Concept Recovery from C++ Generic Functions

Peter Pirkelbauer[1] Damian Dechev[2] Bjarne Stroustrup[1]

[1]Texas A&M University

[2]University of Central Florida

Software Language Engineering 2010

The C++ Standard Template Library (Overview)

Parasol

STL

- Data structures (vector, list, deque, map, set, etc.)
- Algorithms

STL Principles

- Generic Programming
  - Type-safe
  - Built-in memory management
  - High performance
- Concept based library
  - Iterators, Functions, Predicates, Containers, . . .
  - Formalized for C++0x: Gregor et al. [3]

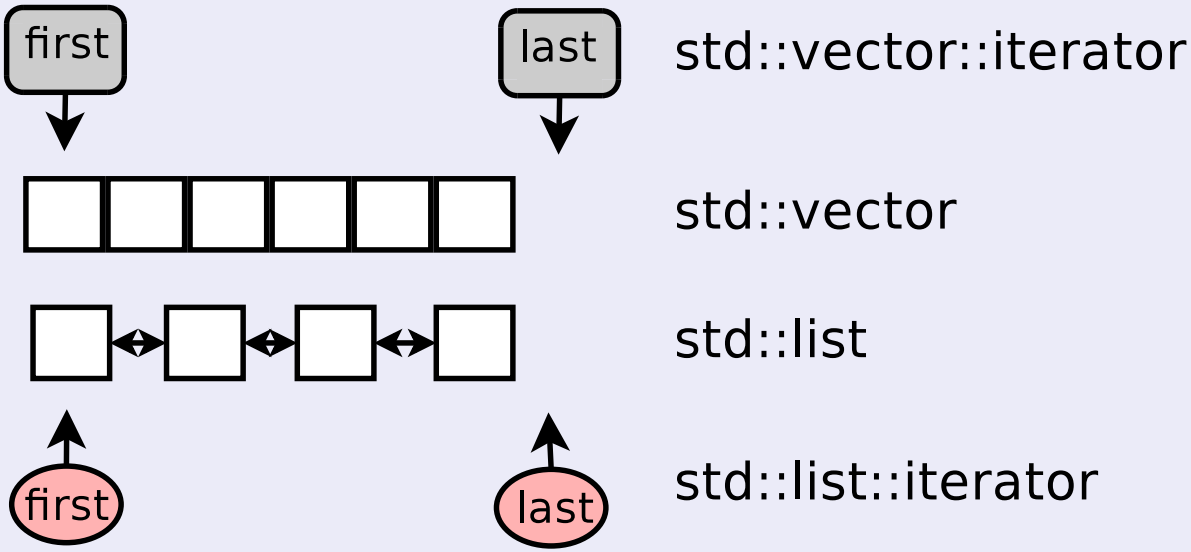
The C++ Standard Template Library (Iterators)

Parasol

std::find

```
template<typename InputIter, typename Pred>
InputIter find_if(InputIter first, InputIter last, Pred pred) {
   while (first != last && !pred(*first)) ++first;
   return first;
}
```

Iterators



Concept Recovery from C++ Function Template

Parasol

Motivation

- Template library designers use C++ concepts
- Concepts are not directly expressible in code
- Tool support to analyze template based libraries
  - Extract requirements
  - Analyze requirements

C++ Concepts - Requirement Collection

Unconstrained code

```
template<typename InputIter, typename Pred>
InputIter find_if(InputIter first, InputIter last, Pred pred) {
  while (first != last && !pred(*first)) ++first;
  return first;
}
```

Syntactic Requirements on Concepts

C++ Concepts - Requirement Collection

Unconstrained code

```
template<typename InputIter, typename Pred>
InputIter find_if(InputIter first, InputIter last, Pred pred) {
  while (first != last && !pred(*first)) ++first;
  return first;
}
```

Syntactic Requirements on Concepts

```
concept FindIf <typename InpIter, typename Pred> {
  InpIter::InpIter(const InpIter&); // copy constructor
  ...
}
```

C++ Concepts - Requirement Collection

Unconstrained code

```
template<typename InputIter, typename Pred>
InputIter find_if(InputIter first, InputIter last, Pred pred) {
  while (first != last && !pred(*first)) ++first;
  return first;
}
```

Syntactic Requirements on Concepts

```
concept FindIf <typename InpIter, typename Pred> {
  ...
  typename R0; // associated type
  R0 operator!=(InpIter&, InpIter&);
  ...
}
```

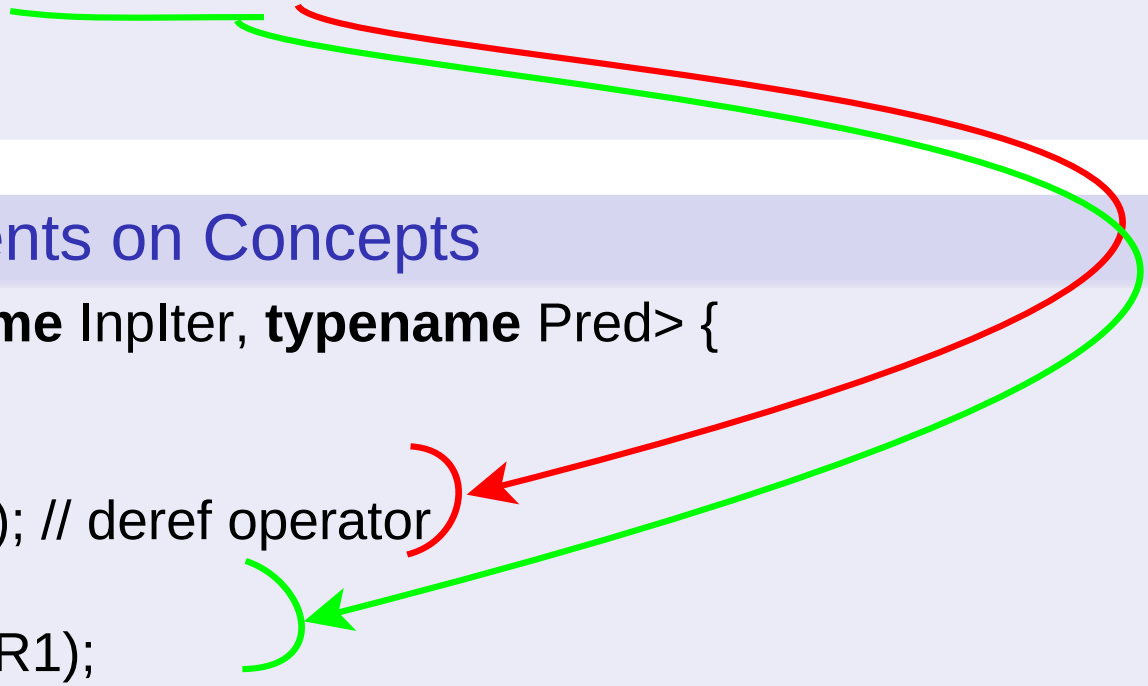
C++ Concepts - Requirement Collection

Unconstrained code

```
template<typename InputIter, typename Pred>
InputIter find_if(InputIter first, InputIter last, Pred pred) {
  while (first != last && !pred(*first)) ++first;
  return first;
}
```

Syntactic Requirements on Concepts

```
concept FindIf <typename InpIter, typename Pred> {
  ...
  typename R1;
  R1 operator*(InpIter&); // deref operator
  typename R2;
  R2 operator()(Pred&, R1);
  ...
}
```



C++ Concepts - Requirement Collection

Unconstrained code

```
template<typename InputIter, typename Pred>
where FindIf<InputIter, Pred>
InputIter find_if(InputIter first, InputIter last, Pred pred) {
  while (first != last && !pred(*first)) ++first;
  return first;
}
```

Syntactic Requirements on Concepts

```
concept FindIf <typename InpIter, typename Pred> {
  InpIter::InpIter(const InpIter&); // copy constructor
  typename R0; // associated type
  R0 operator!=(InpIter&, InpIter&);
  typename R1;
  R1 operator*(InpIter&); // deref operator
  ...
}
```

Concept recovery

Extracted requirements

```
concept FindIf <typename InpIter, typename Pred> {
  InpIter::InpIter(const InpIter&); // copy constructor
  Pred::Pred(const Pred&); // copy constructor
  typename R0; // associated type
  R0 operator!=(InpIter&, InpIter&);
  typename R1;
  R1 operator*(InpIter&); // deref operator
  typename R2;
  R2 operator()(Pred&, R1);
  typename R3;
  R3 operator!(R2);
  typename R4;
  R4 operator&&(R0, R3);
  operator bool (R4); // conversion requirement
  typename R5;
  R5 operator++(InpIter&);
}
```

Concept recovery

Extracted requirements

```
concept FindIf <typename InpIter, typename Pred> {
  InpIter::InpIter(const InpIter&); // copy constructor
  Pred::Pred(const Pred&); // copy constructor
  typename R0; // associated type
  R0 operator!=(InpIter&, InpIter&);
  typename R1;
  R1 operator*(InpIter&); // deref operator
  typename R2;
  R2 operator()(Pred&, R1);
  typename R3;
  R3 operator!(R2);
  typename R4;
  R4 operator&&(R0, R3);
  operator bool (R4); // conversion requirement
  typename R5;
  R5 operator++(InpIter&);
}
```

unordered set of requirements

requirements for a specific implementation

Cluster requirements

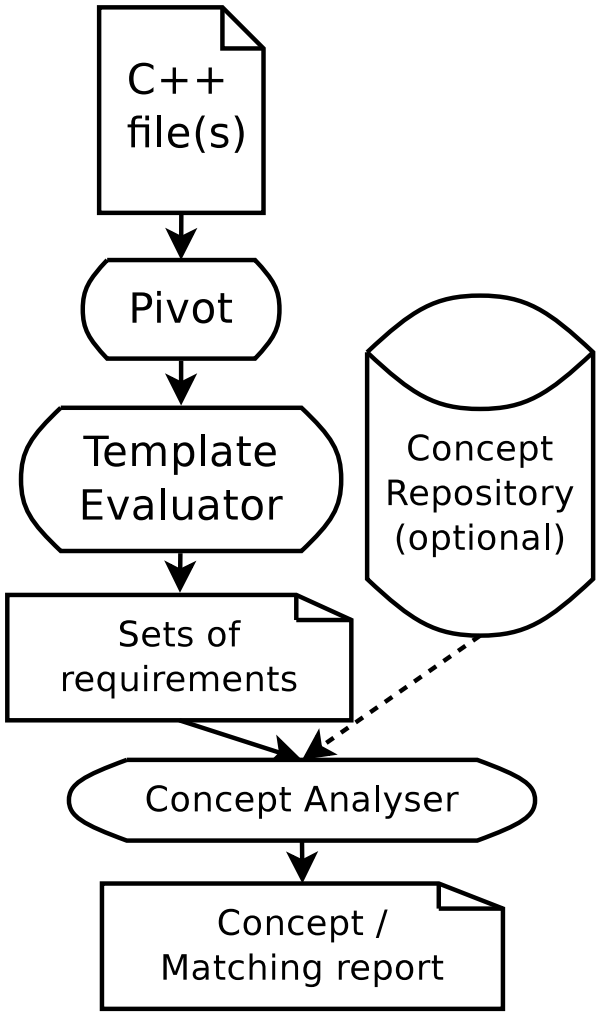
Task

Identify sets of requirements that are meaningful in a broader context.

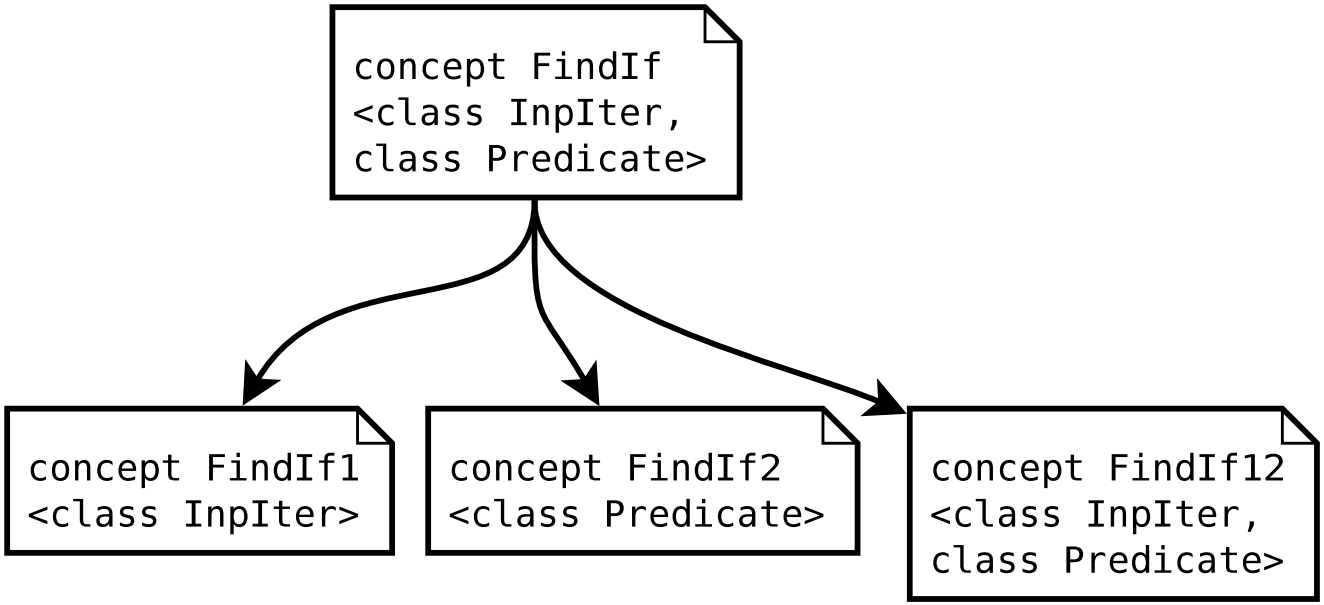
Approach

Concept repository (e.g., containing STL's standard concepts).

Tool Architecture



Partitioning



Matching (example)

Parasol

Partition InpIter

```
concept FindIf <typename InpIter>
{
  InpIter::InpIter(const InpIter&);
  typename R0;
  R0 operator!=(InpIter&, InpIter&);
  typename R1;
  R1 operator*(InpIter&);
  typename R5;
  R5 operator++(InpIter&);
}
```

Concept

```
concept ForwardIterator<typename T>
{
  // associated types
  typename T::value_type;
  typename T::difference_type;
  ...
  // Regularity
  T::T(); // default ctor
  T::T(const T&); // copy ctor
  bool operator!=(const T&, const T&);
  T& operator=(T&, const T&);
  ...
  // Iterator operations
  T::value_type& operator*(const T&);

  // Forward operations
  T& operator++(T&);
  T operator++(T&, int);
}
```

Matching (example)

Partition InpIter

```
concept FindIf <typename InpIter>
{
  InpIter::InpIter(const InpIter&);
  typename R0;
  R0 operator!=(InpIter&, InpIter&);
  typename R1;
  R1 operator*(InpIter&);
  typename R5;
  R5 operator++(InpIter&);
}
```

Concept

```
concept ForwardIterator<typename T>
{
  // associated types
  typename T::value_type;
  typename T::difference_type;
  ...
  // Regularity
  T::T(); // default ctor
  T::T(const T&); // copy ctor
  bool operator!=(const T&, const T&);
  T& operator=(T&, const T&);
  ...
  // Iterator operations
  T::value_type& operator*(const T&);

  // Forward operations
  T& operator++(T&);
  T operator++(T&, int);
}
```

Matching (example)

Partition InpIter

```
concept FindIf <typename InpIter>
{
  InpIter::InpIter(const InpIter&);
  typename R0;
  R0 operator!=(InpIter&, InpIter&);
  typename R1;
  R1 operator*(InpIter&);
  typename R5;
  R5 operator++(InpIter&);
}
```

Concept

```
concept ForwardIterator<typename T>
{
  // associated types
  typename T::value_type;
  typename T::difference_type;
  ...
  // Regularity
  T::T(); // default ctor
  T::T(const T&); // copy ctor
  bool operator!=(const T&, const T&);
  T& operator=(T&, const T&);
  ...
  // Iterator operations
  T::value_type& operator*(const T&);

  // Forward operations
  T& operator++(T&);
  T operator++(T&, int);
}
```

Matching (example)

Partition InpIter

```
concept FindIf <typename InpIter>
{
  InpIter::InpIter(const InpIter&);
  typename R0;
  R0 operator!=(InpIter&, InpIter&);
  typename R1;
  R1 operator*(InpIter&);
  typename R5;
  R5 operator++(InpIter&);
}
```

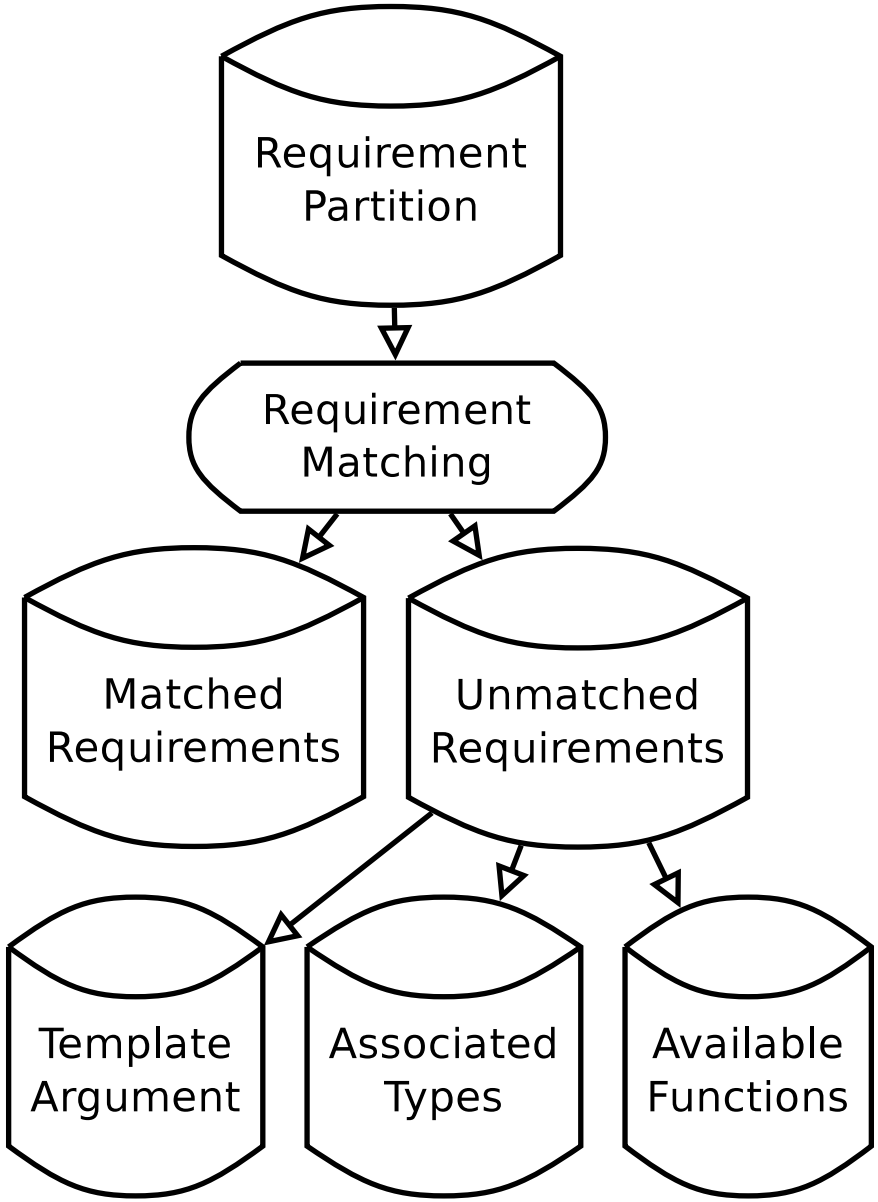
bool

Concept

```
concept ForwardIterator<typename T>
{
  // associated types
  typename T::value_type;
  typename T::difference_type;
  ...
  // Regularity
  T::T(); // default ctor
  T::T(const T&); // copy ctor
  bool operator!=(const T&, const T&);
  T& operator=(T&, const T&);
  ...
  // Iterator operations
  T::value_type& operator*(const T&);

  // Forward operations
  T& operator++(T&);
  T operator++(T&, int);
}
```

Matching results



Elimination of Candidate Concepts

Caller - callee relationships

- A concept valid for the caller must also be valid for the callee

Eliminate non-minimal concepts

- Remove any concept where a base-concept is a candidate

Limitations

Limitations

- Does not find combinations of concepts
  - sufficient for STL
- Considers only syntactic properties of types
  - respects const qualifier
- Considers conversion requirements defined in the concept
- Does not extract requirements from template classes

Validation - Setup

Repository

- TrivialIterator, ForwardIterator, BidirectionalIterator, RandomAccessIterator, Number, UnaryPredicate, BinaryPredicate, UnaryFunction, BinaryFunction

algorithm-header (GCC 4.1.3 Linux)

- contains >9000 non-empty, non-comment LOC
- defines 115 algorithms and 110 helper functions
- real code requires:
  - evaluation of trait classes
  - choose the base case of an algorithm family / trait class
  - extract requirements from static template member functions

Validation - Results

Results (overview)

- 70% correct and unambiguous recognition of iterator concepts
- 10% false positive match alongside the correct iterator concept
- 100% recognition of Predicates and Functions

Validation - Results (cont'd)

Results - details

- STL over-constrains some helper functions
  - __unguarded_linear_insert
- Syntactic requirements do not tell everything
  - 2^{nd} template argument of find_end for bidirectional-iterator
- Matching does not generate conversion requirements
  - fails on advance for random-access iterators

Parasol

Related Work

- Identifying concepts in C++ function templates (Sutton and Maletic [5])
- Concept specification (Dos Reis and Stroustrup [2])
- Type-checking of Haskell's type classes (Petersen and Jones [4]).
- Type inference for dynamically typed languages (Agesen et al. [1])

Conclusion and Future Work

Possible extensions

- Test with other concept based libraries
- Tool adaptations to help understand the nature of concepts

Thank You!

Agesen, O., Palsberg, J., and Schwartzbach, M.

Type inference of SELF.
In *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming* (London, UK, 1993), Springer, pp. 247–267.

Dos Reis, G., and Stroustrup, B.

Specifying C++ concepts.
In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2006), ACM Press, pp. 295–308.

Gregor, D., Järvi, J., Siek, J., Stroustrup, B., Dos Reis, G., and Lumsdaine, A.

Concepts: linguistic support for generic programming in C++.
In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (New York, NY, USA, 2006), ACM Press, pp. 291–310.

Peterson, J., and Jones, M.

Implementing type classes.
In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation* (New York, NY, USA, 1993), ACM, pp. 227–236.

Sutton, A., and Maletic, J. I.

Automatically identifying C++0x concepts in function templates.
In *ICSM '08: 24th IEEE International Conference on Software Maintenance, 2008, Beijing, China* (2008), IEEE, pp. 57–66.