Runtime Concepts for the C++ Standard Template Library

Peter Pirkelbauer
Texas A&M University
College Station, TX, U.S.A.
peter.pirkelbauer@tamu.edu

Sean Parent
Adobe Systems, Inc.
San Jose, CA, U.S.A.
sparent@adobe.com

Mat Marcus
Adobe Systems, Inc.
Seattle, WA, U.S.A.
mmarcus@adobe.com

Bjarne Stroustrup
Texas A&M University
College Station, TX, U.S.A.
bs@cs.tamu.edu

ABSTRACT

A key benefit of generic programming is its support for producing modules with clean separation. In particular, generic algorithms are written to work with a wide variety of unmodified types. The *Runtime concept* idiom extends this support by allowing unmodified concrete types to behave in a runtime polymorphic manner. In this paper, we describe one implementation of the runtime concept idiom, in the domain of the C++ standard template library (STL). We describe and measure the performance of runtime-polymorphic analogs of several STL algorithms. We augment the runtime concept idiom by employing a dispatch mechanism that considers both type and concept information to maximize performance when selecting algorithm implementations. We use our implementation to demonstrate the effects of different compile-time vs. run-time algorithm selection choices, and we indicate where improved language and compiler support would be useful.

Categories and Subject Descriptors

D.1.0 [**Programming Techniques**]: General

General Terms

Design, Languages

Keywords

Generic Programming, Runtime Polymorphism, C++, Standard Template Library

1. INTRODUCTION

ISO C++ [12] supports various programming paradigms, notably object-oriented programming and generic programming. Object-oriented techniques are used when runtime polymorphic behavior is desired. When runtime polymorphism is not required, generic programming is used, as it offers non-intrusive, high performance compile-time polymorphism; examples include the C++ Standard Template Library (STL) [5], the Boost Libraries [1], Blitz++ [17], STAPL [4].

Recent research has explored the possibility of a programming model that retains the advantages of generic programming, while borrowing elements from object-oriented programming, in order to support types to be used in a runtime-polymorphic manner. In [15], Parent introduces the notion of non-intrusive value-based runtime-polymorphism, which we will refer to as the *runtime concept* idiom. Marcus et al. [14], [3], and Parent [16] extend this idea, presenting a library that encapsulates the common tasks involved in the creation of efficient runtime concepts. Järvi et al. discuss generic polymorphism in the context of library adaptation [13].

A key idea in generic programming is the notion of a *concept*. A concept [10] is a set of semantic and syntactic requirements on types. Syntactic requirements stipulate the presence of operations and associated types. In the runtime concept idiom, a class R is used to model these syntactic requirements as operations. The binding from R to a particular concrete type T is delayed until runtime. Any type T that syntactically satisfies a concept's requirements can be used with code that is written in terms of the runtime concept.

In this paper, we apply these principles to develop a runtime-polymorphic version of some STL sequence containers and their associated iterators. Runtime concepts allow the definition of functions that operate on a variety of container types.

Consider a traditional generic function expressed using C++ templates:

```
// conventional template code
template <class Iterator>
Iterator
random_elem(Iterator first, Iterator last)
{
  typename Iterator::difference_type dist = distance(first, last);
  return advance(first, rand() % dist);
}
// ...
int elem = *random_elem(v.begin(), v.end()); // v is a vector of int
```

Objects of any type that meet the iterator requirement

SAC'08 March 16-20, 2008, Fortaleza, Ceará, Brazil

can be used as arguments to random_elem. However, those requirements cannot be naturally expressed in C++98, (though they can in C++0x), and the complete function definition is needed for type checking and code generation. The resulting code is very efficient, but this style of generic programming does not lend itself to certain styles of software development (e.g. those relying on dynamic libraries).

We can write essentially the same code using the runtime concept idiom:

```
// with runtime concept idiom
wrapper_forward<int>
random_elem(wrapper_forward<int> f, wrapper_forward<int> l)
{
  wrapper_forward<int>::difference_type dist = distance(f, l);
  return advance(f, rand() % dist);
}
// ...
int elem = *random_elem(v.begin(), v.end()); // v is a vector of int
```

However, here the binding between the iterator type and the function is handled at runtime and we can compile a use of random_elem with only the declarations of random_elem available:

```
// with runtime concept idiom:
wrapper_forward<int>
random_elem(wrapper_forward<int> f, wrapper_forward<int> l);
// ...
int elem = *random_elem(v.begin(), v.end()); // v is a vector of int
```

By using runtime concepts, function implementations (e.g.: random_elem) are isolated from client code. The parameter type wrapper_forward subsumes all types that model the concept forward-iterator. The implementation can be explicitly instantiated elsewhere for known element types, and need not be available to callers.

This reduced code exposure in header files makes runtime concepts suitable for (dynamically linked) libraries and when source code cannot be shared. However, the use of runtime concepts comes at a cost. The function random_elem is written in terms of the concept forward-iterator. The runtime complexity of distance and advance is $O(n)$ for forward-iterators, while it is constant time for randomaccess-iterators. Passing iterators of vector<int> as arguments would incur unnecessary runtime overhead.

This paper makes the following contributions:

- We apply the runtime concept idiom to part of a core C++ library (STL) and analyze the runtime overhead.
- We enhances the runtime concept idiom with a prototype of an open, extensible, and loosely coupled algorithm library– a runtime counterpart of the STL algorithms. Its dispatch mechanism selects the best matching algorithm instance according to runtime concept and type information of the actual atguments. This eliminates the need for dynamic dispatch when a matching algorithm instance is present. Runtime analogs of four STL algorithms are presented and their performance is analyzed.
- We explore the problem of writing single algorithms that can simultaneously accommodate runtime-polymorphic variations in both container and element type. Performance measurements indicate that current language level support is not sufficient to support this idiom, but we point out where language enhancements might make it viable in the future.

The structure of this paper is: sections 2 and 3 revisit fundamental ideas of generic programming and illustrate the runtime concept idiom in the STL domain. Section 4 discusses our concrete application to the STL's sequences and algorithms. Section 5 evaluates our prototype implementation and its runtime performance; section 6 compares our model to alternatives; section 7 points to possible extensions and summarizes our contribution.

2. GENERIC PROGRAMMING

The ideal for generic programming is to represent code at the highest level of abstraction without loss of efficiency in both actual execution speed and resource usage compared to the best code written through any other means. The general process to achieve this is known as *lifting*, a process of abstraction where the types used within a concrete algorithm are replaced by the semantic requirements of those types necessary for the algorithm to perform.

Semantic Requirement: Types must satisfy these requirements in order to work properly with a generic algorithm. Semantic requirements are stated in tables, in documentation, and may at times be asserted within the code. Checking types against arbitrary semantic requirements is in general undecidable. Instead, compilers for current C++ check for the presence of syntactic constructs, which are assumed to meet the semantic requirements.

Concept: Dealing with individual semantic requirements would be unmanageable for real code. However, sets of requirements can often be clustered into natural groups, known as *concepts*. Although any collection of requirements may define a concept, only concepts which enable new classes of algorithms are interesting.

Model: Any type that satisfies all specified requirements of a concept is said to be a model of that concept.

Generic Algorithm: A generic algorithm is a derivative of an efficient algorithm, whose implementation is independent from concrete underlying data structures. The requirements that an algorithm imposes on a data structure can be grouped into concepts. An example that is part of the STL is find and the requirement on the template argument is to model forward-iterator.

Concept Refinement: A concept C_r that adds requirements to another concept C_0 is a concept refinement. When compared to C_0, the number of types that satisfy the requirements of C_r decreases, while the number of algorithms that can be directly expressed increases. For example, constant time random access, a requirement added by the concept randomaccess-iterator, enables the algorithm sort.

Algorithm Refinement: Parallel to concept refinements, an algorithm can be refined to exploit the stronger concept requirements and achieve better space- and/or runtime-efficiency. For example, the complexity of reverse for bidirectional-iterator is $O(n)$, while it is $O(n\ lg\ n)$ for forward-iterator (assuming less than $O(n)$ memory usage).

Regularity: Dehnert and Stepanov [9] define regularity based on the semantics of built-in types, their operators, the complexity requirements on the operators, and consistency conditions that a sequence of operations has to meet. Regularity is based on value-semantics and requires operations to construct, destruct, assign, swap, and equality-compare two instances of the same type. This is sufficient for a number of STL data structures and algorithms includ-

ing vector, queue, reverse, find. A stronger definition adds operations to determine a total order, which enables the use of STL's map, set, sort. Code written with built-in types in mind will work equally well for regular user defined types. Programmers' likely familiarity with built-in types makes the notion of regularity important.

3. RUNTIME CONCEPTS

In order to make our exposition as self-contained as possible, and to allow us to experiment with potential improvements, we have implemented and will illustrate the runtime concept idiom with hand constructed classes. For a deeper treatment of the runtime concept idiom, along with its library support and optimizations see Marcus et al. [14] and [3].

The runtime concept idiom employs a three-layer architecture, the concept layer, the model layer, and the wrapper layer. The following example explains the interaction of these three layers based on a *runtime concept* Copyable, that supports copy construction. The operational requirements of runtime concepts are expressed with an abstract base class. Here, the runtime concept for Copyable requires the single operation clone.

```
struct concept_copyable {
  virtual concept_copyable& clone() const = 0;
};
```

Concrete types T and runtime concepts are loosely coupled by means of the runtime model layer. By a *runtime model*, we mean a class template M parametrized on T and inheriting from the runtime concept. A model holds the data and implements the pure virtual functions declared in the runtime concept by forwarding the calls to T:

```
template <class T>
struct model_copyable : concept_copyable {
  model_copyable(const T& val) : t(val) {}
  model_copyable& clone() { return *new model_copyable(t); }
  T t;
};
```

The *wrapper* layer wraps concept_copyable objects and manages their lifetime. It contains operations that guarantee regular semantics (constructor, destructor, etc.) and makes other operations accessible through the dot operator. Here, the implementations of the copy constructor and the assignment operator makes use of the function clone.

```
struct wrapper_copyable {
  template<typename T>
  wrapper_copyable(const T& t) : c(new model_copyable<T>(t)) {}
  // exemplary regular operations
  ~wrapper_copyable() { delete c; }
  wrapper_copyable& operator=(const wrapper_copyable&);
  // use of the concept interface
  wrapper_copyable(const wrapper_copyable& rhs)
  : c(&rhs.c->clone()) {}
  concept_copyable* c;
};
```

That is, wrapper_copyable is what a user can use to create objects that can be copied. for example:

```
wrapper_copyable val(1); // stores 1 in val
wrapper_copyable copy(val); // creates a copy of val
```

The wrapper_copyable constructor deduces the type of the model_copyable instance needed to access the data through concept_copyable. It also creates the model_copyable object needed to hold the value.

4. RUNTIME POLYMORPHIC STL

This section presents our implementation of the runtime concept idiom for iterators and several loosely coupled algorithms. The same techniques can be applied to provide runtime polymorphic STL containers.

4.1 Runtime Concepts of Iterators

To allow for the modeling of iterator concepts, we must extend the idiom from §3 to support concept-layer and model-layer refinements. Following Marcus et al. [14], we employ inheritance in order to support concept and model refinements while minimizing source code duplication.

We illustrate the implementation of runtime concepts and models using the concept forward-iterator and its refinement bidirectional-iterator. For the wrapper class layer implementation we also refer the reader to [14].

4.1.1 Concept Interface Refinements

Runtime concept interfaces are essentially abstract base classes that define a set of function signatures but do not have data members. The code that follows omits the template parameters that corresponds to iterator::reference.

```
template <class ValueType>
struct concept_forward {
  virtual void operator++() = 0;
  virtual concept_forward& clone() const = 0;
  // ...
};

template <class ValueType>
struct concept_bidirectional : concept_forward<ValueType> {
  virtual void operator--() = 0;
  virtual concept_bidirectional& clone() const = 0;
  // ...
};
```

Refinement of runtime concepts are accomplished via inheritance from a base concept class and add new signatures or refine inherited signatures with a covariant return type. Prefix and postfix operators share the same member function declarations and return **void**. The semantically correct implementation of the return value is left to the wrapper classes. Types related to the elements stored inside the container are passed as template arguments (e.g.: value_type).

4.1.2 Model Refinements

Models implement the abstract operations of the runtime concept for concrete types. At the root of the model-hierarchy is the model_base that stores a copy of the concrete iterator.

```
template <class Iterator, class IterConcept>
struct model_base : IterConcept {
  Iterator it;
};
```

The first template argument determines the concrete iterator type. The second template argument corresponds to the concept interface that this model will implement. For an iterator of list<**int**>, these would be list<**int**>::iterator and concept_bidirectional<**int**>, respectively.

```
template <class Iterator, class IterConcept>
struct model_forward : model_base<Iterator,IterConcept> {
  IterConcept& clone();

  bool operator==(const concept_forward& rhs) const
  {
    assert(typeid(*this) == typeid(rhs));
    // ...
  }
  // ...
};

template <class Iterator, class IterConcept>
struct model_bidirectional : model_forward<Iterator,IterConcept> {
  void operator--();
  // ...
};
```

Each model refinement implements the operations defined in the corresponding concept interface. The meaning of the two template arguments Iterator and IterConcept is the same as for model_base. Binary operations (e.g.: **operator**==) require the second argument to have the same dynamic type as the receiver. Conversions could be encoded with double dispatch [7] but are currently not supported.

4.1.3 Model Selection

The function select_model selects the model refinement based on the iterator type tag of T.

```
template<class Iterator>
typename map_iteratortag_to_concept_interface<Iterator>::type*
select_model(const Iterator& it);
```

The template meta function map_iteratortag_to_concept_interface maps the iterator tag to a runtime model. For example, bidirectional_iterator_tag is mapped on model_bidirectional. select_model instantiates this model with the iterator type and the correct concept interface. For list<**int**> this would be model_bidirectional<list<**int**>::iterator, concept_bidirectional<**int**>>. Finally, select_model constructs the model on the heap, and returns a pointer to it.

4.2 The Algorithms Library

The algorithms library prototypes a runtime counterpart of several STL algorithms. Each function in the library originates from an algorithm instantiation with one of our iterator wrappers or a concrete iterator. By default, the library contains an instance for the weakest concept an algorithm supports. For example, the default entry for lower_bound would be instantiated with wrapper_forward. These minimal instantiations are meant to serve as fallback-implementations. To improve performance, the system integrator or even a (dynamically loaded) library can add more specialized functions.

The algorithms are defined in terms of existing STL algorithms and iterator-value_types (e.g.: algolib::advance<**int**>). Consider a library defined on advance and **int** that by default contains an instance for forward_iterator. The sample code adds one generic implementation for wrapper_randomaccess and a specific for list<**int**>::iterator.

```
// add generic implementation suitable for all randomaccess iterators.
algolib::add_generic<
    algolib::advance<int>, // library name
    wrapper_randomaccess<int> // iterator-type
  >();
// add specific implementation for std::list<int>.
algolib::add_specific<
    algolib::advance<int>, // library name
    std::list<int>::iterator // iterator-type
  >();
```

In addition, we provide library access functions with names that match their STL counterparts. The access functions are defined in the same namespace as the wrapper classes. Together with argument dependent look-up (ADL), this enables seamless integration of runtime concepts into user code. The code snippet shows a function that takes two iterator wrapers as arguments and calls the library access functions (i.e.: distance, advance).

```
wrapper_forward<int>
random_elem(wrapper_forward<int> f, wrapper_forward<int> l)
{
  wrapper_forward<int>::difference_type dist = distance(f, l);
  return advance(f, rand() % dist);
}
```

At runtime, a library call selects the best applicable function present based on the dynamic type of the model. Starting with the **typeid** of the actual iterator model, it walks the **typeid**s of the inheritance chain until it finds a suitable algorithm instance or the fallback implementation.

If in our example first and last wrap the concrete type std::list<**int**>::iterator, the dispatch mechanism will peel off all runtime concept layers and call std::advance with a std::list<**int**> iterator. In case the iterators in first and last belong to a std::vector, the runtime model is re-wrapped by a wrapper_randomaccess iterator and std::advance<wrapper_randomaccess> is invoked.

Although the dispatch mechanism is semantically equal to virtual function calls, we rejected alternative library designs, which would model algorithms as pure virtual functions that are declared in concept interfaces. This would break the separation between concept requirements and algorithms. Providing a new algorithm would require adding a new function signature to the concept interface, thereby breaking binary compatibility with existing applications. In addition, such a design would create a number of unused instantiated functions. For example, the class concept_forward would need virtual function declarations for all STL algorithms that are defined for forward iterators (e.g.: adjecent_find, destroy, equal_range, etc.). Consequently, the model classes would need to implement those functions regardless whether a specific program uses them or not.

4.3 Retroactive Runtime Concepts

The goal of retroactive concept modeling is to place requirements on the element type of containers after the container has been declared (and elements have been inserted). This would allow us to write code that operates on a variety of containers and element types. Instead of the introductory example (§1) we would like random_elem to simultaneously handle list<**int**>, vector<**double**>, etc.:

```
retrowrapper_forward<>
random_elem(retrowrapper_forward<> f, retrowrapper_forward<> l)
{
  retrowrapper_forward<>::difference_type dist = distance(f, l);
  return advance(f, rand() % dist);
}
```

For reads and writes through such iterators the well studied variance problems [11] apply. However, sequence-modifying operations (e.g.: sort, rotate, etc.) which only permute the

elements and element modifications through a runtime concept interface are type safe.

Since the concrete element type of the containers are not known, functions that access elements cannot be implemented in terms of any concrete type. Instead, we manipulate the elements through proxy_reference<C> objects. A proxy_reference<C> object maps the element-access operations of the runtime concept C onto the elements. While this is sufficient for manipulating the values through interfaces, this technique fails when argument type deduction is involved.

The original design of the STL tried to allow for arbitrary proxy types by including reference as one of the traits for an iterator but this was found to be insufficient in general. To see this problem, we look at the implementation of swap:

```
template <typename T>
void swap(T& x, T& y)
{
  T tmp(x);
  x = y;
  y = tmp;
}
```

If T is a proxy_reference, then this code will swap the two proxy_references, not the underlying values. What we would like is that the syntax T& would match *any type which is a reference to T* not generate a reference to the proxy type.

The best that we are able to easily achieve is a proxy_reference which behaves as a reference when the referenced value is not mutable. To test the ideas presented in this paper with the standard algorithms we used the following single shared reference scheme:

- proxies maintain a count of the number of proxies referring to a single value.
- when assigning through a proxy if the reference count is greater than one, then a copy of the value is made and all *other* proxies referring to the value are set to refer to the copy.

This relies on the fact that it would be inefficient to make a copy of a value and then assign over it. This is a very fragile and costly solution but it was sufficient to test the ideas in this paper. Solving the proxy dilemma properly in C++ is an open problem.

5. TESTS

To assess the performance cost of runtime concepts we tested the approaches described in sections §3 and §4. The numbers presented in this section were obtained on an Intel Pentium-D (2.8GHz clock speed; 512MB of main memory at 533 MHz) running CentOS Linux 2.6.9-42. We compiled with gcc 4.1.1 using −O3 and −march=prescott. Initially, the vector contained 8 million numbers in ascending order starting from zero. Then we invoke four algorithms: reverse, find of zero, sort, and lower_bound of zero.

vector<T>: As reference point for our performance tests we use the vector instantiated with a concrete type. The table shows the number of cycles each operation needs to complete for a container of **int** and **double** respectively. The column to the right of the number of cycles shows the slowdown factor compared to vector<**int**>. The algorithms with O(n) runtime complexity (i.e.: reverse and find) run approximately twice as long when used with type **double**. This discrepancy can be explained by the size of the stored datatype; **double** is twice as big as **int**.

	int		**double**	
reverse	50135428	1	101270708	2.0
find	24970288	1	52763884	2.1
sort	569752400	1	1170042300	2.1
lower_bound	5628	1	14784	2.6

vector<Scalar>: For this scenario, we defined a concept Scalar that extends the concept Copyable §3 with a function that allows conversions to type **double**. This conversion function is used to implement the concept LessThanComparable required by sort and lower_bound. The test uses an overloaded swap operation to efficiently swap the pointers (same size as **int**) of the polymorphic object-parts. Thus, the runtime of reverse is comparable to the optimal case; and beats the runtime for **double** by a factor of 2. The slowdown of find can be traced back to the equals-operator, which uses one virtual function call and one type-test per iteration. The sort algorithm requires two virtual function calls for each less-than comparison. Compared to the overhead of virtual function calls and type tests, the size of the element type is insignificant.

	int		**double**	
reverse	50260070	1.0	50112846	1.0
find	243300778	9.74	279807066	11.21
sort	10719950962	18.82	11098532844	19.48
lower_bound	20482	3.64	17626	3.13

Sequence<T>: The following table shows the results, when the algorithm library contains instantiations for concrete iterators. The time needed to select the best match is the only overhead that occurs.

	int		**double**	
reverse	50017254	1.0	101235652	2.0
find	24093454	1.0	51413502	2.1
sort	543942098	1.0	1163111236	2.0
lower_bound	17206	3.1	20286	3.6

Only when instances for the concrete iterators are missing, our system resorts to fallback implementations.

	int		**double**	
reverse	4736673970	94.5	4763215828	95.0
find	467545414	18.7	525890540	21.1
sort	16684763676	29.3	16980691560	29.8
lower_bound	19040	3.38	20552	3.7

The 94x slower performance for reverse is unacceptable, even for a fallback implementation. A closer analysis of the fallback test on reverse reveals that three factors contribute to its slowness: the fallback algorithm is bidirectional iterator based, virtual iterator functions, and model allocation on the heap.

To pinpoint and quantify the contribution of each of these factors we performed additional experiments: we started by adding a reverse function operating on randomaccess−iterator concepts, which improved the performance marginally to the factors 91.5x and 94.8x for int and doubles respectively. Each iteration of reverse has one call to std::iter_swap. The

gcc implementation of std::iter_swap calls another function that swaps the two elements to which the iterators point. Each function invocation creates copies of the iterators, which results in 16 million *unnecessary* heap allocations (and deallocations). By providing our own reverse implementation, we eliminated those copies. Then, reverse is only 7.4x slower for **int** (7.6x for **double**). The measured slowdown is less on other architectures (3x on a Pentium-M). Instead of rewriting the STL algorithm, we could adopt Adobe's small object optimization [14] where the wrapper classes reserve a buffer to embed small objects (Adobe's open source library [3]).

vector<RetroactiveScalar>: The following table shows the performance results for retroactively imposing runtime concepts on container elements (§4.3).

	int		double	
reverse	14580637680	290	14702289350	293
find	5525953258	221	5691982996	228
sort	145095555976	254	147783100836	259
lower_bound	30100	5.3	31010	5.5

Since the overhead of operating on Sequences has been determined by the previous test, we tested the performance of our proxyref-implementation by dispensing with runtime concepts for iterators. Instead, we use an iterator-class that has non-virtual access functions and wraps a vector::iterator. As described in §4.3 the iterator abstracts the concrete element type and operates on proxy_references. We used boost::shared_ptr to implement the single shared reference semantics. In the case of reverse, the poor performance is caused by the fact that each iteration requires the construction and destruction of five proxy_reference objects: two when the iterators are dereferenced, two when swap is invoked, and one when a temporary element inside swap is constructed (in total 20 million). Each of these operations performs an update of the shared_ptr. In addition, each temporary element is allocated on the heap (4 million).

6. RELATED WORK

The ASL [3] introduced the runtime concept idiom, employing type erasure [2] to provide the *any regular* library (similar to the boost any library), and its generalization, the *poly* library. The *poly* library generalizes the idiom to support refinement and polymorphic downcasting, encapsulates the common tasks required to create non-intrusive runtime-polymorphic value-based wrappers. The *poly* library design goals and implementation are elaborated in [14].

ASL also provides the *any iterator* library offering runtime-polymorphic iterators for specific types as a proof of concept. Becker [6] presents a similar library. In this paper we focus more on performance.

Bourdev and Järvi [8] discuss a mechanism for falling back to static-dispatch when type erasure is present.

Our work extends the above results to a library of algorithms operating on runtime-polymorphic containers of runtime-polymorphic elements, achieving realistic performance levels by using static dispatch where possible.

7. FUTURE WORK AND CONCLUSION

In the described system, an algorithm library handles the dispatch to the most appropriate algorithm present in the system. Comparable to function overload resolution, the template instantiation mechanism in C++0x considers the concept of all arguments to determine a unique best match. The current implementation of our algorithms, however, finds the best match based on the type of only one argument. Other arguments (i.e.: the second iterator in calls to reverse, sort, etc.) are obliged to conform to a type that the first argument determines. The presented algorithms are extensible with new iterators and sequences, unknown at the compile time, as long as they conform to the STL-defined concepts. Subsequent work is expected to support multiple dispatch and provide for modular runtime concept refinements. Iterator types for which repeated algorithm invocations frequently resolve to the fallback implementations would benefit from a runtime system that is capable of dynamic algorithm instantiation.

In this paper, we have discussed a runtime polymorphic version of several STL algorithms. Our implementation enables us to use STL's sequences where the binding to a concrete data structure is deferred until runtime. To improve runtime performance, if the data structures in use are known to the system integrator, our algorithms can leverage static dispatch. As a result, the runtime overhead becomes negligible for large data sets. We discussed retroactive runtime concept imposition on container elements and pointed out where better language support would be needed to further the runtime generic programming model.

8. ACKNOWLEDGMENTS

We thank Jaakko Järvi, Damian Dechev, Yuriy Solodkyy, Luke Wagner and the anonymous referees for their helpful suggestions.

9. REFERENCES

[1] The Boost C++ libraries, 2002. http://www.boost.org/.

[2] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series).* Addison-Wesley Professional, 2004.

[3] Adobe System Inc. Adobe Source Library. http://opensource.adobe.com, 2005.

[4] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: A Standard Template Adaptive Parallel C++ Library. In *LCPC '01*, pages 193–208, Cumberland Falls, Kentucky, Aug 2001.

[5] M. H. Austern. *Generic programming and the STL: using and extending the C++ Standard Template Library.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[6] T. Becker. Type erasure in C++: The glue between object oriented and generic programming. In K. Davis and J. Striegnitz, editors, *Multiparadigm Programming 2007: Proceedings of the MPOOL Workshop at ECOOP'07*, July 2007.

[7] L. Bettini, S. Capecchi, and B. Venneri. Double dispatch in C++. *Software - Practice and Experience*, 36(6):581 – 613, 2006.

[8] L. Bourdev and J. Järvi. Efficient run-time dispatching in generic programming with minimal

code bloat. In *Workshop of Library-Centric Software Design at OOPSLA'06, Portland Oregon*, Oct. 2006.

[9] J. C. Dehnert and A. A. Stepanov. Fundamentals of Generic Programming. In *Selected Papers from the International Seminar on Generic Programming*, pages 1–11, London, UK, 2000. Springer-Verlag.

[10] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. 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*, pages 291–310, New York, NY, USA, 2006. ACM Press.

[11] A. Igarashi and M. Viroli. On variance-based subtyping for parametric types. In *ECOOP '02: Proceedings of the 16th European Conf. on Object-Oriented Programming*, pages 441–469, London, UK, 2002. Springer-Verlag.

[12] ISO/IEC 14882 International Standard. *Programming languages: C++*. American National Standards Institute, September 1998.

[13] J. Järvi, M. A. Marcus, and J. N. Smith. Library composition and adaptation using C++ concepts. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 73–82, New York, NY, USA, 2007. ACM Press.

[14] M. Marcus, J. Järvi, and S. Parent. Runtime polymorphic generic programming—mixing objects and concepts in ConceptC++. In K. Davis and J. Striegnitz, editors, *Multiparadigm Programming 2007: Proceedings of the MPOOL Workshop at ECOOP'07*, July 2007.

[15] S. Parent. Beyond objects: Understanding the software we write. Presentation at C++ connections, November 2005.

[16] S. Parent. Concept-Based Runtime Polymorphism. Presentation at BoostCon, May 2007.

[17] T. L. Veldhuizen. Arrays in Blitz++. In *ISCOPE '98: Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments*, pages 223–230, London, UK, 1998. Springer-Verlag.