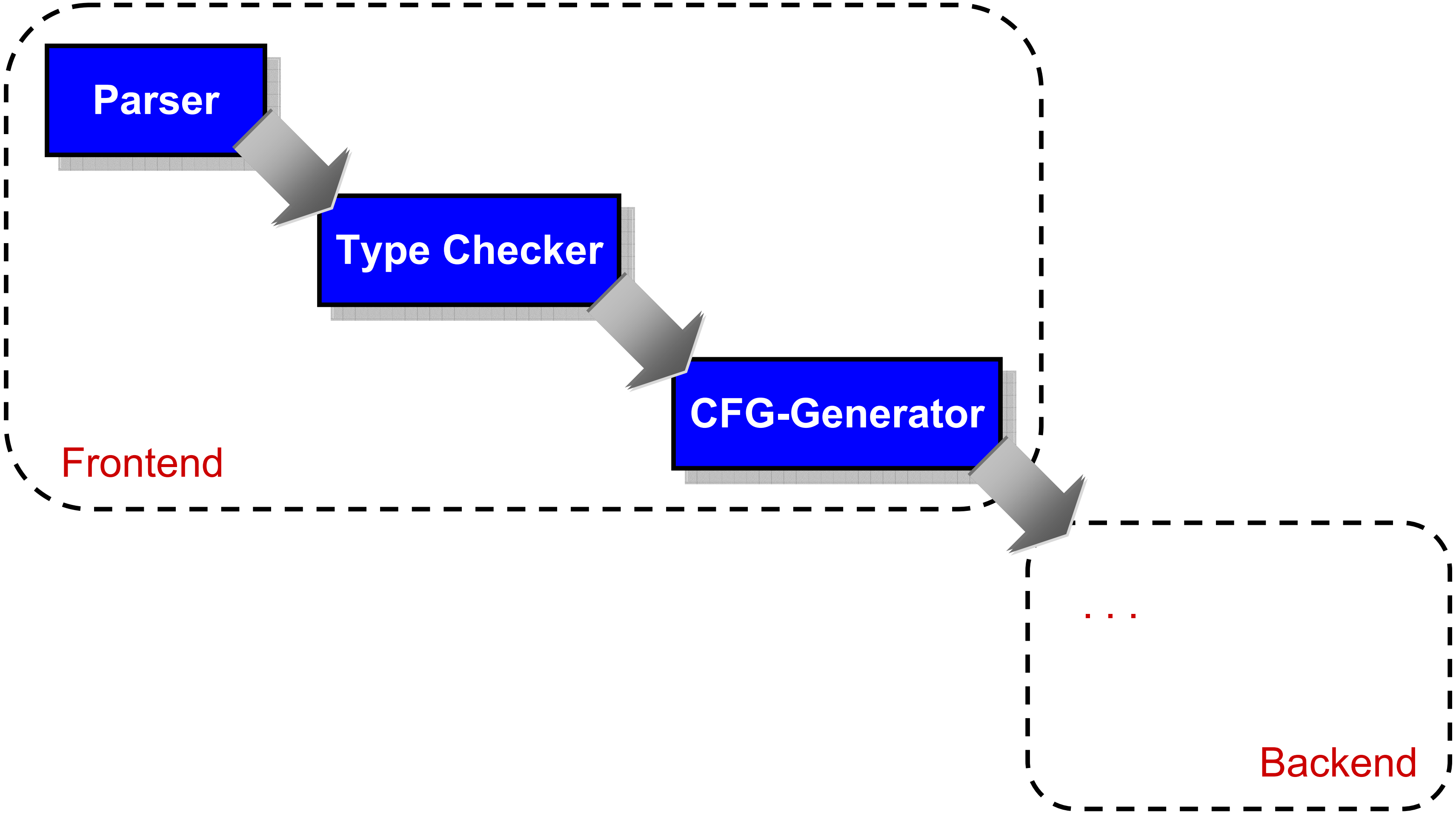
  Encapsulation of complex data structures in template libraries

- Also main challenge when model checking C++

What’s so hard about C++?

- **What kind of C++ are we going to see?**
  - **Heavy use of references and pointers**
  - **Class/Module hierarchy**
  - **Overloading**
  - **Templates**
  - **`new`/`delete`**

Model Extraction for C++



Parser

Type Checker

CFG-Generator

Frontend

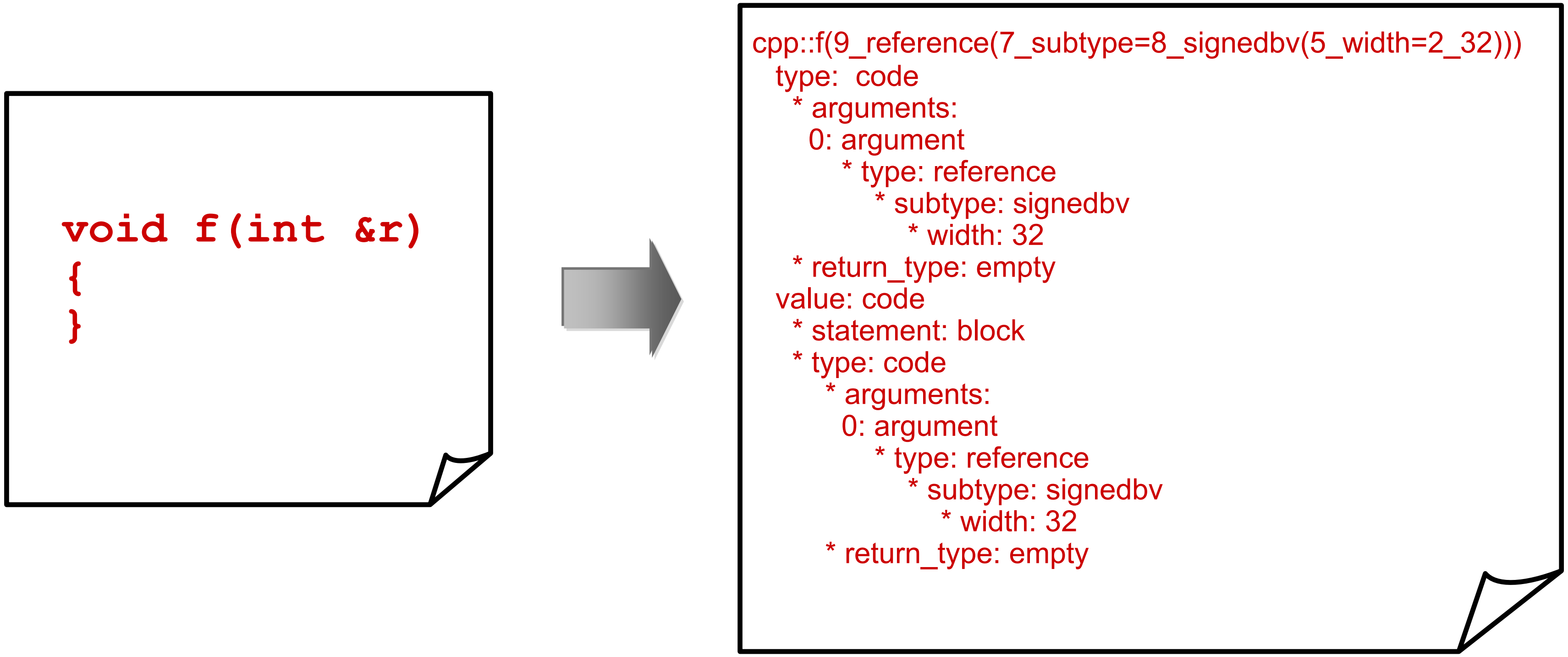
. . .

Backend

Type Checker

- Parse tree to [symbol table]
- Both represented as DAGs
- Algorithm:
  1. Expand templates
  2. Resolve overloading
  3. Class hierarchy
  4. Annotate each sub-expression with type

Type Checker



```
void f(int &r)
{
}
```

```
cpp::f(9_reference(7_subtype=8_signedbv(5_width=2_32)))
  type:  code
    * arguments:
      0: argument
          * type: reference
              * subtype: signedbv
                  * width: 32
    * return_type: empty
  value: code
    * statement: block
    * type: code
        * arguments:
          0: argument
              * type: reference
                  * subtype: signedbv
                      * width: 32
        * return_type: empty
```

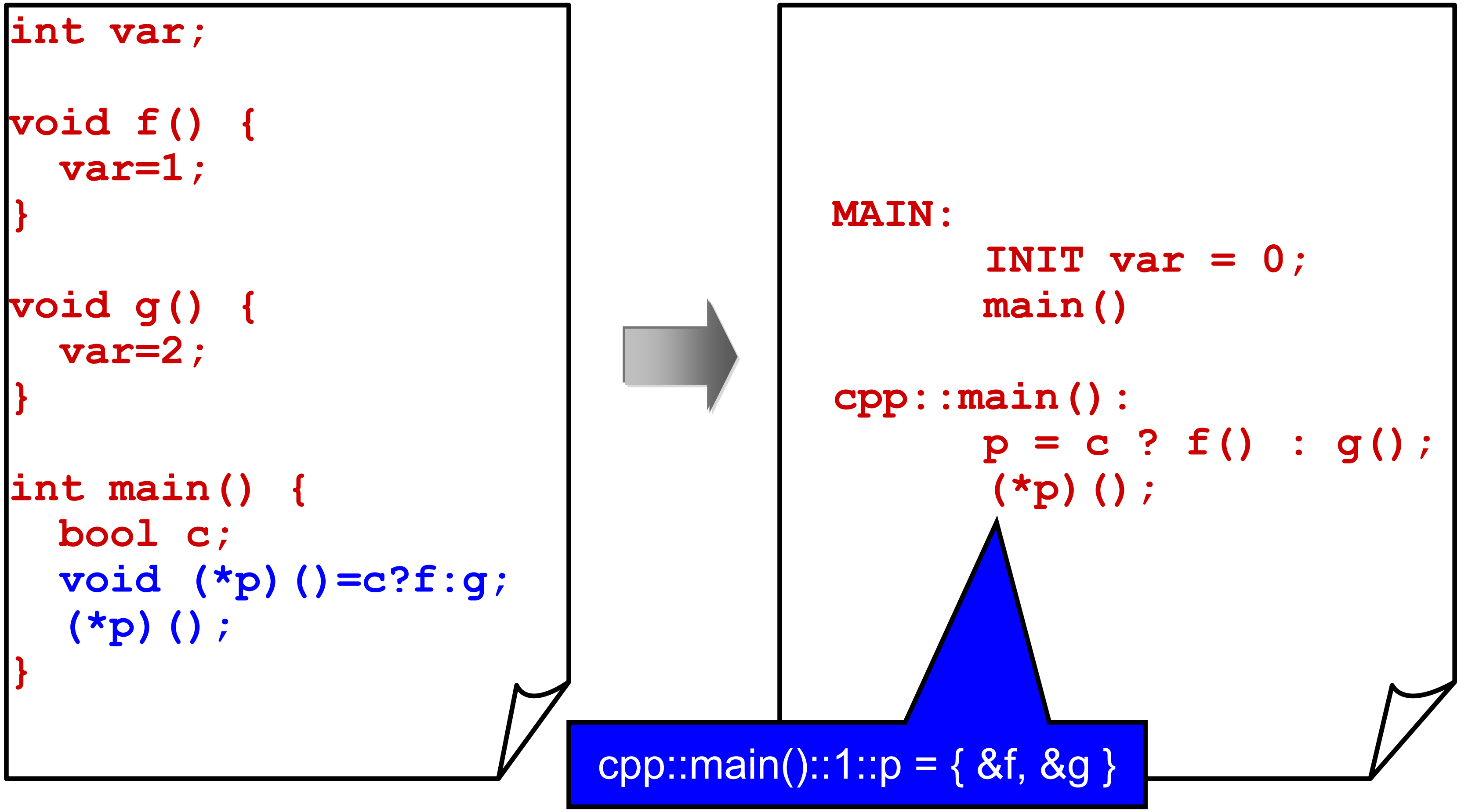
Control Flow Graph

- Symbol table to Control Flow Graph (CFG)
  - Essentially a guarded GOTO program, but with direct function calls
  - `virtual` methods and virtual classes and function pointers require alias analysis

Alias Analysis

- For
  - References
  - Pointers
  - `virtual` tables
- Fixed-point iteration on the CFG
  - → Interleaved with the computation of the CFG
- Control flow sensitive (concurrency!)
- Field sensitive

Alias Analysis



```
int var;

void f() {
  var=1;
}

void g() {
  var=2;
}

int main() {
  bool c;
  void (*p)()=c?f:g;
  (*p)();
}
```

```
MAIN:
      INIT var = 0;
      main()

cpp::main():
      p = c ? f() : g();
      (*p)();
```

cpp::main()::1::p = { &f, &g }

Alias Analysis

```
int var;

void f() {
  var=1;
}

void g() {
  var=2;
}

int main() {
  bool c;
  void (*p)()=c?f:g;
   (*p)();
}
```

```
MAIN:
        INIT var = 0;
        main();

cpp::f():
        var = 1;

cpp::g():
        var = 2;

cpp::main():
        p = c ? f : g;
        IF p != &g THEN GOTO 1
        g();
        GOTO 2
    1:  f();
    2:  SKIP
```

STL

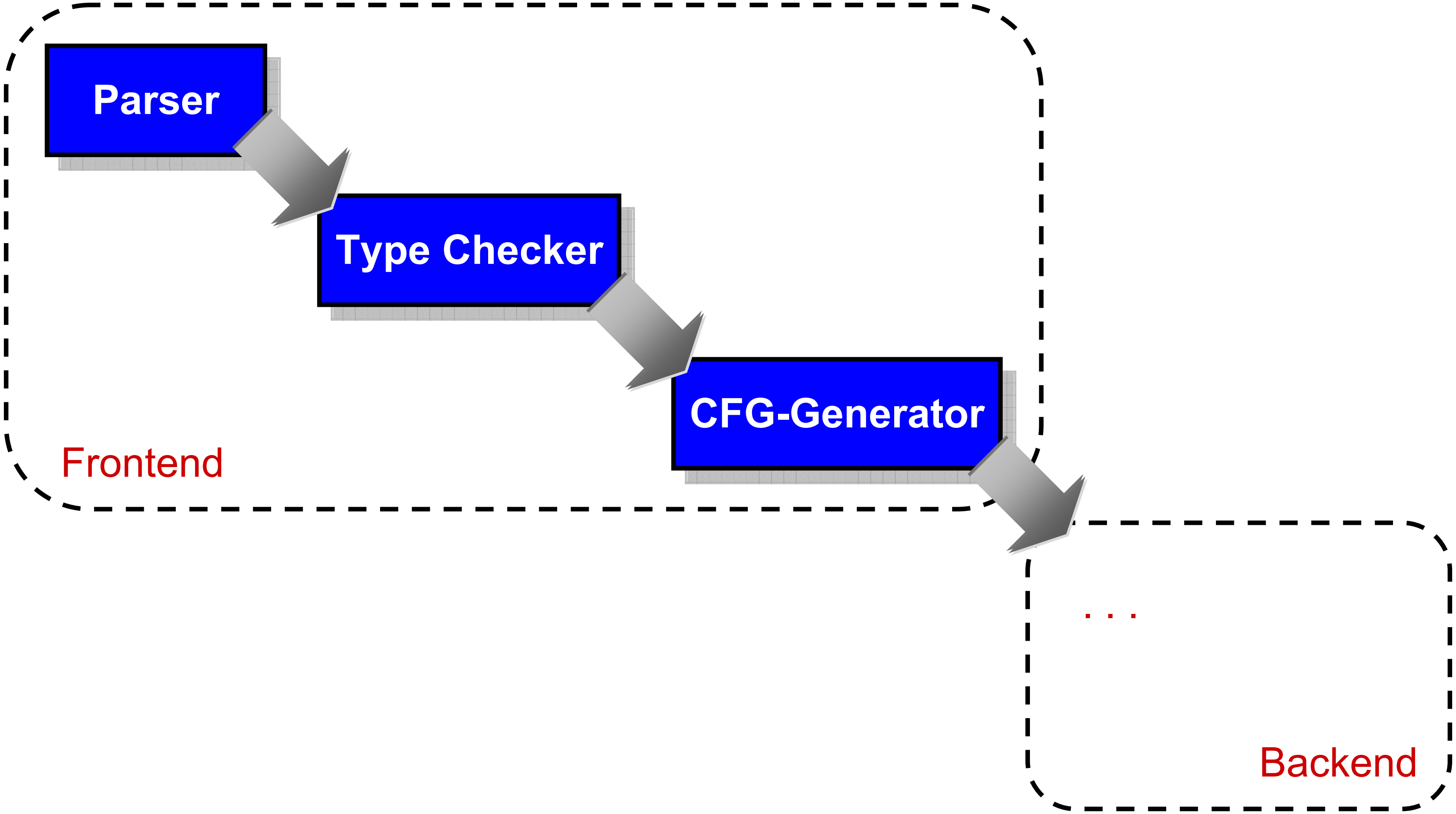
- **Standard Template Library**
- **Encapsulates complex data structures and algorithms**

```
typedef std::hash_map
  <std::string, symbolt, string_hash> symbolst;

. . .

typedef std::vector<nodet> nodest;
```

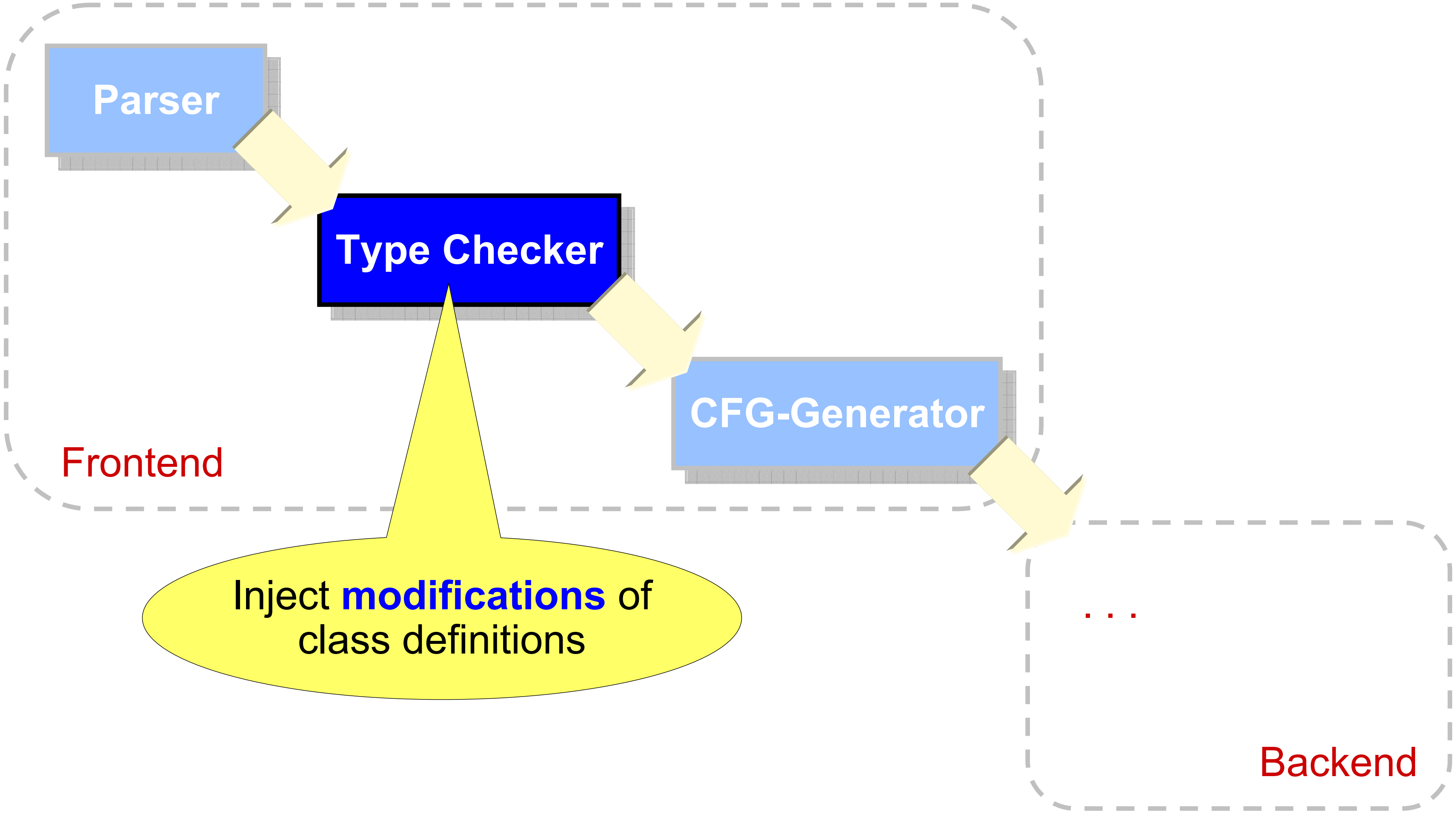
STL

- “Interesting” programs using STL have > 1000 data structures
- Flatten to C?
  - STL implementation highly complex and optimized
  - Don’t want to verify STL together with program
- Let’s assume STL is correct

Model Extraction for C++



Model Extraction with the STL


Parser

Type Checker

CFG-Generator

Frontend

Inject **modifications** of class definitions

. . .

Backend

Abstract STL

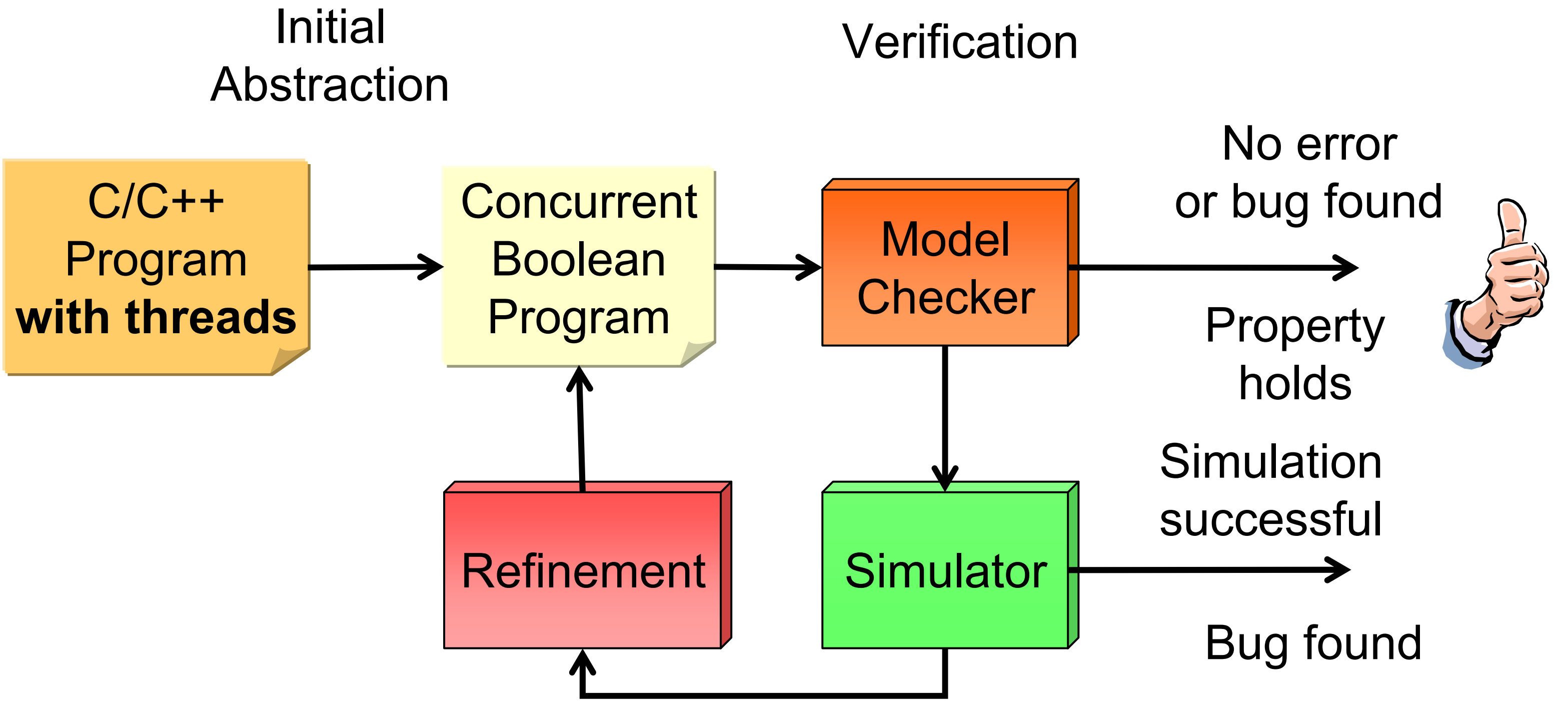
- Manually written abstractions of common STL data types
  - `std::vector`
  - `std::list`
  - `std::set`
  - `std::map`
- Catch errors when using STL
- Catch errors in program that depend on data in containers

STL

```
typedef std::vector<…> T;
T v;

v.push_back(…);
v.reserve(2);

T::const_iterator it=v.begin();

x=*it; ✓

v.push_back(…);

x=*it; ⚡
```

Predicate Abstraction

- **Predicate Abstraction is a successful method to verify programs**



Initial Abstraction

Verification

C/C++ Program **with threads**

Concurrent Boolean Program

Model Checker

No error or bug found

Property holds

Refinement

Simulator

Simulation successful

Bug found

Dynamic Objects

- C++ code tends to make excessive use of dynamic objects
- Algorithm:
  - Allow `*` and `&` in predicates, including pointer arithmetic
  - New: also have quantifiers ∀, ∃
  - Maintain active bit $\alpha(o)$ and object size state variables
  - Flow control-sensitive points-to analysis

Dynamic Objects

```
struct s {
  s *n;
  int i;
} *p;
...
```

	Preconditions	Postconditions
`p=new s;`	–	$\alpha(*p)$
`p->n=new s;`	$\alpha(*p)$	$\alpha(*p)$, $\alpha(*(p\text{->}n))$
`p->n->i=p->i+1;`	$\alpha(*p)$, $\alpha(*(p\text{->}n))$	$p\text{->}n\text{->}i=p\text{->}i+1$

Questions?