Adaptation of SVL and TSFCore for Interoperation

Anthony Padula

October, 2003
This work was done with the help of

Roscoe Bartlett (SNL)
Bart van Bloemen Waanders (SNL)
William W. Symes (Rice)
Problem OON packages express algorithms using common mathematical concepts. Implementation of these concepts differ in both semantics and syntax. This makes direct combination of OON packages impossible.

Solution Given sufficient semantical overlap, adapter classes can be written to cope with syntactical differences. May then combine packages.

Objective of Project Identify structural features of OON libraries which influence interoperability, considering both the programming efficiency and runtime efficiency of adaptation.

Illustrative Example To solve transient optimal control problems, combine

Moocho optimization library based on TSFCore (Sandia)

TSOpt time–stepping simulator based on SVL (Rice)
Outline

- Adapting low-level data containers.
- Issues in adapting high-level types.
- The Example
**Common Truth: Arrays**

Examples of classes which serve as the encapsulation of an array of contiguous data, but are all implemented in slightly different manners:

<table>
<thead>
<tr>
<th>Package</th>
<th>Array Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVL</td>
<td>LocalDataContainer</td>
</tr>
<tr>
<td>TSFCore</td>
<td>SubVector</td>
</tr>
<tr>
<td>TNT</td>
<td>Array1D</td>
</tr>
<tr>
<td>C++ STL</td>
<td>vector</td>
</tr>
<tr>
<td>OOQP</td>
<td>OOQPVector</td>
</tr>
</tbody>
</table>
Some methods of data access:

A. expose data pointers (e.g. `SVL::LocalDataContainer`, `TSFCore::SubVector`)

B. indexing operator `[]` (e.g. `stl::vector`, `TNT::Array1D`)

C. complete encapsulation, but list of ’standard’ methods (e.g. `OOQPVector`). A method for copying in/out is often provided.

Adaptation is possible between packages which use the same method, as well as down the list $A \rightarrow B$. Impossible to go up the list efficiently $B \rightarrow A$. 
Compatibility

SVL and TSFCore both use method A. They provide slightly different capabilities, but have enough semantic overlap to adapt efficiently.

- **TSFCore::SubVector y from SVL::LocalDataContainer x:**

  ```
  SubVector<Scalar> y;
  y.initialize(go, sd ? sd : x.getSize() - (fe-1),
               x.getData()+(fe-1), 1);
  ```

  Requires some pointer arithmetic, but no copying.

- **SVL::LocalDataContainer from a TSFCore::SubVector:** uses LocalSubVector adapter = subclass of LDC.
template<class Scalar>
LocalSubVector: public LocalDataContainer {
public:
/** return size of local data container */
virtual int getSize() { return s->subDim(); }
/** return address of data array */
virtual Scalar * getData() {
    return const_cast<Scalar *>(s->values());
}
/** virtual copy constructor */
SVL::DataContainer * clone() {
    return new LocalSubVector<Scalar>(*s);
}
};
Composing Adapters

Low-level containers are encapsulated at a higher level by DataContainer in SVL and Vector in TSFCore, examples of the Composite pattern.

Operations on data are implemented by SVL::FunctionObject and TSFCore::RTOp which are examples of the Visitor pattern. A visitor can pass through the high-level interface to gain access to the low-level containers.

Remaining Steps:

1. Adapt the visitors using the low-level data storage adapters
2. Adapt the composites using the visitor adapters
3. Combine tools written to the various interfaces to produce an application.
Figure 1: Class diagram for FORTOp
Adaptation Issues

Several critical differences between the visitors $\text{RTOp}$ and $\text{FunctionObject}$:

• forms of parameter lists
• reduction handling
• pervasiveness of functions related to parallelism
Different Parameter Lists

The footprint for the `RTOp::apply_op` method is

```c
void apply_op( const int num_vecs
, const RTOpPack::SubVectorT<Scalar> sub_vecs[]
, const int num_targ_vecs
, const RTOpPack::MutableSubVectorT<Scalar>
    targ_sub_vecs[]
, RTOp_ReductTarget reduct_obj ) const;
```

The footprint for a `BinaryFunctionObject::operator()` method is

```c
virtual void operator()
    (LocalDataContainer<Scalar> &,
     LocalDataContainer<Scalar> &);
```
Reductions

A reduction is an operation which takes one or more data containers as input and produces a result of an arbitrary type as output.

The default assumptions are that every RTOp is a reduction and every FunctionObject is not. Further the type RTOp::ReductTarget is really a void *, while SVL had no formal return type at all and simply used a templated RetType in the interface.

**Problem** With a templated RetType, impossible to dynamically cast a FunctionObject to a UnaryFunctionObjectRedn without knowing the return type apriori.

Thus, in the case of the FORTOp adapter, since the only type info is void *, *we’re stuck!*
Solution

Add an abstract base class to SVL for the return type ⇒ no need to template the reduction interfaces. Then adaptation is possible.

class RetType {
public:
    RetType() {}  
virtual RetType & operator=(const RetType & r) = 0;
virtual RetType * clone() const = 0;
virtual void reinitialize() = 0;
virtual void write( SVLException & e) = 0;
virtual ostream & write( ostream & str) = 0;
};
This suggests a new base `Reduction` class:

class Reduction {
protected:
    RetType & result;
public:
    Reduction( RetType & res) : result(res) {} 
    virtual void setResult() { result.reinitialize();}
    virtual void setResult(RetType & res) { result = res;}
    virtual RetType & getResult() { return result; }
    virtual RetType * createRetType() {
        RetType * temp = result.clone();
        temp->reinitialize();
        return temp;
    }
    virtual void accumulateResult(RetType & res1) = 0;
};
Parallel Pervasiveness

**RTOp** base class contains methods to admit parallelism through an MPI-compatible interface.

**SVL** intended to handle parallelism through subclassing and wrappers

**RTOp** example methods:

- `void get_op_type_num_entries( int* num_values, int* num_indexes, int* num_chars) const;`
- `void extract_op_state( int num_vals, Scalar val_data[], int num_indexes ,RTOp_index_type index_data[], int num_chars, RTOp_char_type char_data[]) const;`
• `void get_reduct_type_num_entries( int* num_values, int* num_indexes, int* num_chars) const;`

• `void reduce_reduct_objs( RTOp_ReductTarget in_obj, RTOp_ReductTarget inout_obj) const;`

These methods make adaptation difficult when coming from a package lacking such functionality in the base class. We must dynamically cast to a SVL subclass which offers sufficient functionality.
Workaround for Parallel Pervasiveness

Existing infrastructure for remote classes and Streamable objects.

Given Streamable FOs and RetTypes, make a SVLStream object which, instead of dumping data to the network, buffered the data so we could implement the needed functionality.

Thus, was created the StateExtractor. Pretends to be a SVLStream in order to

- sort data into a double, char, and int buffer as things are fed in.
- provide counts on the current number of items in its buffers.
- copy buffers into arrays
- do these in reverse, in order to load a state instead of extracting one.
Example Application

Combine adapters to build an application

1. Define transient system of differential equations \( c\left(\frac{dy}{dt}, y, u\right) = 0 \) using TSFCore.

2. Convert constraint \( c\left(\frac{dy}{dt}, y, u\right) = 0 \) to a least–squares function

\[
F(u, y_d) = f(y(u), u, y_d) = \|y_d - y(u)\|^2
\]

using TSOpt.

3. Solve the problem using Moocho.
Figure 2: Example application using several different packages
Figure 3: Class diagram for FORTOp
Figure 4: Sequence of calls to apply a UFO to a TSFCore::Vector
Figure 5: Class diagram for RTOpFO
Figure 6: Sequence of calls to apply a RTOp to a SVL::DataContainer