Proceedings of the RESOLVE Workshop 2002

Columbus, OH

June 17-19, 2002

Edited by Stephen H. Edwards

Virginia Tech TR #02-11
June 11, 2002

Individual papers in this collection are copyrighted by their original authors.

Department of Computer Science
Virginia Tech
660 McBryde Hall, MS 0106
Blacksburg, VA 24061

Preface ... iv

Research

Research Position Papers

A More Seamless Checking Wrapper for Raw C++ Pointers ... 1
Joseph E. Hollingsworth, Indiana University Southeast

RESOLVE Machine Paradigm Applied to Image Compression Algorithms ... 7
Joseph E. Hollingsworth and W. Christopher Lang, Indiana University Southeast

Mathematical Foundations for Reusable Software ... 9
Joan Krone, Denison University
William F. Ogden, The Ohio State University

Reaping More from Lazy Initialization Using Dynamic Reconfiguration ... 18
Nigamanth Sridhar, The Ohio State University

Good News and Bad News About Software Engineering Practice ... 22
Bruce W. Weide, The Ohio State University

Research Technical Papers

Making the Case for Assertion Checking Wrappers ... 28
Stephen H. Edwards, Virginia Tech

Specification and Verification of Performance Correctness ... 43
Joan Krone, Denison University
William F. Ogden, The Ohio State University
Murali Sitaraman, Clemson University

The Pragmatics of Integrative Programming ... 56
Larry Latour, Ling Huang, and Tom Wheeler, University of Maine

Design Issues Toward an Object Oriented RESOLVE ... 64
Roy Patrick Tan, Virginia Tech

Why Swapping? ... 72
Bruce W. Weide, Scott M. Pike, and Wayne D. Heym, The Ohio State University

Integration and Conceptual Modeling ... 79
Thomas J. Wheeler, University of Maine

Specifying and Verifying Collaborative Behavior in Component-Based Systems ... 95
Levent Yilmaz, Trident Systems Incorporated
Stephen H. Edwards, Virginia Tech

Education

Education Position Papers

Fostering Early Development of Expert-Like Knowledge Structures in Computer Science Students 105
Paolo Bucci and Timothy J. Long, The Ohio State University

Subsetting Language Elements in Novice Programming Environments 108
Peter J. DePasquale, Virginia Tech

Testing in Undergraduate Computer Science: Test Cases Need Formal Specifications 112
Allen Parrish, University of Alabama

Education Technical Papers

Capturing the Reference Behavior of Linked Data Structures 115
Gregory Kulczycki, Clemson University
William F. Ogden, The Ohio State University
Joseph E. Hollingsworth, Indiana University Southeast

Displacement Violation Checking and Ghost Facilities 133
Murali Sitaraman and Greg Kulczycki, Clemson University

Preface

The goal of the RESOLVE Workshop 2002 was to bring together educators and researchers interested in:

- Refining formal approaches to software engineering, especially component-based systems, and
- Introducing them into the classroom.

The workshop served as a forum for participants to present and discuss recent advances, trends, and concerns in these areas, as well as formulate a common understanding of emerging research issues and possible solution paths. The topics of interest solicited from participants included:

- Infusing software engineering techniques into computer science education:
    - Design with reuse
    - Design for reuse
    - Modular reasoning
    - Component-based software
    - Formal specification
    - Formal verification
    - Software testing
    - Using RESOLVE in the undergraduate curriculum
    - Pedagogical techniques to help teach the above topics
    - Using RESOLVE in the graduate curriculum
- Software engineering research on formal approaches to component-based systems:
    - Specification and verification of performance properties
    - Modular approaches to detecting component interface violations
    - Trade-offs among testing, formal verification, and model checking
    - Combining concurrency-oriented formalisms with model-based behavioral specification approaches
    - Formal characterization of user interfaces
    - Formal modeling of file system behavior
    - Formal characterization of mathematical and program types
    - Software engineering environments and tools
    - The Java Modeling Language (JML)
    - RESOLVE language and implementation issues

The RESOLVE Workshop 2002 was chaired by Stephen Edwards (Virginia Tech) and sponsored by the Department of Computer and Information Science at The Ohio State University. The remainder of the program committee consisted of Joseph Hollingsworth (Indiana University Southeast), Murali Sitaraman (Clemson University), and Bruce Weide (The Ohio State University).

A More Seamless Checking Wrapper for Raw C++ Pointers

Joseph E. Hollingsworth
Computer Science Department
Indiana University Southeast
4201 Grant Line Road
New Albany, IN 47150 USA

jholly@ius.edu
Phone: +1 812 941 2425
Fax: +1 812 941 2637
URL: http://homepages.ius.edu/jholly

Abstract

The "original" checked pointers component for C++ [Pike00] did a very good job of detecting many precondition violations for raw C++ pointers, including dereferencing a NULL or dangling pointer and deleting a dangling pointer. However, the syntax for invoking `new` and `delete` was slightly different than the standard syntax. These differences added extra confusion to student understanding of C++ pointers particularly when the student referenced commercial C++ books showing the standard syntax. This paper describes how we overloaded `new` and `delete` in order to make client usage of the checked pointers component conform to standard C++ syntax. Also described is how we added reference counts to detect storage leaks.

Keywords

checking components, checked pointers, esoteric C++ hacking

Paper Category: technical paper
Emphasis: research, education

1. Introduction

Our first year using the checked C++ pointers component with undergraduate students went very well except when the slightly different syntax for invoking `new` and `delete` confused the students. This was bothersome for two reasons: first we do not want to add any more confusion to the process of learning about pointers; and second, we attempt to use a version of RESOLVE/C++ that actually looks and feels like "real" C++. (We do this in hopes that students will be tempted to continue to use RESOLVE/C++ in other courses and maybe even at their place of work.) So we sat about trying to change the checking wrapper component (named "Pointer") so that client usage of `new` and `delete` would be exactly identical to what it would be if the Pointer component were not being used at all. Of course since Pointer is a template, the actual declaration of the pointer types will never be the same as if Pointer were not being used. But at least these differences are isolated to the declarative part of the program.

Checked pointers are implemented by two separate components: Pointer, a template which is directly used by the client programmer, and Pointer_Map, a class which underlies Pointer, and is the workhorse for detecting pointer violations (e.g., deletion of dangling pointers). Here are some of the changes made to the original Pointer template and the Pointer_Map class:

- implemented Pointer_Map so that it maintains reference counts for detecting storage leaks
- implemented Pointer_Map so that all operations (except Report_Allocation) are protected (better incapsulation)
- implemented Pointer_Map's Report_Allocation so that it reports the size of each allocated chunk of storage and that chunk's reference count; it also now reports a grand total number of bytes currently allocated
- overloaded Pointer's
    - `operator new`
    - `operator delete`
    - copy constructor for passing Pointer objects by value
    - `operator =` so that it can take a Pointer* and Pointer&
    - `operator <<`
- implemented Pointer so that it inherits from the Pointer_Map class

- implemented Pointer's destructor so that it detects storage leaks
- implemented Pointer's `operator =` so that it detects storage leaks

Ultimately, client programs that use Pointer are computing with Pointer objects everywhere and through esoteric (tricky) use of overloading Pointer's operations (e.g., `new`, `delete`, `operator ->`, etc.) it looks like the client program is computing with real raw C++ pointers, but it is not. The really slick thing about it is, after a quick change of the pointer declarations, i.e., eliminating the use of Pointer in only the declarative part of the program, the client program really does compute with raw C++ pointers. Furthermore, none of Pointer relies on any other RESOLVE/C++ components. Therefore, this checked pointers component can be used by anybody writing C++ code who also wants to adhere to standard C++ syntax.

2. The Problem

During development of new client software, to reduce development effort and defects in the production version, the RESOLVE discipline recommends that the client programmer utilize checked versions of components used by the new client [Hollingsworth00]. If you take the view that raw pointers in C++ are a service providing component, then it follows that there should be a checked version of raw C++ pointers, and that this checked version should be used during the development phase of any client that relies on raw C++ pointers.

A requirement of a checking component is that it must have the exact same syntactic interface as its corresponding unchecked component [Edwards98], so that the unchecked version can be substituted for the checked version without any change to the system's executable part (change to the system's declarative part is permitted). The original checked Pointer template introduced in [Pike00] met many of the requirements for being a checked version of raw C++ pointers with a couple of exceptions: calls to the checked `new` and `delete` were not identical to those used with raw C++ pointers and checking for the creation of "garbage" (i.e., storage leaks) was not implemented. These two exceptions proved to be bothersome when using the checked Pointer template with students in an undergraduate data structures class.

Undergraduate students learning to use pointer types in C++ were confused by the differences they saw in calls made to Pointer's `new` and `delete` as compared to standard C++ syntax (and no amount of explaining the difference seemed to help). Furthermore, their programs often created garbage which the Pointer template did not detect.

The problem became, how to create a version of the checked Pointer template that would detect the creation of garbage at run time and whose use by a client program would syntactically conform to that of standard raw C++ pointers.

3. Working the Problem

The running example used throughout this section is based on allocating and deallocating a "node" of a linked list. This particular node holds an integer as its data value and an address to the next node in the linked list.

The variable "head" holds the address to the first node in the linked list.

The code to the right illustrates the standard syntax for allocating and deallocating a node using `new` and `delete`. This is the look and feel that we are after when using the Pointer checking wrapper.

```
typedef Node_Struct* Node;

struct Node_Struct {
   int value;
   Node next;
};

Node head;

head = new Node_Struct;
...
delete head;
```

3.1 Storage Allocation

Part of the declaration of the Pointer template and declarations used by a client are shown to the right. The first step taken in order to make Pointer's `new` and `delete` to have normal C++ pointer look and feel is to overload Pointer's `new` and `delete` operators.

A close examination of `new`'s return type shows that it is void*. But the variable head does not hold an address to a Node. In fact when using the Pointer wrapper, head is not a plain variable anymore, head is an object that holds a `Pointer <Node_Struct>.` This poses a problem right away. The client of Pointer will be computing with object's of type `Pointer <Node_Struct>`, not `Pointer <Node_Struct>*` which is what `new` returns.

As far as we can tell, it is not possible to change `new`'s return type.

```
template <class T>
class Pointer
{
   ...
   void* operator new (size_t s);
   void operator delete (void *p,
size_t s);
   ...
private:
   T* rep;
};

//-----------------------------
// Client declaration using Pointer
```

Which leads to the next step of overloading `operator =`. Working under the assumption that when new is invoked it will be on the rhs of an assignment statement, we can overload `operator =` so that it takes `Pointer <Node_Struct>*`. Below are the headers for the overloaded `operator =`:

```
Pointer& operator = (Pointer* rhs);
Pointer& operator = (Pointer& rhs);
```

```
struct Node_Struct;
typedef Pointer <Node_Struct>
Node;
struct Node_Struct {
   int value;
   Node next;
};

Node head;
```

What's Really Going On

Example:

```
head = new Pointer <Node_Struct>;
// in other words: head.operator = (new Pointer <Node_Struct>);
```

The implementation of `new` ignores the size information it has been passed in parameter size_t. `new` is not allocating storage for a `Pointer <Node_Struct>`, it is allocating storage for type T, i.e., the template parameter (in the running example type T is Node_Struct). Even though `new` is a member function of Pointer, it acts like a static member function in that it has no access to data members of objects declared from Pointer. Therefore the only thing `new` can really do is return the address of the allocated storage. Before returning the address, `new` calls Pointer_Map's Add operation to add the address to Pointer_Map's memory of allocated storage. So `new` is really returning a pointer of type T*, which then gets handed off to Pointer's `operator =` (again assuming `new` appears on the rhs of assignment). The implementation of `operator =` takes the address in rhs and stores it in head.rep; head.rep is declared at T* (see private part of Pointer's declaration above). Before the assignment of rhs to head.rep is made, testing is done to detect the creation of garbage. `operator =` has Pointer_Map decrement the reference count of the address in head.rep, then checks to see if its count has gone to zero. If its count has gone to zero, then the storage addressed by head.rep is about to become garbage and `operator =` signals a run-time error. If its count is greater than zero, then that means the storage is aliased and is not about to become garbage. In that case `operator =` has Pointer_Map increment the reference count of the storage addressed by rhs and then does the assignment: `rep = (T*)rhs;`

3.2 Storage Deallocation

What's Really Going On

Example:

```
delete head;
```

Refer back to the declaration of `delete` in Pointer (above). You will notice that `delete` expects to receive a void*. But look at the example above, `delete` is getting head, and head is not a pointer variable, it is an object of type `Pointer <Node_Struct>`. This appeared to be a show stopper, and it was until we added the following type conversion operation to the Pointer template:

```
operator Pointer* ();
```

When "**`delete`**` head;`" gets compiled, the compiler realizes that head is not the right type (its an object, not a pointer variable), so it examins the Pointer template and discovers the type conversion operation and uses it. So what really happens is (1) **`operator`**` Pointer* ()` gets invoked on head (which just returns `this` for head), and then (2) `delete` gets the `this` pointer for head and uses it to get to head.rep. At that point `delete` checks to see if Pointer_Map has head.rep in its memory, if not, head.rep is a dangling reference and `delete` signals a run-time error. If it is, it has Pointer_Map remove it from its memory, and then deallocates the storage.

We considered having `delete` decrement the reference count for the storage addressed by head.rep, and if it is still greater than zero, signal that dangling references are about to be created. But then we remembered that dangling references are not a problem until they are dereferenced somewhere downstream in the computation. It they are never dereferenced, then there is no problem. If they are, then other parts of Pointer will detect the illegal dereferening of dangling pointers.

3.3 The Rest of the Story for Detecting Storage Leaks

Implementing the Destructor

If the object head goes out of scope and the storage addressed by head.rep has a reference count of one, then garbage is about

to be created. To detect this situation, ~Pointer () is implemented so that it asks Pointer_Map if head.rep is still in its memory (i.e., it was not removed by `delete`), then it decrements its reference count and then checks to see if it is zero. If it is zero, then that means the storage is not aliased and the destructor signals a run-time error that there is a storage leak.

Implementing the Copy Constructor

But a destructor also gets invoked when value parameters go out of scope, therefore a copy constructor has to be implemented for template Pointer. When a client program passes a pointer variable it might contain an address to an allocates piece of storage, a dangling address or NULL. The copy constructor must first check to see if the address is in Pointer_Map's memory, if so, must have Pointer_Map increment its reference count. When the actual value parameter goes out of scope then the destructor will behave properly and not signal any errors.

Implementing `Pointer& operator = (Pointer& rhs);`

This operator gets called whenever one pointer is assigned to another. It must use the reference count of the object on the lhs to detect when garbage is created and it also must increment the reference count on the rhs.

3.4 The Client's Declarative Part

In order to painlessly switch from using the Pointer template to using real raw C++ pointers (and back), some care needs to be taken when declaring the data types.

To the right is an example using conditional compilation based on the symbol _DEBUG being defined. If _DEBUG is defined, then checked pointers are used, if not then regular raw C++ pointers are used.

Below is a snippet of code using the typedef'ed symbols and what it looks like after being transformed by the preprocessor.

```
struct Node_Struct;

#ifdef _DEBUG
typedef Pointer <Node_Struct> Node;
typedef Node Node_Rep;
#else
typedef Node_Struct* Node;
typedef Node_Struct Node_Rep;
#endif

struct Node_Struct {
   int value;
   Node next;
};
```

Code as it appears in the program.

```
   Node head;

   head = new Node_Rep;
   ...
   head->value = 17;
   ...
   delete head;
```

Code as it appears after preprocessing.

```
// Debug mode, _DEBUG is defined
   Pointer <Node_Struct> head;

   head = new Pointer <Node_Struct>;
   ...
   head->value = 17;
   ...
   delete head;
```

```
// Release mode, _DEBUG is not defined
   Node_Struct* head;

   head = new Node_Struct;
   ...
   head->value = 17;
   ...
   delete head;
```

3.5 How `new` Can Record the Line Number and Filename

To facilitate debugging, the original Pointer wrapper described in [Pike00] also recorded the line number and filename where storage was allocated. When the software developer asked for a report of memory usage, this additional information was displayed with each allocated piece of storage. When moving to the new Pointer template with `new` and `delete` overloaded, it appeared that we were going to have to forfeit this capability. But as is often the case with C++, there is usually some workaround. As it turns out when overload operator `new`, additional parameters can be added to the syntactic interface, the C++ reference manual refers to this as *placement syntax*. Below is the real header for new used in Pointer. It uses default values for the line number and filename for the cases where the user does not care to keep track of this additional information.

```
   void* operator new (size_t s, int line_num = __LINE__, char* file = __FILE__);
```

Of course this is not standard syntax for invoking `new`, so once more we have to rely on the preprocessor to help us out as is illustrated below. When in debug mode, `new` is redefined using the preprocessor to include the additional parameters.

```
#ifdef _DEBUG
#define new new (__LINE__, __FILE__)
#endif

   Node head;

   head = new Node_Rep;
   ...
   head->value = 17;
   ...
   delete head;

#undef new
```

3.6 Signaling Run-Time Errors by Invoking the Debugger

The Pointer checking component can be implemented a number of different ways for signaling the software developer when it detects a run-time error. Currently our department uses Microsoft's Visual C++ environment which comes with an interactive debugger. We have chosen to implement Pointer so that it invokes the debugger when a run-time error is encountered. Control is passed to the debugger and it can be used by the student (with minimal training) to find the exact line in the code where the run-time error was detected by Pointer. All errors are reported through an operation provided by Pointer_Map, it is shown below. The operations OutputDebugString and DebugBreak come from the Microsoft environment. OutputDebugString permits sending output to the debugger's output window, and DebugBreak transfers control to the debugger.

```
void __Pointer_Map::Report_Error (char* msg)
{
   OutputDebugString (msg);
   OutputDebugString ("\n");
   DebugBreak ();
} // Report_Error
```

Pointer_Map also provides Report_Allocation, an operation which displays the current memory allocation. Below is a sample displaying the memory allocation for a One_Way_List implemented using a Bookkeeping node at the head of the list and nodes similar to the Node used above. In this example, there are 5 lists with one list containing 9 items. The 5 bookkeeping nodes each have size 16 and were allocated by a call to new on line 203, and the 9 regular nodes have size 12 allocated by a call to new on line 20. The reference count for each node is 1 as is shown by: (rc:1).

```
Current Memory Allocation
=====================================================
Pointer report: Currently allocated memory locations:
-----------------------------------------------------
188 bytes currently allocated
Addr: 0x002F0988 (z:16) (rc:1) (Allocated in: owlist2.hc Line: 20)
Addr: 0x002F4038 (z:12) (rc:1) (Allocated in: owlist2.hc Line: 203)
Addr: 0x002F3908 (z:12) (rc:1) (Allocated in: owlist2.hc Line: 203)
Addr: 0x002F0A50 (z:16) (rc:1) (Allocated in: owlist2.hc Line: 20)
Addr: 0x002F41A8 (z:12) (rc:1) (Allocated in: owlist2.hc Line: 203)
Addr: 0x002F0B18 (z:16) (rc:1) (Allocated in: owlist2.hc Line: 20)
Addr: 0x002F3A78 (z:12) (rc:1) (Allocated in: owlist2.hc Line: 203)
Addr: 0x002F4318 (z:12) (rc:1) (Allocated in: owlist2.hc Line: 203)
Addr: 0x002F0CA8 (z:16) (rc:1) (Allocated in: owlist2.hc Line: 20)
Addr: 0x002F3BE8 (z:12) (rc:1) (Allocated in: owlist2.hc Line: 203)
Addr: 0x002F4488 (z:12) (rc:1) (Allocated in: owlist2.hc Line: 203)
Addr: 0x002F3D58 (z:12) (rc:1) (Allocated in: owlist2.hc Line: 203)
Addr: 0x002F3EC8 (z:12) (rc:1) (Allocated in: owlist2.hc Line: 203)
Addr: 0x002F2808 (z:16) (rc:1) (Allocated in: owlist2.hc Line: 20)
=====================================================
```

3.7 What's Not Handled

Currently Pointer is not interchangeable for all possible uses of pointer types in C++. Here is a list of things that Pointer

cannot currently handle.

- `n = x098233;`
  This is the assignment of a fixed memory address to a pointer type. Problems with Pointer start to arise because it does not know that this address references storage that was never allocated by `new` in the first place.
- **`delete new`** `Datatype;`
  This just allocates a piece of storage and hands it right back to `delete` for deallocation. The timestamp for the allocated piece of storage does not get set correctly before `delete` gets a hold of the address.
- pointer arithmetic
  Since operator ++ and operator -- can be overloaded, the Pointer template might be able to handle pointer arithmetic. We would be inclined to avoid these topics in undergraduate courses, however, if we would like to see Pointer get adopted by practicing software developers, it might be worth working on this problem.
- using pointers as arrays
  Since operator [] can be overloaded, the Pointer template might be able to handle array access.
- linked structures containing cycles
  We have made no attempt to make Pointer try to detect when garbage is being created in the presence of cycles in the linked structures.

4. Related Work

As noted elsewhere in this paper the "original" checked pointers component for C++ developed by [Pike00] did a very good job of detecting many precondition violations for raw C++ pointers, including dereferencing a NULL or dangling pointer and deleting a dangling pointer. We were driven to make changes to their original Pointer component because we wanted client usage to conform to regular C++ syntax. Along the way Bruce Weide suggested that we make a few other changes, e.g., adding detection of garbage creation through reference counting; thanks Bruce!

5. Conclusion

The introduction claims that this new Pointer template can be used by anybody writing C++ code who also wants to adhere to standard C++ syntax. Section 3.7 identifies at least two areas where that is not currently true, programs using pointer arithmetic and pointers as arrays. We urge others in the RESOLVE community who are still using C++ to adopt this newer version of the Pointer template and find ways to enhance its functionality. Finally, we believe that the new Pointer template should be publicized in forums that practicing C++ hackers read in an attempt to get widespread adoption.

References

[Edwards98]
Edwards, S.H., Hollingsworth, J.H., Shakir, G., Sitaraman, M., Weide, B.W., “A Framework for Detecting Interface Violations in Component-Based Software” in Proceedings of the Fifth International Conference on Software Reuse, Victoria, B.C., Canada, June, 1998

[Pike00]
Pike, S., Weide, B., Hollingsworth, J., “Checkmate: Cornering C++ Dynamic Memory Errors With Checked Pointers”, in Proceedings of Special Interest Group on Computer Science Education (SIGCSE) 2000, Austin, TX, March, 2000.

[Hollingsworth00]
J. Hollingsworth, L. Blankenship and B. Weide, "Experience Report: Using RESOLVE/C++ for Commercial Software", in ACM’s Software Engineering Notes, Vol. 25, No. 6, pps. 11 – 19, November 2000.

RESOLVE Machine Paradigm Applied to Image Compression Algorithms

Joseph E. Hollingsworth
Computer Science Department
Indiana University Southeast
4201 Grant Line Road
New Albany, IN 47150 USA

jholly@ius.edu
Phone: +1 812 941 2425
Fax: +1 812 941 2637
URL: http://homepages.ius.edu/jholly

W. Christopher Lang
Mathematics Department
Indiana University Southeast
4201 Grant Line Road
New Albany, IN 47150 USA

wclang@ius.edu
Phone: +1 812 941 2391
Fax: +1 812 941 2637
URL: http://homepages.ius.edu/wclang

Abstract

Image compression algorithms such as JPEG are widely used when delivering internet content. The algorithms typically have distinct phases and tradeoffs of quality versus performance. It seemed to us that attempting to recast these algorithms as a machine would provide for an interesting challenge for the RESOLVE machine paradigm.

Keywords

machine paradigm, image compression algorithms

Paper Category: position paper
Emphasis: research

1. Introduction

JPEG2000, the latest JPEG standard for image compression described in [Taubman02] contains many phases including: image pre-processing, image partitioning, applying a mathematical transformation to each partition, scaling the transformed data, and coding the scaled data. These image compression algorithms appear to be good candidates for recasting as machines as described by the RESOLVE machine paradigm. The authors of [Bucci02] outline several desirable properties of software machines including: hide data representations, hide algorithms used, hide when specific actions are executed, etc. We seek to build an image compression machine possessing these properties.

2. The Position

It is our position that we can attack this problem at several different levels including:

- developing a RESOLVE specification of a mathematical model for precisely describing the image data and the transformations that are applied to it;
- developing an interface that hides the data structures and algorithms used to implement an image compression algorithm;
- developing smaller machines that represent some of the individual steps, e.g., the wavelet or Fourier transform, or the block coder; and
- developing an implementation for the interface.

3. Related Work

We base our work on [Bucci02] and [Weide94], in conjunction with our own experience at implementing sorting machine in various languages and with various algorithms.

4. Conclusion

Image compression algorithms (e.g., JPEG) comprise many steps and possess opportunities for making tradeoffs in image quality versus performance. These properties of image compression algorithms appear similar in nature to the properties possessed by sorting algorithms which have been successfully recast, and effectively captured as a sorting machine. We understand that image compression algorithms may be quite difficult to recast, we also know that if we persist, we will learn a lot along the way.

At the workshop we would like to discuss with other participants the work that they might have done with image compression algorithms. We would also gain from experiences related to recasting other algorithms as machines.

References

[Bucci02]
Bucci, P., Heym, W., Long, T.J., Weide, B.W., "Algorithms and Object-Oriented Programming: Bridging the Gap" in 33rd SIGCSE: Technical Symposium on Computer Science Education, Covington, KY, February, 2002.

[Taubman02]
Taubman, Marcellin, *JPEG2000: Image Compression Fundamentals, Standards and Practice*, Kluwer, 2002.

[Weide94]
Weide, B.W., Ogden, W., and Sitaraman, M., "Recasting algorithms to Encourage Reuse", *IEEE Computer Society*, 11(5), September 1994, pp. 80-88.

Mathematical Foundations for Reusable Software

Joan Krone
Dept. of Mathematics and Computer Science
Denison University
Granville, OH 43023 USA
krone@denison.edu
Phone: +1 740 587 6484
Fax +1 740 587 5749
http://www.denison.edu/~krone

William F. Ogden
CIS Department
The Ohio State University
Columbus, OH 43210 USA
ogden@cis.ohio-state.edu
Phone +1 614 292 6004

Abstract

Assuming that software should be reused only when it has been certified as correct and assuming that a piece of generic software is an excellent candidate for reuse, we address the problem of writing and proving assertions that are expressed over any possible type that might be used in a given generic component. We note that semantics for such programs are relational, requiring us to do fixed point theory for relations on very large collections of items from various domains. Current mathematical set theory is not adequate for doing the kind of fixed point theory we need. Here we introduce some mathematics in the form of *math units* to be part of RESOLVE and show how those units support fixed point theory over general relations.

Keywords

Specification, verification, complete partial orders, set theory, fixed points.

Paper Category:
Emphasis: research

Introduction

Software engineers agree that building every new system from scratch is unrealistic, so they seek out existing software that appears to fit into the design of their new project, thereby reducing the number of new components to write. However, it often turns out that the existing components may not quite fit or may have previously undetected errors that show up only in this new application. These two problems, that of knowing exactly what a piece of software does and being sure that it is correct, can be solved only when software components are formally specified and verified [3, 4, 5].

Formal specification and verification require the use of some formal language for specifying what the software does and a proof system that can establish the correctness of it. We take an approach that uses mathematical assertions as part of the language for writing software, i.e., an assertive language which includes both executable constructs and specification assertions.

To complete this system for creating software, it is necessary to specify not only executable code in the software, but also the mathematical units used in proving correctness. Here we introduce syntax for specifying these units as components to be included in a given software project and give an example illustrating the language as it applies to a particular mathematical system as proof of concept. Ultimately, it will be necessary to build a supporting compiler that not only translates the executable code, but also includes a verifier that processes these math units and proves correctness of each component.

The Problem

One of the most important tools for creating reusable software is the capability to design and implement generic components. Instead of finding it necessary to write separate container classes depending on what the contents may be, we can write one class indicating that the contents may be of any type the user chooses, including types defined by the user. However, generic components are the most daunting aspects of reusable software from the point of view of proving correctness.

Certifying generic components as correct requires us to quantify over collections of mathematical objects so large that they are not sets. For example, suppose we have a generic stack component allowing us to place entries of any existing type or any type to be defined in the future as contents of the stack. Then it is necessary for us to make assertions of the form "for any entry type," followed by whatever statement we need to prove. Since users may introduce new types at will, our specification language and our proof system must allow us to express assertions involving any type that might be defined. It is also our intention to automate our proof system. Of course, this means that we need syntax that is reasonable for a program to process and we need semantics to support the system. We contend that traditional mathematical set theory does not support these needs.

Another aspect of the problem involved in formally specifying and verifying generic software is that traditional semantics, defined in terms of functions, are inadequate for RESOLVE, indeed for any assertive language that permits generic components. As shown in [2], we need relational semantics that must explain generic components written over general types.

Steps Toward Solution

We have described some of the challenges of formally dealing with generic software written for reuse. Here we address two aspects of those challenges. First, we present an example of a math unit as proof of concept that mathematics can be formalized in such a way as to make automated processing a possibility, i.e., we exhibit a syntax for expressing mathematical theories. We choose to show a math unit for ordering theory since it is necessary to use in describing the math unit in addressing semantic issues.

Secondly, we address part of the relational semantics challenge by introducing a mathematical theory to do fixed points over relations on general types (sets). This is done with CPO theory (complete partial ordering), a necessary basis for proving functional correctness of **while** loops.

Basic Ordering Theory

We present ordering theory as fundamental to a variety of areas in mathematics. In particular it is used for the development of well orderings, necessary for explaining complete partial orders we use in writing semantics.

In this short math unit, the only keywords used are **Math_Unit**, **uses** (to designate the list of other math theories needed in the one being defined. In the case of Basic_Ordering_Theory, Basic_Binary_Relation_Properties is listed as a math unit needed. That unit is also included here so that it is easy to see where some of the language used in the ordering theory came from. The keyword **Def**., as the abbreviation suggests, indicates a definition. As we will see in the next unit, other keywords are available including **Theorem**, **corollary**, and **Inductive_Def**., each taking the obvious meaning. When proofs for theorems are included, additional keywords such as **Assumption** and **Goal** appear. A math unit that includes theorems without proofs is called a **Precis**. It serve as an interface for those who want to use it for writing a new math unit, since it shows only what is needed by the user. The proofs are included in the math unit by the same name as the **Precis**.

This math unit uses another unit in which basic binary relation properties are introduced. Those properties include reflexivity, transitivity, symmetry, anti-symmetry, and others. The ordering unit can be written concisely with those properties available:

```
Math_Unit Basic_Ordering_Theory;
                    uses Basic_Binary_Relation_Properties;

    Def. Is_Preordering( ( □: D: Set ) ⊴ ( □: D ): 𝔹 ): 𝔹 = ( Is_Reflexive( ⊴ ) and
                                                                    Is_Transitive( ⊴ ) );

    Def. Is_Partial_Ordering( ( □: D: Set ) ⊴ ( □: D ): 𝔹 ): 𝔹 = ( Is_Preordering( ⊴ ) and
                                                                    Is_Antisymmetric( ⊴ ) );

    Def. Is_Total_Preordering( ( □: D: Set ) ⊴ ( □: D ): 𝔹 ): 𝔹 = ( Is_Transitive( ⊴ ) and
                                                                    Is_Total( ⊴ ) );

    Def. Is_Total_Ordering( ( □: D: Set ) ⊴ ( □: D ): 𝔹 ): 𝔹 = ( Is_Total_Preordering( ⊴ ) and
                                                                    Is_Antisymmetric( ⊴ ) );

    Def. Is_Strict_Partial_Ordering( ( □: D: Set ) ⊲ ( □: D ): 𝔹 ): 𝔹 = ( Is_Transitive( ⊲ ) and
                                                                    Is_Asymmetric( ⊲ ) );

    Def. Is_Strict_Ordering( ( □: D: Set ) ⊲ ( □: D ): 𝔹 ): 𝔹 = ( Is_Strict_Partial_Ordering( ⊲ )
```

Def. Is_Symmetric((□: D: *Set*) ρ (□: D): 𝔹): 𝔹 = (∀ x, y: D, **if** x ρ y, **then** y ρ x);

Def. Is_Antisymmetric((□: D: *Set*) ρ (□: D): 𝔹): 𝔹 = (∀ x, y: D,
if x ρ y **and** y ρ x, **then** x = y);

Def. Is_Asymmetric((□: D: *Set*) ρ (□: D): 𝔹): 𝔹 = (∀ x, y: D, **if** x ρ y, **then** ¬ y ρ x);

Def. Is_Total((□: D: *Set*) ρ (□: D): 𝔹): 𝔹 = (∀ x, y: D, x ρ y **or** y ρ x);

Theorem Rln1: ∀ D: *Set*, ∀ ρ: D×D→𝔹, **if** Is_Total(ρ) **then** Is_Reflexive(ρ));

Def. Is_Trichotomous((□: D: *Set*) ρ (□: D): 𝔹): 𝔹 = (∀ x, y: D, x ρ y **or** x = y **or** y ρ x);

end Basic_Binary_Relation_Properties;

Complete Partial Orderings

We turn now to the most challenging problem mentioned earlier, that of introducing the mathematical foundations for doing denotational semantics for RESOLVE or any other assertive language that supports generics. Traditional approaches to denotational semantics assume that each construct within a given program segment computes a function and that the meaning of the entire program is the function found by performing functional composition on the functions associated with each segment. However, as was pointed out in [2], our programs require relational semantics in many cases.

Hence, in order to give meaning to our programs we must associate a relational meaning with each construct, and explain how relational composition works. For example, when writing denotational semantics for loops or recursive procedures, it will be necessary to consider chains of relations, rather than chains of functions. We then need to find least upper bounds on those chains, which means that we need to show their existence.

Making things even more complicated is the fact that our programs are written over general types, not just integers. This requires that our supporting set theory give us the expressiveness to quantify over such things as all stacks of any entry type, something that traditional set theory does not do.

In fact, establishing a theory of sets presents something of a paradox. On the one hand, the description of the nature of sets must be simple enough that we can believe that it really identifies what we want. On the other hand, that same description should imply the existencd of any arbitrarily strange set that we might ever conceive of. This isn't quite attainable, so we'll have to settle for getting at least the sets that people have needed so far, with the possibility of augmenting the description of the nature of sets if we discover that something we need is missing.

Another conundrum arises from the problem of what domain to use to describe set theory. If the foundational thesis is correct, then set theory, being a mathematical theory, should be described using set theory! Having the collection of all the sets you'd ever want to talk about be a member of itself, turns out to be just as bad an idea as you'd expect it to be. The way out of this problem is to make a distinction between the formal set theory that we're trying to describe and an informal (bigger) set theory in our metalanguage.

We begin formulating our foundational theory of sets in terms of just two concepts. The first is the collection *Set*, which we intend to denote all possible sets, from which all the objects to be discussed will come. The second is the two-argument predicate x ∈ y, which we intend to denote that the set x is a member of the set y. For our purposes we just need to know that this set theory exists and is consistent. For details, see [1].

In our basic set theory development in [1], the only object type under discussion is sets, so we don't feel the need for a type system. As we develop the various specialized domains we need for ordinary mathematics within this set theory framework, having a type system proves to be a great convenience. This is because, first, types are familiar and natural from informal usage, second, they shorten our mathematical expressions, and third, they provide an easy mechanical check that frequently highlights areas in our mathematical developments where our thinking has gotten jumbled.

In our approach we add a type specification scheme to our basic predicate calculus system and we insist that all variables in our mathematical language be explicitly introduced by a construct such as a quantifier or a definition, which will be used to assign them a type. The types will just be sets from our underlying set collection and in fact, the types will be the general mechanism by which the underlying sets will be introduced into our mathematical language. Note that we adhere to this approach to typing in the ordering theory unit previously shown, as we do in all math units.

Assuming the set theory we need is available, we define complete partial order theory over *Set*, first providing definitions and theorems that describe CPO's in general. We then consider the kinds of relations that we need for our program semantics and introduce definitions and theorems that deal with those relations.

Precis Complete_Partial_Order_Theory;
uses Well_Ordering_Theory;

Def. Is_CPO(⊑: (D: *Set*)⊠D→𝔹, ⊥: D): 𝔹 = (Is_Partial_Ordering(⊑) **and** ∀ x: D, ⊥ ⊑ x
and ∀ C:℘(D), **if** ∀ x, y: C, x ⊑ y **or** y ⊑ x, **then** ∃ b: D ∋ ∀ x ∈ C, x ⊑ b **and**
∀ c: D, **if** ∀ x ∈ C, x ⊑ c, **then** b ⊑ c);

Theorem CPO1: ∀ ⊴: (D: *Set*)⊠D→𝔹, **if** Is_Well_Ordering(⊴)
then Is_CPO(⊴, GLBwrt(⊴, D));

Def. Chain(⊑: (D: *Set*)⊠D→𝔹): ℘(℘(D)) = { C:℘(D) | ∀ x, y: C, x ⊑ y **or** y ⊑ x };

Corollary ∀ D: *Set*, ∀ ⊑: D⊠D→𝔹, ∀ ⊥: D, ∀ C: Chain(⊑), **if** Is_CPO(⊑, ⊥)
then ∃! b: D ∋ ∀ x ∈ C, x ⊑ b **and** ∀ c: D, **if** ∀ x ∈ C, x ⊑ c, **then** b ⊑ c;

Implicit Def. LUBwrt(⊑: (D: *Set*)⊠D→𝔹, ⊥: D, C: Chain(⊑)): D **is**
if Is_CPO(⊑, ⊥) **then** ∀ x: C, x ⊑ LUBwrt(⊑, ⊥, C) **and**
∀ c: D, **if** ∀ x: C, x ⊑ c **then** LUBwrt(⊑, ⊥, C) ⊑ c **and**
if ¬ Is_CPO(⊑, ⊥) **then** LUBwrt(⊑, ⊥, C) = ⊥;

Theorem CPO2: ∀ D: *Set*, ∀ ⊑: D⊠D→𝔹, ∀ ⊥: D, ∀ C: Chain(⊑), ∀ B:℘(D),
if B ⊆ C **then** B ∈ Chain(⊑) **and** LUBwrt(⊑, ⊥, B) ⊑ LUBwrt(⊑, ⊥, C);

Theorem CPO3: ∀ ⊑: (D: *Set*)⊠D→𝔹, ∀ C: Chain(⊑), ∀ x ∈ C, **if** x ≠ LUB(C)
then ∃ y ∈ C ∋ x ≠ y **and** x ⊑ y.

Theorem CPO4: ∀ C: Chain(⊑), **if** C ≠ ∅ **and** LUB(C) ∉ C, **then** ‖C‖ ≥ $\aleph_0$.

Def. Is_Monotonic_for(⊑: (D: *Set*)⊠D→𝔹, ≼: (R: *Set*)⊠R→𝔹, f: D→R): 𝔹 = (
∀ x, y: D, if x ⊑ y, then f(x) ≼ f(y)).

Theorem CPO5: ∀ D, R: *Set*, ∀ ⊑: D⊠D→𝔹, ∀ ≼: R⊠R→𝔹, ∀ f: D→R, ∀ C: Chain(⊑),
if Is_Monotonic_for(⊑, ≼, f) **then** f[C] ∈ Chain(≼).

Def. Is_MFP_for($\sqsubseteq$: (D: $\mathcal{S}et$)$\boxtimes$D$\to\mathbb{B}$, f: D$\to$D, p: D): $\mathbb{B}$ = (f(p) = p **and** $\forall$ q: D,

if f(q) = q, **then** p $\sqsubseteq$ q).

Implicit def. FPAwrt($\sqsubseteq$: (D: $\mathcal{S}et$)$\boxtimes$D$\to\mathbb{B}$, $\bot$: D, $\trianglelefteq$: (W: $\mathcal{S}et$)$\boxtimes$W$\to\mathbb{B}$, f: D$\to$D, w: W): D **is**

if Is_CPO($\sqsubseteq$, $\bot$) **and** Is_Well_Ordering($\trianglelefteq$) **then**

FPAwrt($\sqsubseteq$, $\bot$, $\trianglelefteq$, f, w) = LUBwrt($\sqsubseteq$, $\bot$, f[FPAwrt($\sqsubseteq$, $\bot$, $\trianglelefteq$, f, [Predr_Set(w)]
])

and if $\neg$ Is_CPO($\sqsubseteq$, $\bot$) **or** $\neg$ Is_Well_Ordering($\trianglelefteq$) **then** FPAwrt($\sqsubseteq$, $\bot$, $\trianglelefteq$, f, w) = $\bot$.

Theorem CPO6: $\forall$ $\sqsubseteq$: (D: $\mathcal{S}et$)$\boxtimes$D$\to\mathbb{B}$, $\forall$ $\bot$: D, $\forall$ $\trianglelefteq$: (W: $\mathcal{S}et$)$\boxtimes$W$\to\mathbb{B}$, $\forall$ f: D$\to$D, $\forall$ q: D,

if f(q) = q **then** $\forall$ w: W, FPAwrt($\sqsubseteq$, $\bot$, $\trianglelefteq$, f, w) $\sqsubseteq$ q

Theorem CPO7: $\forall$ $\sqsubseteq$: (D: $\mathcal{S}et$)$\boxtimes$D$\to\mathbb{B}$, $\forall$ $\bot$: D, $\forall$ $\trianglelefteq$: (W: $\mathcal{S}et$)$\boxtimes$W$\to\mathbb{B}$, $\forall$ f: D$\to$D,

if Is_CPO($\sqsubseteq$, $\bot$) **and** Is_Well_Ordering($\trianglelefteq$) **and** Is_Monotonic_for($\sqsubseteq$, $\sqsubseteq$, f) **then**

Is_Monotonic_for($\trianglelefteq$, $\sqsubseteq$, FPAwrt($\sqsubseteq$, $\bot$, $\trianglelefteq$, f, $\square$)) **and** $\forall$ w: W,

if Predr_Set(w) $\neq$ W **then** FPAwrt($\sqsubseteq$, $\bot$, $\trianglelefteq$, f, suc(w)) = f(FPAwrt($\sqsubseteq$, $\bot$, $\trianglelefteq$, f, w)).

Theorem CPO8: $\forall$ $\sqsubseteq$: (D: $\mathcal{S}et$)$\boxtimes$D$\to\mathbb{B}$, $\forall$ $\bot$: D, $\forall$ f: D$\to$D, **if** Is_CPO($\sqsubseteq$, $\bot$) **and**

Is_Monotonic_for($\sqsubseteq$, $\sqsubseteq$, f) **then** $\exists$ p: D $\ni$ Is_MFP_for($\sqsubseteq$, f, p).

Def. (S:$\wp$(D: $\mathcal{S}et$)) ($\sqsubseteq$: D$\boxtimes$D$\to\mathbb{B}$)$_{SS}$ (T: $\wp$(D)): $\mathbb{B}$ = ($\forall$ s: S, $\exists$ c: T $\ni$ s $\sqsubseteq$ c **and**

$\forall$ t: T, $\exists$ b: S $\ni$ b $\sqsubseteq$ t).

Corollary: $\forall$ $\sqsubseteq$: (D: $\mathcal{S}et$)$\boxtimes$D$\to\mathbb{B}$ **if** Is_Preordering($\sqsubseteq$) **then** Is_Preordering($\sqsubseteq_{SS}$)

Def. Cvx($\sqsubseteq$: (D: $\mathcal{S}et$)$\boxtimes$D$\to\mathbb{B}$): $\wp$($\wp$(D)) = { S: $\wp$(D) | S $\neq\varnothing$ **and** $\forall$ x, y, z: D,

if x $\sqsubseteq$ y $\sqsubseteq$ z **and** {x, z} $\subseteq$ S, **then** y $\in$ S }

Theorem CPO9: $\forall$ $\sqsubseteq$: (D: $\mathcal{S}et$)$\boxtimes$D$\to\mathbb{B}$, **if** Is_Preordering($\sqsubseteq$)

then Is_Partial_Ordering_of(Cvx($\sqsubseteq$), $\sqsubseteq_{SS}$).

Theorem CPO10: $\forall$ $\sqsubseteq$: (D: $\mathcal{S}et$)$\boxtimes$D$\to\mathbb{B}$, $\forall$ $\bot$: D, $\forall$ S:$\wp$(D)$\sim${$\varnothing$}, **if** Is_CPO($\sqsubseteq$, $\bot$)

then {$\bot$} $\sqsubseteq_{SS}$ S.

Def. CCwrt($\sqsubseteq$: (D: $\mathcal{S}et$)$\boxtimes$D$\to\mathbb{B}$, S:$\wp$(D)): Cvx($\sqsubseteq$) = { y: D | $\exists$ x, z: S $\ni$ x $\sqsubseteq$ y **and** y $\sqsubseteq$ z }

Theorem CPO11: $\forall$ $\sqsubseteq$: (D: $\mathcal{S}et$)$\boxtimes$D$\to\mathbb{B}$, $\forall$ S:$\wp$(D), $\forall$ T: Cvx($\sqsubseteq$), **if** S $\subseteq$ T,

then CCwrt($\sqsubseteq$, S) $\subseteq$ T.

Theorem CPO12: $\forall$ $\sqsubseteq$: (D: $\mathcal{S}et$)$\boxtimes$D$\to\mathbb{B}$, $\forall$ G:$\wp$(Cvx($\sqsubseteq$)), ($\bigcap_{S \in G}$ S): Cvx($\sqsubseteq$)$\cup${$\varnothing$}.

Def. Shadow_for($\sqsubseteq$: (D: $\mathcal{S}et$)$\boxtimes$D$\to\mathbb{B}$, T:$\wp$(D)): $\wp$(D) = { x: D | $\exists$ y: T $\ni$ x $\sqsubseteq$ y }

Theorem CPO13: $\forall \sqsubseteq: (D: \mathcal{S}et) \boxtimes D \to \mathbb{B}, \forall S, T: \wp(D)$, **if** Is_Preordering($\sqsubseteq$) **and** $S \subseteq$ Shadow_for($\sqsubseteq$, T) **then** Shadow_for($\sqsubseteq$, S) $\subseteq$ Shadow_for($\sqsubseteq$, T).

Theorem CPO14: $\forall \sqsubseteq: (D: \mathcal{S}et) \boxtimes D \to \mathbb{B}, \forall S, T:$ Cvx($\sqsubseteq$), **if** Is_Preordering($\sqsubseteq$) **then** $S \sqsubseteq_{SS} T$ **iff** $S \subseteq$ Shadow_for($\sqsubseteq$, T) **and** $\forall$ z: T, $S \cap$ Shadow_for($\sqsubseteq$, {z}) $\neq \varnothing$.

Corollary: $\forall \sqsubseteq: (D: \mathcal{S}et) \boxtimes D \to \mathbb{B}, \forall S, T:$ Cvx($\sqsubseteq$), **if** Is_Preordering($\sqsubseteq$) **and** $S \sqsubseteq_{SS} T$ **then** Shadow_for($\sqsubseteq$, S) $\subseteq$ Shadow_for($\sqsubseteq$, T).

Theorem CPO15: $\forall \sqsubseteq: (D: \mathcal{S}et) \boxtimes D \to \mathbb{B}, \forall \bot: D, \forall$ C: Chain($\sqsubseteq$), **if** Is_CPO($\sqsubseteq, \bot$) **then** $C \subseteq$ Shadow_for($\sqsubseteq$, {LUBwrt($\sqsubseteq, \bot$, C)}).

Def. Crown_wrt($\sqsubseteq: (D: \mathcal{S}et) \boxtimes D \to \mathbb{B}, \mathcal{S}: \wp(\wp(D))$): $\wp(D) = \{$ u: D $\mid \forall S: \mathcal{S}$, $S \cap$ Shadow_for($\sqsubseteq$, {u}) $\neq \varnothing \}$

Theorem CPO16: $\forall \sqsubseteq: (D: \mathcal{S}et) \boxtimes D \to \mathbb{B}, \forall \mathcal{S}: \wp(\wp(D))$, Crown_wrt($\sqsubseteq, \mathcal{S}$) $\in$ Cvx($\sqsubseteq$) $\cup \{\varnothing\}$.

Theorem CPO17: $\forall \sqsubseteq: (D: \mathcal{S}et) \boxtimes D \to \mathbb{B}, \forall \mathcal{S}: \wp(\wp(D)), \forall$ T: Cvx($\sqsubseteq$), **if** $\forall S \in \mathcal{S}, S \sqsubseteq_{SS} T$, **then** $T \subseteq$ Crown_wrt($\sqsubseteq, \mathcal{S}$).

Theorem CPO18: $\forall \sqsubseteq: (D: \mathcal{S}et) \boxtimes D \to \mathbb{B}, \forall \mathcal{S}: \wp(\wp(D)), \forall$ x, y: D, **if** $x \sqsubseteq y$ **and** $y \notin$ Crown_wrt($\sqsubseteq, \mathcal{S}$) **then** $x \notin$ Crown_wrt($\sqsubseteq, \mathcal{S}$).

Def. Interior_wrt($\sqsubseteq: (D: \mathcal{S}et) \boxtimes D \to \mathbb{B}, \bot: D$, C: Chain($\sqsubseteq$)): Chain($\sqsubseteq$) = { x: C $\mid$ $x \neq$ LUBwrt($\sqsubseteq, \bot$, C~{x}) }.

Theorem CPO19: $\forall \sqsubseteq: (D: \mathcal{S}et) \boxtimes D \to \mathbb{B}, \bot: D, \forall$ C: Chain($\sqsubseteq$), **if** Is_CPO($\sqsubseteq, \bot$) **then** $C \subseteq$ Interior_wrt($\sqsubseteq, \bot$, C) $\cup$ {LUBwrt($\sqsubseteq, \bot$, Interior_wrt($\sqsubseteq, \bot$, C))}.

Theorem CPO20: $\forall \sqsubseteq: (D: \mathcal{S}et) \boxtimes D \to \mathbb{B}, \bot: D, \forall$ C: Chain($\sqsubseteq$), **if** Is_CPO($\sqsubseteq, \bot$) **then**
$\forall$ t: Interior_wrt($\sqsubseteq, \bot$, C)~{LUBwrt($\sqsubseteq, \bot$, C)}, $\exists$ u: Interior_wrt($\sqsubseteq, \bot$, C) $\ni$ $t \neq u$ **and** $t \sqsubseteq u$.

Theorem CPO21: $\forall \sqsubseteq: (D: \mathcal{S}et) \boxtimes D \to \mathbb{B}, \bot: D, \forall$ C, C′: Chain($\sqsubseteq$), **if** Is_CPO($\sqsubseteq, \bot$) **and**
$C \subseteq C'$ **then** Interior_wrt($\sqsubseteq, \bot$, C) $\subseteq$ Interior_wrt($\sqsubseteq, \bot$, C′)

Def. (C: Chain($\sqsubseteq: (D: \mathcal{S}et) \boxtimes D \to \mathbb{B}$) Is_Coextensive_with ($\mathcal{C}$: Chain($\sqsubseteq_{SS} \upharpoonright$ Cvx($\sqsubseteq$))): $\mathbb{B}$ = (
$\exists \bot: D \ni$ Is_CPO($\sqsubseteq, \bot$) **and** Interior_wrt($\sqsubseteq, \bot$, C) $\subseteq \bigcup S$ **and**

$\text{LUBwrt}(\sqsubseteq, \bot, C) \in \text{Crown_wrt}(\sqsubseteq, \mathscr{C})$ **and**

$$C \sim \{\text{LUBwrt}(\sqsubseteq, \bot, C)\} \subseteq \bigcup_{S \in \mathscr{C}} S \sim \text{Crown_wrt}(\sqsubseteq, \mathscr{C})\ ).$$

Def. $\text{Is_Chain_Mediating}(\ \sqsubseteq: (D: \mathit{Set}) \boxtimes D \to \mathbb{B}\ ): \mathbb{B} = (\ \exists\ \bot: D \ni \text{Is_CPO}(\ \sqsubseteq, \bot\ )$ **and**

$\forall\ \mathscr{C}: \text{Chain}(\ \sqsubseteq_{SS} \upharpoonright \text{Cvx}(\sqsubseteq)\ ), \forall\ u: \text{Crown_wrt}(\sqsubseteq, \mathscr{C}), \exists\ C: \text{Chain}(\sqsubseteq) \ni$

C Is_Coextensive_with $\mathscr{C}$ **and** $\text{LUBwrt}(\sqsubseteq, \bot, C) \sqsubseteq u\ ).$

Def. $\text{Is_Chain_Covering}(\ \sqsubseteq: (D: \mathit{Set}) \boxtimes D \to \mathbb{B}\ ): \mathbb{B} = (\ \exists\ \bot: D \ni \text{Is_CPO}(\ \sqsubseteq, \bot\ )$ **and**

$\forall\ \mathscr{C}: \text{Chain}(\ \sqsubseteq_{SS} \upharpoonright \text{Cvx}(\sqsubseteq)\ ), \forall\ C: \text{Chain}(\sqsubseteq), \forall\ T: \text{Cvx}(\sqsubseteq)$, **if** C Is_Coextensive_with $\mathscr{C}$ **and**

$\forall\ S: \mathscr{C}, S \sqsubseteq_{SS} T$ **then** $\text{LUBwrt}(\sqsubseteq, \bot, C) \in \text{Shadow_for}(\ \sqsubseteq, T\ ).$

Theorem CPO22: $\forall\ \sqsubseteq: (D: \mathit{Set}) \boxtimes D \to \mathbb{B}$, **if** $\text{Is_Chain_Mediating}(\ \sqsubseteq\ )$ **and**

$\text{Is_Chain_Covering}(\ \sqsubseteq\ )$ **then** $\text{Is_CPO}(\ \sqsubseteq_{SS} \upharpoonright \text{Cvx}(\sqsubseteq), \{\bot\}\ ).$

def. $\text{Im_of}(\ a: A: \mathit{Set}, (b: A)\ \rho\ (x: D: \mathit{Set}): \mathbb{B}\ ): \wp(D) = \{\ y: D \mid a\ \rho\ y\ \},$

def. $((b: A: \mathit{Set})\ \rho\ (x: D: \mathit{Set}): \mathbb{B}) \sqsubseteq_R: (D: \mathit{Set}) \boxtimes D \to \mathbb{B}\ )\ ((b: A)\ \sigma\ (x: D): \mathbb{B}): \mathbb{B} = (\ \forall a: A,$

$\text{Im_of}(\ a, \rho\ ) \sqsubseteq_{SS} \text{Im_of}(\ a, \sigma\ ),$

corollary: $\forall\ A: \mathit{Set}, \forall\ D: \mathit{Set}, \forall \sqsubseteq_R: (D: \mathit{Set}) \boxtimes D \to \mathbb{B},\ \ \text{Is_Preordering}(\ A \boxtimes D, \sqsubseteq_R\ ),$

def. $\text{Is_Ord_Resp_for}(\sqsubseteq: (D: \mathit{Set}) \boxtimes D \to \mathbb{B},\ \ (a: A: \mathit{Set})\ \rho: A \boxtimes D \to \mathbb{B}\ (x: D: \mathit{Set})\ ): \mathbb{B} = (\ \forall a: A,$

$\exists\ w: D \ni a\ \rho\ w$ and $\forall x, y, z: D$, if $a\ \rho\ x$ and $a\ \rho\ z$ and $x \sqsubseteq y \sqsubseteq z$, then $a\ \rho\ y\ ).$

def. $\text{ORR}(A: \mathit{Set}, (D: \mathit{Set}) \sqsubseteq (D): \mathbb{B}): \wp(A \boxtimes D \to \mathbb{B}) =$

$\{\ \rho: A \boxtimes D \to \mathbb{B} \mid \text{Is_Ord_Resp_for}(\sqsubseteq, \rho\ )\ \}.$

Implicit_Def. $(A: \mathit{Set} \boxtimes D: \mathit{Set} \to \mathbb{B}) \bullet_{\sqsubseteq} (\sigma: B: \mathit{Set} \boxtimes D \to \mathbb{B}\ ): \wp(A \boxtimes D \to \mathbb{B})$ **is**

$\forall u: A, \forall v, w: B, \forall x,y,z: D, u\ (\rho \bullet_{\sqsubseteq} \sigma) y$ **iff** $u\ \rho\ v\ \sigma\ x$ **and** $u\ \rho\ w\ \sigma\ z$ **and** $x \sqsubseteq y \sqsubseteq z;$

$\forall A, B, D: \mathit{Set}, \forall \sqsubseteq: D \boxtimes D \to \mathbb{B}, \forall \rho: A \boxtimes B \to \mathbb{B}, \forall \sigma: B \boxtimes D \to \mathbb{B}, \rho \bullet_{\sqsubseteq} \sigma \in \text{ORR}.$

corollary: $\forall\ A, B, D: \mathit{Set}, \forall \sqsubseteq: D \boxtimes D \to \mathbb{B}, \forall \rho, \forall \sigma, \forall u: A, \forall y: D,$

if $u(\rho \circ \sigma)y$, **then** $u(\rho \bullet_{\sqsubseteq} \sigma)y.$

⋮

end Complete_Partial_Orders;

It is not feasible to explain here what each line of the CPO theory is about. For our purposes it suffices to point out that the first part of the unit introduces the usual concepts of upper bounds, least upper bounds, and chains found in any discussion of complete partial orders. However, here those concepts are presented in terms of our general set theory. Similarly, the ideas of monotonicity and convexity in our general setting are made rigorous.

Our goal is to define composition of relations in such a way that we can express the idea of chains of relations based on the intuitive notion of less defined, i.e., one relation is less defined than another. This gives us the expressive power to talk about monotonic chains of relations and then to pose questions concerning least upper bounds for those chains. Once we have this capability, we can define semantics for loops and recursive procedures in terms of these least upper bounds.

For example, to explain the meaning of a **while** loop, one can associate with each iteration a relation describing what has the loop has done up to that point. With each successive iteration, the loop so far is associated with a new relation "more defined" than the previous one, eventually leading to a least fixed point of the functional that gives meaning to the loop as a whole.

As the dots in the last section of the CPO theory indicate, additional definitions and theorems will be added to complete the unit that will serve as a basis for doing denotational semantics for our RESOLVE programs.

A Look to the Future

Developing denotational semantics for assertive programs proves to be a challenge in many ways. Not only is it necessary to give meaning to traditional programming constructs, but it is also necessary to provide meaning for such constructs as concepts, realizations, and math units. Sorting out how to do set theory and then how to use it for doing complete partial orders is a good beginning, but there is certainly more work to do to address all the semantic issues that exist in our approach to doing software.

References

1. Ogden, W.F., CIS 680 Course Packet, Spring, 2002.

2. Sitaraman, M, Weide, B. and Ogden, F., "Using Abstraction Relations to Verify Abstract Data Type Representations," IEEE Transactions on Software Engineering, March 1997, 157-170.

3. Sitaraman, M., and Weide, B.W., eds. Component-based software using RESOLVE. *ACM Software Eng. Notes 19,4* (1994), 21-67.

4. Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B. W., Long, T. J., Bucci, P., Heym, W., Pike, S., and Hollingsworth, J. E., "Reasoning About Software-Component Behavior," *Procs. Sixth Int. Conf. on Software Reuse*, LNCS 1844, Springer-Verlag, 2000, 266-283.

5. Special issue on Real-Time Specification and Verification, *IEEE Trans. on Software Engineeering*, September 1992.

Reaping More from Lazy Initialization Using Dynamic Reconfiguration

Nigamanth Sridhar
Computer and Information Science
Ohio State University
2015 Neil Ave
Columbus OH 43210 USA

nsridhar@cis.ohio-state.edu
Phone: +1 614 292 8234
URL: http://www.cis.ohio-state.edu/~nsridhar

Abstract:

Lazy initialization is a technique that can be used to improve the performance of software components. The startup cost of any component is greatly reduced, and the cost of initialization is amortized over the various operations to the component throughout its lifetime. However, after a certain point in the component's lifetime, this stops being a benefit, and becomes a performance penalty instead. This paper examines the use of dynamic reconfiguration to reap the benefits of lazy initialization and yet stopping it from being a penalty.

Keywords. Lazy initialization, modularization, dynamic reconfiguration, parameterization.

Paper Category: Position paper.

Emphasis: Research.

1 Introduction

Among a set of implementations of a given component, all other things being equal, the fastest implementation is preferred. We are therefore, constantly looking for ways to make our software components more efficient. Lazy initialization is one technique that improves the performance of components [4,5]. Component implementations that use lazy initialization typically do not allocate memory and initialize variables when they are declared by a client program. Instead, they wait for the client to actually use the variable, and on the first such use, allocate the requisite amount of memory and initialize the variable. After this, the actual operation is performed.

It is easy to see that this approach greatly reduces the startup cost of the application, because there is no time spent in initializing variables at startup. So the time taken to initialize a bunch of variables at one time has now been spread over a longer period of time while the application has been making useful progress.

However, the downside to lazy initialization is that in long-running systems, the cost of checking whether the variable has been initialized in *every* operation could become very high. If we can find some way by which the component could stop checking once its variables have been initialized, then we can exploit the low startup cost of lazy initialization and the low cost of an implementation that assumes initialized variables.

This paper examines the use of dynamic reconfiguration to achieve just this. In Section 2, we present a more detailed introduction to lazy initialization including some instances where the performance improvement is surprisingly great. We describe dynamic reconfiguration in some detail, and the specific domain of dynamic reconfiguration we use in this paper -- module replacement in Section 3. After presenting some ideas on how dynamic reconfiguration can be included in component implementations in Section 4, we conclude in Section 5.

2 Lazy Initialization

Consider an operating system, which at startup, creates and initializes a number of processes and data stores. The startup time for any operating system is considerably large given the number of services that need to be started up. These services really need to be started up when the machine is booted up. For example, the FTP server on a machine cannot really be started as response to any user action, since the ftp requests come from *outside* the machine, and the user at the machine typically may not even know about the requests until they arrive.

So while it cannot be avoided that startup costs of certain applications will be high, the effort on our part should be on reducing such a cost to as little as possible. Lazy initialization is a technique that greatly helps in this respect. The creation of services and objects cannot be avoided, but they need not be necessarily initialized at the time they are created. The initialization of these data stores can be postponed until they are used for the first time. And if some objects never get used during the lifetime of the application, all the better - we need not waste the time it takes to initialize the object, and further to finalize and destroy the object.

But is the cost of initialization typically that large in order to realize an appreciable gain in performance using lazy initialization? Consider an **Array** component that creates generic **Array** objects of arbitrary size. Let us suppose that a client program instantiates this component to create **Array**s of **Array** objects to create two dimensional **Array**s. When the client creates an object of this type, look at how long it takes to initialize it. The top level array needs to be allocated memory, which is linear in the amount of time it takes to initialize each of the second-level arrays. This time is itself linear in the time it takes to initialize each item in the array. The cost of initializing a 100 x 100 array of this type could be considerably large depending on the initialization cost of the individual items!

As an alternative, consider an implementation of this component that is represented as follows. An **Array** object is just a logical reference[1] to an array of logical references to object of the item type that the **Array** is instantiated with. The cost of creation for such a structure is still linear in the number of elements in the array. However, the cost of creating any single reference is extremely small. So regardless of what the actual item type in the array is, the cost of creation is still linear in the cost of creating a single reference.

So what happens with this array going forward? Every time an operation tries to access a particular location in the array, that location is initialized and the access is done. So instead of paying the cost of initializing the entire data store up front, the cost is amortized over multiple accesses to the data store.

If lazy initialization is such a great improvement in performance, why don't we use it all the time? What are the downsides to using lazy initialization for all applications?

The problem crops up only in systems that have a long running time. On the first access to any part of the data store, that part is initialized. However, in spite of this, every access to this data store entry in the future will also have to go through the check to see if the entry has been initialized or not. Though this is just a simple check, it could turn out to be a considerable cost that overtakes the benefit of lazy initialization. In a system that runs for a very long time - days, or even months - such a cost at every access proves very expensive. In the next section, we examine the use of dynamic reconfiguration to address this problem.

3 Dynamic Reconfiguration

Dynamic reconfiguration [1] broadly refers to changing (adding, deleting or modifying) some part of a software system *while* the system is running. The change is performed without disrupting the running of the system in any way. Dynamic reconfiguration is essential for supporting the operation and evolution of long-lived and highly-available systems for which it is either not possible or not economic to terminate execution in order to carry out invasive maintenance activities. For such applications, systems must be reconfigured ``on-the-fly'' at runtime.

Several dimensions of an application's configuration can be changed at runtime. In this paper, we focus on a common mode of dynamic reconfiguration known as *module replacement*. Module replacement -- also called *hot swapping* -- involves rebinding or substituting the implementation of a module at runtime. Module replacement is a very powerful technique when software is decomposed into modules that encapsulate independent design decisions that are likely to change over the lifetime of an application [2].

In the particular case in hand, how do we reconfigure the **Array** component? In other words, what modules do we replace in order to achieve the desired change? At the outset, it seems like we need to change the entire implementation of the **Array** component. This might not be so easy, since this might mean that there has to be a way for the new component to acquire ``knowledge'' from the old component about the objects already created, and their state and representations.

Further, if the representations used by the old and new implementations of **Array** are not the same, the reconfiguration may require that the representations of *all* the objects created so far be changed to correspond to the new representation. Such a change only seems like a whole performance loss, and defeats the whole purpose of the reconfiguration.

A more clever way to do this would be to recognize that the initialization and finalization of data objects are independent design decisions. So the component could be parameterized by **Initializer** and **Finalizer** modules. In the beginning, the **Array** component is instantiated with a **Lazy_Initializer** implementation which just creates objects but uses a lazy scheme

for initialization. Later on in the lifetime of the application, the component could be reconfigured to replace the **Initializer** module with a regular implementation that does not check if an object is initialized on every access.

The question that remains to be answered is when does this reconfiguration get triggered, and who does this. This can be done in a variety of ways. The component could itself be coded in such a way that it somehow ``senses'' that lazy initialization is proving to be a performance burden on the system and ``heal'' itself. On the other hand, the client program could make the change, by instructing the component to change itself. Or, the user could provide some kind of external stimulus to the application - such as a console command - to trigger the change.

4 Implementation

How do we go about implementing the ideas that have been described in the preceding sections? This section presents a way in which dynamically reconfigurable components can be implemented in popular languages and component technologies such as Java and .NET.

In previous work, we introduced the **service facility pattern** [3] as a way of implementing parameterized software components in languages and environments that did not support generic programming through first-class language constructs. A service facility exports zero or more data types, plus the operations defined on the data types it exports. A client can create a service facility object, request data objects from the service facility, and also use the facility to perform operations on the data objects it uses. In effect, a service facility object acts just like a RESOLVE facility, except that the service facility is a runtime entity that gets created, instantiated and destroyed during the lifetime of an application.

Languages like C++ and Ada have language constructs (templates and generics, respectively) that allowed the programmer to create a single component that could be specialized at compile-time by supplying appropriate template parameters. In languages like Java, however, which do not support a generic programming mechanism, creating parameterized components is quite a challenge. The service facility pattern provides a way for creating parameterized software components in such languages.

The C++ preprocessor uses a template and its actual parameters to create a new type that can be used in the program. This step is called *template instantiation*. Once a template has been instantiated, variables of that type can be declared and used. The service facility pattern also involves two steps: (1) ``instantiating'' the service facility by supplying the appropriate parameters and (2) creating and using variables of the type exported by the service facility. Each parameter to the service facility are supplied by invoking a method of the form **set*ParameterName*** on the service facility object. A client using service facilities first performs this step, similar to the template instantiation stage in C++.

Since template parameters are bound to the service facility at runtime, they can also be *changed* at runtime. In fact, the change is as simple as re-invoking the correct **set*ParameterName*** method with the new parameter. In our **Array** example, the **Array_SerF** service facility would have operations **setInitializer** and **setFinalizer** to set the initialize and finalize modules respectively. So at some point in the application's lifetime, the client program could make a call on the **Array_SerF** object to change the initialization strategy from lazy initialization to regular initialization.

So how does this reconfiguration affect the client program? Since the change in the initialization strategy does not really affect how the rest of the component functions, no other change is necessary in the component. The representations of the existing data can remain the same. Since the service facility object acts as a wrapper to the actual **Array** variables, it presents a nice abstraction barrier behind which the configuration can be changed, while still presenting a view to the client that nothing has changed. The client still depends only on the published interface of the component, and as long as that remains intact, the client does not see the change.

5 Conclusion

In this paper we have presented a way for component implementers to effectively use the lazy initialization approach to improving the performance of software components, while at the same time, keeping the potential performance penalties of the approach to a minimum. By dynamically changing a part of a component's implementation, we are able to leverage the fast startup guaranteed by lazy initialization, and the long term benefits of not having to (unnecessarily) checking on whether a particular data object has been initialized or not.

The idea of dynamic reconfiguration can be extended to other similar areas. For instance, in components that have state that only changes monotonically, it may be possible to dynamically change the checking of preconditions of operations depending on how the system is behaving over its lifetime.

Bibliography

1

KRAMER, J., AND MAGEE, J.
Dynamic configuration for distributed systems.
IEEE Transactions on Software Engineering SE-11, 4 (1985), 424-436.

2

PARNAS, D. L.
On the criteria to be used in decomposing systems into modules.
Communications of the ACM 15, 12 (Dec. 1972), 1053-1058.

3

SRIDHAR, N., WEIDE, B. W., AND BUCCI, P.
Service facilities: Extending abstract factories to decouple advanced dependencies.
In *Proceedings of the 7th International Conference on Software Reuse* (Austin TX, April 2002), no. 2319 in LNCS, Springer-Verlag, pp. 309-326.

4

WEIDE, B. W.
Software Component Engineering.
OSU Reprographics, 1997.

5

WEIDE, B. W., AND HARMS, D.
Efficient initialization and finalization of data structures: Why and how.
Tech. Rep. OSU-CISRC-8/89-TR11, Ohio State University, Columbus OH, August 1989.

Footnotes

... reference[1]
Note that the use of references in this case is not exposed to the client programmer in any way. The references are all buried below the abstraction barrier that the component provides. The client uses the component only through its published interface, and does not have access to the actual representations.

---

nsridhar@cis.ohio-state.edu

Good News and Bad News About Software Engineering Practice

Bruce W. Weide
Dept. of Computer and Information Science
The Ohio State University
Columbus, OH 43210-1277 USA

weide.1@osu.edu
Phone: +1 614 292 1517
Fax: +1 614 292 2911
URL: http://www.cis.ohio-state.edu/~weide

Abstract

Good news: some language/design features that support effective software engineering practices (including some of those advocated by the RESOLVE group) have been appearing incrementally in commercial software technologies. Bad news: none of the RESOLVE-specific innovations is yet among them. In fact, despite incrementally embracing some good ideas, commercial software technologies overall have regressed in the sense that they have become so complicated that the ill effects of their complexity easily outweigh the isolated benefits offered by these incremental improvements. Significant additional progress might have to wait for existing approaches to collapse under their own weight.

Keywords

Commercial software technologies, C++, CORBA, Java, .NET, RESOLVE

Paper Category: position paper
Emphasis: research

1. Introduction

The popular vision of the bright future for information technology relies heavily on Moore's Law of hardware improvement: that memory capacity, communication bandwidth, and raw computing power double roughly every eigtheen months. Drowned in the wake of enthusiasm is "God's Law", expressed by William Buxton [Buxton02] as the fact that human capacity for understanding is essentially constant. Buxton was talking about the complexities faced by the human end-users of software systems, but the same principle applies to the human software engineers who design, build, and maintain them. If software engineers try to build systems that grow in complexity at anything like the pace of Moore's Law in an attempt to do something with all those available bytes and cycles (Buxton calls this tendency "Buxton's Law"), then those programs will quickly dwarf the abilities of software engineers to manage their scale.

The above observations imply that the technical barrier to "scaling up" in software engineering is that software design approaches must support compositional, or modular, reasoning techniques about software behavior. That is, a necessary condition for scalability is that the behavioral effects of software changes are localized and predictable [Weide95]. Put otherwise, *component-based* development is not necessarily *scalable* development. So, the overarching goal of the RESOLVE project has long been to provide a rigorous mathematical foundation for scalable software development, to put more science behind the engineering, in the hope of making it intellectually manageable for engineers to reason soundly about the behaviors of the software systems they build--even as those systems get bigger and bigger and offer more and more functionality and better and better performance.

The argument that ultimately it is necessary to be able to reason modularly about software system behavior has led Bill Ogden to characterize the key ideas underlying the RESOLVE work as "inevitable", i.e., virtually certain to be adopted by software engineering practitioners--eventually. An examination of current software engineering practice reveals that there is both good news and bad news on this front, which I discuss in support of the position stated in Section 2.

Buxton's analysis of Moore's Law vs. God's Law raises the question of whether we might *already* have exceeded the level of complexity that software engineers can be expected to deal with. The consequences of such a landmark event, especially in the case of embedded software and other mission-critical systems, are certain to be disastrous. Indeed I now believe that, regrettably, the only way we will see significant improvements in software engineering practice is if some truly catastrophic event(s) can be blamed on defective software. Yes, software engineering practice has improved over the years if you look at

some of the details outlined in Section 2. But numerous "bad habits" that we try to teach our students to avoid also have become institutionalized in commercial software technologies (CSTs) such as Microsoft's .NET framework, the Java milieu, CORBA, OpenSource alternatives, etc. Moreover, there is just an astonishingly high level of overall intellectual complexity involved in dealing with CSTs. Microsoft, Sun, and apparently all their "competitors" are on the same bandwagon, obliging software developers to deal with new and greater complication at every turn. Here I see only bad news, as discussed in support of the position stated in Section 3.

2. We Have Had No Impact On SE Practice

Some of my friends have noted that my mood has been sort of "down" lately, and it's partly because of what a realistic analysis reveals about the impact so far of a research program lasting nearly 20 years:

> None of the RESOLVE innovations has had the slightest influence on CSTs, *even* via their inevitability.

It is worth noting that several of the ideas that we have always touted in RESOLVE and its ancestors have now been more-or-less embraced by the purveyors of CSTs. We see clear evidence of the widespread recognition that these are good ideas. Unfortunately, we also see evidence that the problems addressed by these ideas and/or the recommended solutions are (to put it charitably) incompletely understood by those who have injected them into CSTs. The following table illustrates what I mean for several examples.

Idea	Evidence of Recognition of Value	Evidence of Incomplete Understanding
separating specifications from implementations	interfaces are first-class units in Java, C#, IDLs	interfaces are just signatures, with not even syntactic slots for behavioral specifications
allowing for multiple interchangeable implementations of a single specification	design-to-interfaces is recommended or even required practice in all modern CSTs; design patterns to address the multiple-implementation issue, especially the abstract factory pattern, are widely used	design-to-interfaces is not design-by-contract; design patterns are clumsy compared to relatively simple language mechanisms that could address the issue head-on
having a standard set of "horizontal", general-purpose, domain-independent components such as lists, trees, maps, etc., in a component library	STL and java.util include such components and they are widely used	designs of the components in the STL, java.util, etc., are subtly but importantly different from what we would have created, in that they do not support modular reasoning; no one has provided convincing empirical evidence that there is much value in not recreating such "simple" components from scratch, although any developer worth his salt would now readily testify to this despite the unaddressed reasoning problems
templates are a useful composition mechanism	C++ added a template mechanism and eventually it "worked" on most/all C++ compilers because the STL required it; Java is supposed to get a template mechanism soon	Java doesn't yet have templates, and when it does they will be a weak substitute for what is actually required; parameterized components aren't on the radar screen in the .NET literature
having value semantics is useful even for user-defined types	STL users are advised to override the C++ assignment operator to make a deep copy [Musser01]; "clone" is an integral part of Java and the .NET framework; .NET languages use "boxing" to try to eliminate ugly syntax associated with Java's wrappers for value types	*no one* except us has noticed that swapping is a much better alternative than deep copying to achieve value semantics; Java's approach to cloning was so hopelessly botched from the start that even the designers of the java.util classes never found a way to make it work; the .NET framework simply adopts the Java approach to cloning, flaws and all; eliminating ugly syntax in .NET languages does not eliminate ugly semantics, but rather makes it harder to notice that something funny is going on when combining value and reference types
reasoning about programs that use pointers/references is complicated and error-prone	early hype about Java proclaimed that "[p]ointers are one of the primary features that enable programmers to put bugs into their code... Thus the Java language has no pointers." [Gosling96]; authors of a	early Java hype was later recanted when someone realized that merely eliminating pointer syntax did not actually solve the fundamental problem with pointers [Weide01]; it's not known how many instructors have adopted the

	mainstream C++ textbook [Koenig00] argue that students find values much easier to deal with than references and pointers, and advocate teaching the latter as late as possible	improved C++ pedagogy offered by [Koenig00], but it is clear that no one else has adopted the improved pedagogy offered by us [Sitaraman01]
problems related to storage management, such as memory leaks, are serious	Java and .NET try to eliminate developer concern by mandating garbage collection as the solution; it appears that even the current GNU C++ compiler is implemented on top of a garbage-collecting C++ substrate	reliance on garbage collection has some bad consequences for performance of interactive systems, renders mainstream CSTs useless or dangerous for building real-time applications, and seems to require the developer to know details of the garbage collector implementation in order to manage scarce resources other than memory

With the generally positive development that good ideas are making their way into CSTs, why is my first position so downbeat? To my knowledge, the RESOLVE work has *never* been cited by anyone responsible for introducing any of these ideas into CSTs; in fact, it's rarely been cited by anyone except us. So, what I mean by the first position is that, even if RESOLVE had never existed and even if we hadn't written a single paper about our work, CSTs would still be just what they are now.

One of the obvious problems we've always faced has been the all-or-none nature of RESOLVE. Almost any little part of our technology that you decide *not* to adopt is likely to result in the inability to do modular reasoning in some cases. The positive developments listed above were adopted incrementally, and--this is a key point--apparently without any explicit concern for whether they might improve support for modular reasoning. This is why I don't think any of the apparent progress is because inevitability has already kicked in. The inevitability argument isn't based on the notion that some of the ideas are "cool" in isolation, but that together they are indispensable for modular reasoning. This rationale hasn't sold at all.

3. CST Complexity Is Out Of Control

My second position also has a negative tone, I'm afraid:

> The intellectual load imposed by current CSTs has already exceeded the ability of some software engineers (e.g., me) to cope with their complexity.

One of the most important perks of being a tenured professor is that (at some universities) you're eligible to take a sabbatical leave at reduced pay once every n years. Of course, the term "sabbatical" suggests that $n = 7$ is the appropriate choice. Ohio State chose $n = 8$ for some reason; but that's beside the point. I'm just glad there *is* a sabbatical program here and that "reduced pay" is still almost enough to live on (if you save during the other seven years and the markets are kind to you).

With seven out of every eight years spent exploring the state of the art in software engineering, especially in a formal-methods context as we do, it would be easy for me to get lost in the ivory tower and ignore what's happening in "the real world". I have therefore eagerly taken advantage of sabbatical leaves whenever I've been eligible in an attempt to avoid this hazard. Each time, an important personal objective has been to make sure that I have gained some understanding of the current state of the practice of software engineering. This has meant learning something about the CSTs of the day and actually using some of them. As noted above, today's CSTs include Microsoft's .NET framework and its COM/DCOM/COM+ predecessors, the Java language and libraries, CORBA, and OpenSource tools such as NetBeans (all of which became important only well after my last sabbatical). They also still include the C++ language and libraries (which were barely around eight years ago, as the STL was only marginally compilable at the time).

Well, this is one of those years--my third sabbatical. I almost titled this paper "What I Did On My Sabbatical". But, as you will see, this section is more about what I did *not* do.

On my first sabbatical in 1985-86, my CST experience involved working with Mike Stovsky (a graduate student at the time) to design and build a Macintosh application called MacSTILE, and a companion tool called the Part Protector. I wrote MacSTILE, and Stovsky wrote the Part Protector, which was a proof-of-concept for part of his Ph.D. dissertation work. We wrote these systems in C and used the (relatively new, at the time) "Macintosh toolbox". I considered this sabbatical successful in that I finished what I set out to accomplish and in the process learned in depth one of the most important CSTs of the day. Amazingly, MacSTILE and the Part Protector *still run* on the newest Macintoshes! The source code hasn't been touched in at least 12 years. Ah, the good old days.

On my second sabbatical in 1993-94, my CST experience involved doing some of the early development of the RESOLVE/C++ ideas that were pioneered by Steve Edwards and Sergey Zhupanov (graduate students at the time). I considered this sabbatical a success in that I learned a lot of details about object-oriented programming using one of the most

important CSTs of the day. Although the software I wrote was subsequently replaced, by me and others, over the following couple years, the core of this project remains in place today and is used by about 1000 students each year in our CS1/CS2 sequence. The biggest problem we've faced recently involved upgrading to the new GNU C++ compiler and watching it collect its own garbage for minutes at a time while we were trying to compile a small program. (We believe this is the result of a compiler bug that is encountered on this particular program; at least, we hope so.) In summary, things were more complicated in C++ than in C, but I could still get my mind around them thanks to the expertise of Edwards and Zhupanov.

On my current sabbatical in 2001-2002, my CST experience has involved working with Paolo Bucci, Wayne Heym, and Tim Long on the next generation of the Software Composition Workbench tool. Bucci built a prototype version a few years ago as a Java application and it has been used by our students ever since for some of their CS1/CS2 assignments. The sabbatical year isn't over yet, but it is notable that we have made considerably less progress on this project than we had imagined we would. Why? I'm sure my colleagues will unselfishly blame themselves for some of the troubles we've endured, but this would be very unfair to them and would distract us from the real problem: the nearly unmanageable intellectual complexity of CSTs today.

We chose to build the new SCW using Java servlets. This was not a rash decision. We didn't want to select something that was too new and unstable (e.g., wait for .NET), or something that was too old and crusty although new since my last sabbatical (e.g., write a plain old Java application). I still don't think servlets were a bad choice in terms of the complexity of the particular CST compared to the alternatives. Especially after having attended a .NET workshop for the past two days, I am convinced that .NET wasn't what we needed; it is even more complicated!

What did we have to do to build a Web app?

- Bucci had to install and configure Apache, the Tomcat servlet engine, and our chosen IDE, NetBeans. Fortunately, the rest of us didn't need to learn how to do this, and I still don't know how. Suffice to say that just from watching part of the process I could tell that it was a lot easier installing and configuring Symantec C on the Mac in 1985-86 or using emacs and gcc on Unix in 1993-94. Admittedly, all the pieces of this present-day CST are somewhat (in truth, only marginally) more "powerful" than the older technologies, but this also makes them more unwieldy. Witness, for example, the configurability profiles of Apache and Tomcat, both of which seem incomprehensible to someone like me who's not an expert in networking and security. And these are quite tame compared to NetBeans. Among programs I've used, only emacs is in the same league as NetBeans in terms of the complexity of configuration. I suppose that theoretically it's possible to use NetBeans "out of the box" just like emacs, but as a practical matter this doesn't work so well. For some reason, unless you turn off dozens of options ("modules") that you never use, NetBeans uses enough memory to put Microsoft Word to shame. And no matter what you do, NetBeans seems to garbage-collect so often and for so long on each occurrence that sometimes there is literally enough time to get yourself a cup of Java (er, coffee) before you can type in the next character.
- We had to learn HTML at a much more serious level than we already knew it "by osmosis": frames, forms, hidden inputs, etc. Actually, this was the easy part. We also had to internalize the outrageously convoluted operational model of a web app, in which the servlet repeatedly handles a GET or POST request from an HTML form in the browser and sends back a new HTML page in response. The details of this interaction model are, to put it mildly, totally unnatural for someone like me who's used to reasoning about procedure calls. There seems to be no comprehensible way to write a web app that does even approximately what MacSTILE did rather easily in 1986 with the Macintosh toolbox, or what Bucci did with a standalone Java application with the prototype of the SCW tool. I believe designing the detailed structure of a web app is akin to writing assembly code for an instruction set with delayed branching, although I've not thought in depth about the possible parallels here.
- There seems to be no standard way for web apps to interact with HTML forms that are even moderately complex and context-dependent, so we had to figure out how to do several different things that didn't appear to be at all straightforward: bring up new browser windows, conditionally close browser windows, etc. The only way to do this seems to be to embed JavaScript in the HTML responses from the servlet. This meant we had to learn JavaScript, too, and a lot of it because most of what we needed to do did not entail merely copying code out of a book. And the name notwithstanding, JavaScript is nothing like Java except in its ugly syntax. Even the object model is completely different.
- Once we used JavaScript, we knew we were headed for testing trouble. Everything had to be tested under both Netscape and Internet Explorer, and many of the JavaScript features we thought we needed didn't work the same under both popular browsers.

Where this left us was, as of April 2002, not very far along. We decided to junk the web app idea and write a Java application. Fortunately, by following RESOLVE principles when designing the Java code of the servlet, it seems we will be able to use all the back-end components as-is and just unplug the web-based user interface code and plug in a new one using Java's Swing package. With luck, we'll be able to get a decent second prototype working Real Soon Now.

It might be argued that other CSTs would have caused fewer problems. I doubt it. My limited understanding of .NET as obtained from about twleve hours of instruction from an expert, for example, does nothing to instill any such confidence.

4. Conclusion

My railing against the complexity of CSTs should not be interpreted as unmitigated criticism of their purveyors. Of course, these folks have software to get out the door, they have competition (of sorts), and most important they have to make things at least partly backward-compatible with their previous offerings. Still, they've made some strange decisions because they just don't appear to understand God's Law. Yes, computing has changed. Some additional complexities over the way we used to do things are simply necessary to create a robust and general software technology for today. For example, I'd argue that having specifications is one necessary additional complexity that still is *not* in CSTs. Instead, behind nearly every new complexity that *is* in today's CSTs lies one or more of the following:

- inadequate understanding of the problem to be solved;
- ignorance of better solutions that had already been suggested; and/or
- failure to elaborate the criteria for an acceptable solution--among which *must* be both support for modular reasoning, and understandability by reasonably competent software engineers.

Let me close on a more positive note. The incremental adoption of some important ideas in CSTs should give us some hope that other innovations we've been promoting will eventually appear in CSTs, too. And it could mean that our deeper understanding of some of the new CST features that practitioners now have to deal with (deriving from our explorations of their mathematical foundations and our experience with their use in combination with many other such features) might give us "hooks" into the practitioner's world that we might be able to leverage in the future to have some impact on software engineering practice. This suggests some questions for discussion at the workshop:

- Which yet-do-be-adopted good ideas would we most like (or are we most likely) to see in CSTs, assuming that all such ideas are to be introduced incrementally?
- What, if anything, can we do to better "market" RESOLVE ideas; e.g., can we show that any have value as incremental improvements *even in the absence* of arguments about modular reasoning?

About a decade ago, Joe Hollingsworth asked that we devote some RSRG meetings to developing a "business plan". It was a useful exercise, resulting (as I recall) in a plan to write what turned out to be the *ACM Software Engineering Notes* special section on RESOLVE [Sitaraman94], among other things. It's probably time to do that again, even though coming out of this sabbatical I now believe that no matter what we do, it is questionable whether we can really influence CSTs even via inevitability until the world experiences some serious software-caused disasters.

References

[Buxton02]
Buxton, W., "Less is More (More or Less)", in *The Invisible Future*, P. Denning, ed., McGraw-Hill, 2002, pp. 145-179.

[Gosling96]
Gosling, J., and McGilton, H. *The Java Language Environment: A White Paper*, Sun Microsystems, Inc., 1996; http://java.sun.com/docs/white/langenv/ viewed 8 May 2002.

[Koenig00]
Koenig, A., and Moo, B.E., *Accelerated C++: Practical Programming by Example*, Addison-Wesley, 2000.

[Musser01]
Musser, D.R., Derge, G.J., and Saini, A., *STL Tutorial and Reference Guide, Second Edition*, Addison-Wesley, 2001.

[Sitaraman94]
Sitaraman, M., and Weide, B.W., eds., "Special Feature: Component-Based Software Using RESOLVE", *Software Engineering Notes 19*, 4 (Oct. 1994), 21-67.

[Sitaraman01]
Sitaraman, M., Long, T.J., Weide, B.W., Harner, J., and Wang, C., "A Formal Approach to Component-Based Software Engineering: Education and Evaluation", *Proceedings 2001 International Conference on Software Engineering*, IEEE, 2001, 601-609.

[Weide95]
Weide, B.W., Heym, W.D., and Hollingsworth, J.E., "Reverse Engineering of Legacy Code Exposed", *Proceedings 17th International Conference on Software Engineering*, ACM Press, 1995, 327-331.

[Weide01]

Weide, B.W., and Heym, W.D., "Specification and Verification with References", *Proceedings OOPSLA Workshop on Specification and Verification of Component-Based Systems*, ACM, October 2001; http://www.cs.iastate.edu/~leavens/SAVCBS/papers-2001 viewed 8 May 2002.

Making the Case for Assertion Checking Wrappers

Stephen H. Edwards
Dept. of Computer Science
Virginia Tech
660 McBryde Hall, MS 0106
Blacksburg, VA 24061 USA

edwards@cs.vt.edu
Phone: +1 540 231 5723
Fax: +1 540 231 6075
URL: http://people.cs.vt.edu/~edwards/

Abstract

Defensive programming practices have not kept up with the evolution of current programming languages or techniques. While appropriate for use by the original developer, embedded assertions are difficult to use and control for clients of reusable components distributed in binary form--a perspective that is increasingly important with the rise of component-based development. A new strategy for using run-time assertion checks based on design by contract is proposed where checking code is placed in a separate wrapper class that can easily be inserted or removed. Use of a factory pattern for component creation allows client code to remain unchanged when switching between enabled or disabled checking (typically without requiring recompilation). There is no run-time penalty for checks when they are disabled, yet full control over when to enable checks--and how thoroughly checking is performed--can be left up to the client of the component. Further, both the checking wrapper and the underlying component can be distributed in compiled form, giving the client control without requiring source code access or recompilation of the component itself.

Keywords

Defensive programming, design by contract, wrapper class, decorator, unit test, precondition, postcondition, invariant, object-oriented programming, debugging aids

Paper Category: technical paper
Emphasis: research

1. Introduction

Defensive programming is a well-known coding practice with a 30-year history [Parnas72]; it has been taught to programmers for decades. The core idea is this: if a routine or module depends on certain assumptions in order to operate correctly, add code to check that those assumptions hold at run-time. When the assumptions reflect how "correct" clients should interact with your software, such "defensive" checks protect your code from incorrect or inappropriate usage.

Within the realm of object-oriented programming, Bertrand Meyer's concept of *design by contract* [Meyer92, Meyer97] lays out a clear division of responsibilities between a class implementation and its clients regarding what each party may assume and what each party is obligated to ensure. Phrasing interface contracts as method preconditions, method postconditions, and class invariants, either informally or using some model-based specification approach [Wing90], is one way to clearly and precisely define both sides of the contract between a component and its clients.

Unfortunately, while programming languages and design methods have evolved since the early 70's, mainstream practice regarding how to provide run-time checking of interface contracts in the spirit of defensive programming has changed little. Generally speaking, programmers still write Boolean tests to check preconditions (well, those that are easy or cheap to check, perhaps) and insert this code at the start of method bodies. A preprocessor or a specially designed tool may be used to conditionally include or exclude this checking code during the build process. This strategy is typified by the use of `assert` macros in C++ or similar utility classes in Java.

Run-time assertion checks do provide proven benefits [Voas97, Edwards00]. Unfortunately, the traditional approach of conditionally including assertion checking directly within class methods is most useful to the original developers of a component--those who have direct access to its implementation source code and can easily recompile it with alternative settings. Indeed, the general philosophy when using assertions is to enable them during development and testing, but disable or remove them for deployment.

That strategy is more difficult to manage for clients or reusers of such a component--particularly when the component has been purchased commercially or is distributed in compiled form only. In this case, the original developer may complete development of and release a software component, which may then be used by many other developers in creating a variety of software applications. Of course, these component clients can still benefit greatly from assertion checks in the original component; after all, defensive checks are powerful tools for detecting incorrect uses of a component by errant clients.

The central problem is this: when assertion checks are embedded directly within object methods, either one must pay some run-time penalty for the checks, even when they are not enabled, or one must recompile the methods without checks to eliminate this overhead. For the original developer, either option is available. For later reusers, unless source code is available, they must live with the choices of the

original developer (who typically removes checks).

Component-based development and reuse of commercially or publicly distributed software parts is rapidly becoming the norm. This article proposes a different approach to providing run-time precondition, postcondition, and invariant checks using *violation checking wrappers*. Rather than placing checking code inside the component, such a wrapper isolates checks in a separate layer between the component and its client(s). This approach, which is more in-line with object-oriented design, addresses the shortcomings of including the checks in the component itself and delivers the following benefits:

- Checking code is completely separated from the underlying component.
- Checks are completely transparent to both the client and the component.
- Checks are transformed into a separate component that can be distributed in compiled form along with the underlying component.
- Component reusers can easily control the insertion or removal of checks without requiring recompilation of the underlying component, the checking code, or the client code.
- When a wrapper is removed (to turn off checks), there is no run-time cost to the underlying component.
- The approach does not require a preprocessor, a separate tool, or a language extension, and will work in most object-oriented languages.

The remainder of the article describes the basics of violation checking wrappers, gives an example of their use, and discusses possible extensions and enhancements. In addition, the relationship of this approach to previous work is also presented.

2. Violation Checking Wrappers

2.1. Decorating with Checks

The basic idea at the heart of violation checking wrappers is simple: place the component under consideration in a wrapper, or decorator [Gamma95], that takes on the responsibility for performing defensive checks. To support the full breadth of design by contract, the wrappers presented here will support checks for preconditions, postconditions, and invariant assertions. Such a wrapper provides exactly the same syntactic interface as the component itself, and delegates the work involved in carrying out each operation to the component held inside. Figure 1 illustrates this simple idea.

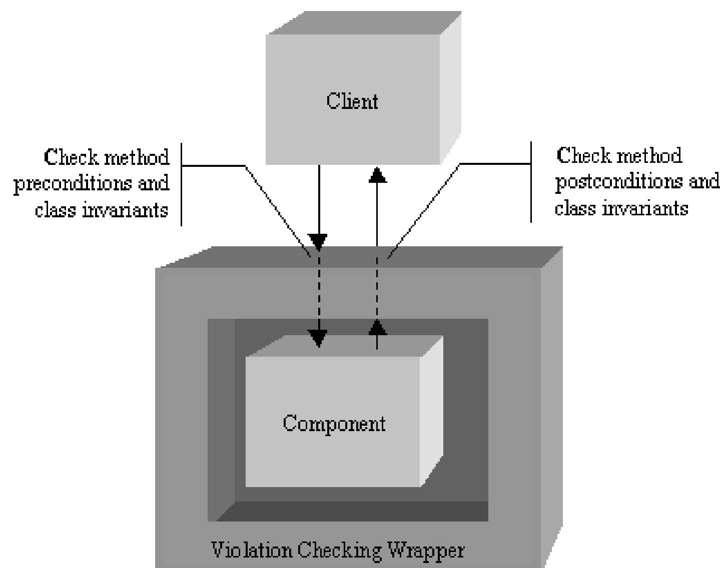


Figure 1. A wrapper surrounds the component, implementing all necessary assertion checking.

Placing checking code in a separate class or component is a simple idea, but it refocuses attention with dramatic results. Assertions are no longer embedded in the original implementation, making the original simpler and cleaner while alleviating fears that assertion code causes too much clutter. Further, when checks are promoted to the level of a separate artifact, it becomes easier for a programmer to write longer, more thorough checking code than will typically fit into an `assert` macro or procedure call. More thorough assertions lead to easier testing and better defect observability [Voas97].

Perhaps more importantly, however, placing checking code into a separate class elevates the checks from the level of individual statements inside a method up to the level of a useful component abstraction. Managing all of a component's checks as a separate class makes it easier

to insert or remove them together, and makes it easier to parameterize the entire group by other decisions. For example, instead of hard coding one action when a check fails--say, printing a message and terminating the program--one can introduce a separate notification function (or class) as a parameter to the checking wrapper. This notification action can be a template parameter in languages that support generic programming, or a run-time parameter in other languages.

If the run-time checks are separated from the underlying component implementation, how will those run-time checks be implemented? Allowing the checking code to have direct access to the internal data stored within the underlying component provides the closest match to directly embedded checks. This simple approach will be explored here; elsewhere, this approach has been compared to more sophisticated alternatives that do not require direct access to internal component state [Edwards98].

Implementing a direct access checking wrapper is straightforward: simply move any violation checks one would normally place inside the methods of the underlying component into a new class, and ensure that within this new class, all internal data members in the underlying implementation are visible.

Just as one can separate the checking code from the base component, one can also separate different kinds of checks into separate classes. Rather than providing a single class that performs all checking, the most obvious division is to provide three different classes: one to check invariant properties, one to check preconditions, and one to check postconditions. This division of labor allows one to easily select which checks to perform by adorning the underlying component with a different form of decorator.

2.2. The Client View

The technique proposed here will work in virtually any object-oriented setting, with components that range from individual classes in a programming language to more full-fledged CORBA, Java Beans, or COM components. To provide a concrete illustration, the examples in this paper will be presented in terms of Java classes for simplicity.

The violation detection wrapper strategy is simple and easy to use from the client's point of view. To use a component, one simply needs an interface (corresponding to the component's public contract) and a factory for creating new instances. For a Java class C, creating and manipulating objects is done in the normal way:

```
C my_c = CFactory.instance().newC( constructor-parameters );

my_c.aMethod( ... );
```

Since construction of new objects is handled through the factory pattern [Gamma95], decisions about whether or not to use wrappers (or which checks to enable) can be controlled and localized elsewhere. The example presented in Section 3 shows a simple way to provide programmatic control over factory preferences, although many other options (controlling wrapper choices via a separate initialization file, via a control panel user interface, or via a development tool's property sheet feature) are straightforward.

2.3. A Micro-architecture for Violation Detection

Given the simplicity of the client perspective when using wrapped components, what must the developer do to provide these capabilities? The most obvious answer is to start with a simple decorator pattern, as shown in Figure 2 (based on [Gamma95]). If one is working in Java, place the wrapper in the same package as the underlying component implementation and ensure that all data members are declared without any access modifier or as `protected`. This will give the wrapper visibility of the data members in the underlying implementation in order to carry out checks.

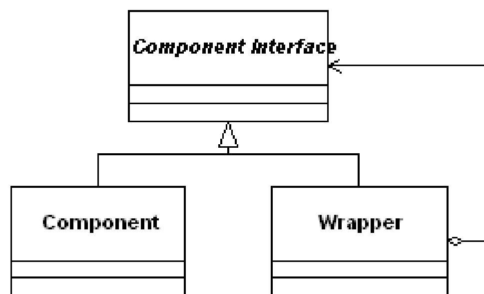


Figure 2. The basic structure of the decorator pattern.

Unfortunately, this simple arrangement has a number of limitations. First, it does not account for the factory services needed to create objects. Second, it does little to provide a uniform way to control what action(s) are taken in response to failed checks. Third, this approach does not immediately provide a way to selectively enable or disable certain classes of checks. Fourth, if the underlying component is subclassed later, it can difficult to reuse the wrapper for the parent class in creating the subclass later unless care is taken with the internal design of the wrapper class.

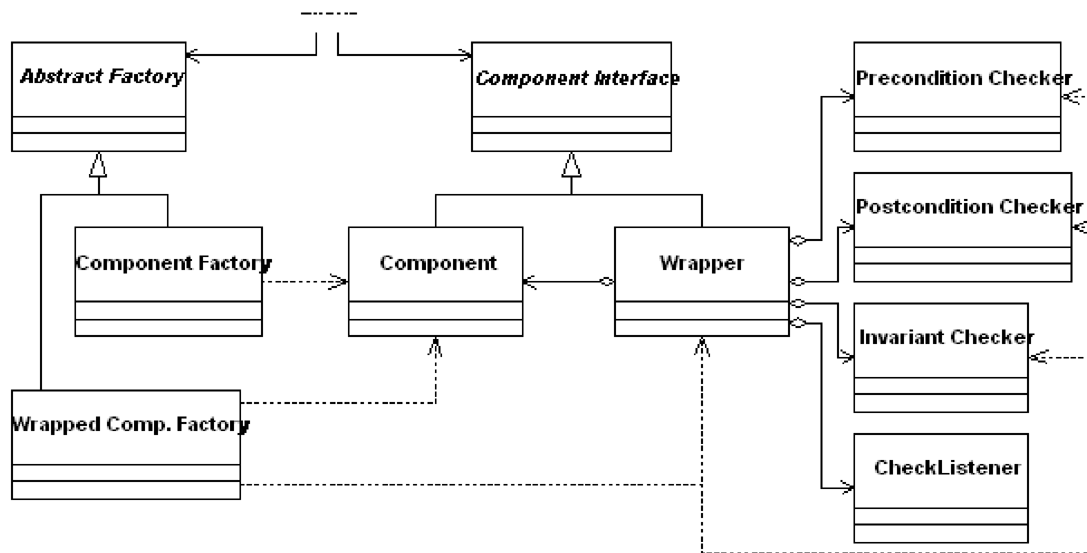


Figure 3. The violation detection wrapper micro-architecture.

To address these issues, Figure 3 presents a more detailed "micro-architecture" for checking wrappers. Only the public component interface and the factory class(es) are needed by client code. The center of Figure 3 consists of the core decorator pattern, with one slight modification. Here, the violation detection wrapper performs checks specific to the implementation of a particular concrete component, and thus wraps that concrete class rather than any instance of the public component interface. It is possible instead to mimic the decorator pattern exactly, providing one violation detection wrapper that performs checking for any implementation of the public component interface (given suitable supporting classes), but at the expense of additional type casts.

The abstract factory for creating new instances appears on the left of Figure 3. The factory implementations provide a way to create wrapped or unwrapped instances of the component for use by client code.

The right side of Figure 3 introduces several new classes that serve to partition the work performed by the wrapper. Instead of directly embedding the run-time checking code within the wrapper, these checks have been partitioned into three helper classes: one each to perform precondition, postcondition, and invariant checks. In addition, a new `CheckListener` interface has been introduced to serve as a point of separation between the checks being performed (the responsibility of the wrapper) and the action(s) to take based on the result of a check (the responsibility of the `CheckListener`).

The end result of this structural design is that the wrapper itself has a simple, regular structure that is directly driven by the component's public interface. Further, when one creates a subclass of the underlying component, subclasses of the three checking helper classes are easy to create and the same base wrapper can still be used. Finally, careful construction of the wrapper itself will allow a factory to create a wrapped object that carries any combination or subset of the three checking helper classes, in order to tailor the degree of checking provided.

3. A Simple Example

To serve as an example base component, Figure 4 presents a simplified interface for a Java `Vector`. This interface is inspired by the `java.util.Vector` class, but only includes a core set of methods for brevity and simplicity. A `Vector` embodies a growable array of objects that can be accessed by a numeric index. A `Vector` automatically increases in size to accommodate new elements as they are added. One item of a Vector's state is a capacity increment--that is, the amount to "grow by" when the object needs to resize to accommodate a newly inserted entry.

```
public interface Vector
{
    void   insertElementAt( Object obj, int index );
    Object elemetAt       ( int index );
    void   setElementAt   ( Object obj, int index );
    void   removeElementAt( int index );
    int    size();
    int    capacity();
    Object clone();
}
```

Figure 4. A simplified `Vector` interface.

Figure 5 sketches a possible implementation of this `Vector` interface using an array of `Object` references. The `VectorArrayImpl` class borrows heavily from the `java.util.Vector` implementation in Sun's Java SDK (with simplifications) and is meant to allow readers familiar with such vector implementations to apply their existing understanding in the context of this example. The implementation of only one method is shown for brevity.

```
class VectorArrayImpl implements Vector
{
    protected Object[] elementData;
    protected int      elementCount;
    protected int      capacityIncrement;

    public void insertElementAt(Object obj, int index )
    {
        ensureCapacityHelper( elementCount + 1 );
        System.arraycopy( elementData, index,
                          elementData, index + 1,
                          elementCount - index );
        elementData[index] = obj;
        elementCount++;
    }

    private void ensureCapacityHelper( int minCapacity )
    {
        ...
    }

    // ... more implementation code here ...
}
```

Figure 5. A simplified `Vector` implementation.

The `VectorArrayImpl` class has only three data members: `elementData` is an array of `Object` references; `elementCount` stores the number of objects currently contained in the `Vector` (the number of slots used in the array); and `capacityIncrement` determines how much additional space is requested when the `Vector` must grow. The `insertElementAt()` method shown in Figure 5 is typical of most `Vector` methods. This method inserts a new object at the given zero-indexed location in the `Vector`, "pushing" all objects at that index and beyond up by one to make room. It uses `ensureCapacityHelper()` (whose implementation is not shown) to resize the underlying array, if necessary.

3.1. A Vector Wrapper

Figure 6 illustrates the basic structure of a violation detection wrapper designed using this approach. The four data members in the wrapper maintain references to the wrapped concrete component and the three checking helper objects. The constructor of the wrapper is used to initialize these data members. Figure 6 only shows the implementation of one wrapped method: `insertElementAt()`. All other public methods are wrapped in exactly the same way.

```
class VectorArrayImpl_Wrapper implements Vector
{
    protected VectorArrayImpl            vector;
    protected VectorArrayImpl_PreChecks  pre_check;
    protected VectorArrayImpl_PostChecks post_check;
    protected VectorArrayImpl_InvCheck   check;

    protected VectorArrayImpl_Wrapper( VectorArrayImpl v,
                                       VectorArrayImpl_PreChecks  pre,
                                       VectorArrayImpl_PostChecks post,
                                       VectorArrayImpl_InvChecks  inv) {
        vector     = v;
        pre_check  = pre;
        post_check = post;
        check      = inv;
        // Check that invariant holds on (presumably newly constructed) vector
        if ( check != null )
        {
            check.invariant( vector );
        }
    }

    public void insertElementAt(Object obj, int index )
    {
```

```
        // Checks before the method is executed
        if ( check     != null ) check.invariant( vector );
        if ( pre_check != null ) pre_check.insertElementAt( vector, obj, index );

        // Save incoming values if needed for postcondition checking purposes
        VectorArrayImpl old_vector;
        Object          old_obj;
        if ( post_check != null )
        {
            old_vector = (VectorArrayImpl) vectore.clone();
            old_obj    = obj.clone();
        }

        // Delegate to wrapped object
        vector.insertElementAt( obj, index );

        // Checks after the method has completed
        if ( check      != null ) check.invariant( vector );
        if ( post_check != null ) post_check.insertElementAt(
                                      vector, old_vector, obj, old_obj, index );
    }

    // ... more wrapped methods here ...
}
```

Figure 6. A `VectorArrayImpl` wrapper.

The implementation of the wrapper's `insertElementAt()` method is divided into four sections. First, the wrapped vector's internal representation invariant and the method's precondition are checked by calling on the supporting helper objects. Note that calls to these helper objects are guarded within `if` statements. This allows the wrapper to be constructed with `null` references for one or more of the supporting helper objects, effectively "disabling" the corresponding checks within the wrapper.

Second, the wrapper saves the state of the wrapped object and any method parameters that might be modified by the underlying operation. This step is needed for full postcondition checking, since many postconditions describe an object's new state or a method's return value in terms of the "old" values at the time of method invocation. This work is also guarded by an `if` statement so that it is only performed when postcondition checks are enabled.

Third, the wrapper invokes the underlying method on the wrapped object. Fourth, the wrapped object's invariant is checked again and the corresponding postcondition is checked. Note that invariant checking always happens first, both in the first half of the operation before the wrapped object's underlying method is invoked, and also in the second half after that method returns. If the invariant were not checked first--before either the precondition or the postcondition--then it is possible that the other checks could be performed on a wrapped vector in an (erroneously) inconsistent state, causing the checking code to crash. Proper ordering of the calls within the wrapper methods aids in preventing such problems wherever possible.

3.2. The CheckListener

Figure 7 presents the `CheckListener` interface. A `CheckListener` provides two versions of a basic `notify()` method. Both versions take identifying information about the kind of check as well as the class and method in which checking is being performed. The `notify()` methods also take the Boolean result of the check performed, indicating whether the check was successful or not. This allows `CheckListeners` that trace or monitor all checking actions, rather than just failed checks. The second version of `notify()` also takes a message identifying the condition being checked (a string version of the boolean expression being tested, perhaps) for more informative presentation. While this `CheckListener` design is simple, additional enhancements or features are easy to add.

```
public interface CheckListener
{
    static final int PRECONDITION       = 1;
    static final int POSTCONDITION      = 2;
    static final int ABSTRACT_INVARIANT = 3;
    static final int CONCRETE_INVARIANT = 4;

    void notify( int     type,
                 String  classname,
                 String  method,
                 boolean result );
    void notify( int     type,
                 String  classname,
                 String  method,
                 boolean result,
                 String  condition_msg );
```

```
}
```

Figure 7. The notification interface for assertion failures.

The `CheckListener` interface makes it easy to construct classes that will respond to checks in different ways. It is simple to construct a `CheckListener` that throws a specific exception intended to halt the current program or thread when a check fails. Alternatively, messages about failed (or successful) checks can be directed to a log file, presented in a separate GUI window, or filtered and processed in any number of ways.

3.3. Checking Preconditions

The precondition checking helper class for the example vector implementation is shown in Figure 8. When created, this helper object receives a reference to the `CheckListener` it will notify for each check it performs. Other than the constructor, the precondition checker exports one method corresponding to each method in the `Vector` interface. Since one precondition checking object may be used to perform checks for many `VectorArrayImpl` objects, the `VectorArrayImpl` being checked is passed in as an additional parameter for each precondition check.

```
class VectorArrayImpl_PreChecks
{
    protected CheckListener listener;

    protected VectorArrayImpl_PreChecks( CheckListener cl )
    {
        listener = cl;
    }

    public void insertElementAt( VectorArrayImpl vector,
                                 Object          obj,
                                 int             index )
    {
        listener.notify(
                Check_Listener.PRECONDITION,
                "Vector_Array_Impl",
                "insertElementAt",
                index >= 0 &&
                index <= vector.elementCount
            );
    }

    // ... more precondition checking methods here ...
}
```

Figure 8. A precondition checking class.

Figure 8 only shows the implementation of one method in the precondition checking helper class. Other methods are similar. The precondition check for `insertElementAt()` is relatively simple. There are no requirements placed on the object being stored in the vector. The only restriction is that the given index is between zero and the size of the vector, including both end points.

3.4. Checking Postconditions

Figure 9 presents the postcondition checking helper class for the example vector implementation. As with the precondition checker, this helper object receives a reference to a `CheckListener` via its constructor. The postcondition checker also exports one method corresponding to each method in the `Vector` interface. For each incoming value that might be modified by the operation being checked, the postcondition checker takes in *two* values: one represents the "old" value on entry to the operation, while the other represents the "new" value after the operation has completed. This includes the component being operated upon--in this case, the wrapped vector.

```
class VectorArrayImpl_PostChecks
{
    protected CheckListener listener;

    protected VectorArrayImpl_PostChecks( CheckListener cl )
    {
        listener = cl;
    }

    public void insertElementAt( VectorArrayImpl new_vector,
                                 VectorArrayImpl old_vector,
                                 Object          new_obj,
```

```
                                Object          old_obj,
                                int             old_index )
    {
        // First, check the simple conditions
        boolean result =
            new_vector.elementCount == old_vector.elementCount + 1  &&
            new_obj == old_obj                                      &&
            new_vector.elementData[index] == old_obj;

        // Now check that the first segment of the vector is unchanged
        if ( result )
        {
            for ( int i = 0; i < index; i++ )
            {
                if ( new_vector.elementData[i] != old_vector.elementData[i] )
                {
                    result = false;
                    break;
                }
            }
        }

        // Finally, check that the second segment of the vector is unchanged
        if ( result )
        {
            for ( int i = index + 1; i < new_vector.elementCount; i++ )
            {
                if ( new_vector.elementData[i] != old_vector.elementData[i - 1] )
                {
                    result = false;
                    break;
                }
            }
        }

        listener.notify(
                Check_Listener.POSTCONDITION,
                "Vector_Array_Impl",
                "insertElementAt",
                result
            );
    }

    // ... more postcondition checking methods here ...
}
```

Figure 9. A postcondition checking class.

The postcondition checker's `insertElementAt()` method illustrates this convention. Both old (before insertion) and new (after insertion) values for the vector are passed in, as are both old and new values of the object being inserted. Because Java only supports one parameter mode--pass by value--parameters of scalar data types cannot be modified inside methods. As a result, only the old (incoming) value of the `index` is provided for checking. However, since all objects are represented by reference, it is not possible to tell from a method's signature whether or not the method's implementation might modify such a parameter (even if inadvertently or incorrectly). For Java, this means that old and new values for all objects involved in a method call must be available to check a postcondition properly. In other languages with richer parameter passing modes, it may be possible to use parameter mode declarations to determine when both old and new values must be available for proper postcondition checking.

For the purposes of illustration, Figure 9 shows a complete check of the postcondition for `insertElementAt()`. In addition to \checking the `elementCount` and `capacityIncrement` data members, the postcondition checker also checks every element within the valid region of the `elementData` array to ensure that the contents of the vector were modified properly (and that other cells of the array were not modified inappropriately). Such checking provides the best error detection capabilities when violation detection wrappers are used [Edwards00]. In practice, the component implementer may decide on the best tradeoff between the cost of performing such checks and the degree of error detection sought. Fortunately, with suitable modifications to the factory scheme, the violation detection wrapper approach would even support a component designer who wanted to provide both cheap (but less complete) checks and complete (but more expensive) checks, leaving the final decision up to the component integrator.

3.5. Checking Invariants

Implementing checks that correspond to the preconditions is all that is necessary to detect invalid uses of the component--where a client breaches the interface contract. One might assume that also checking the postconditions would be enough to detect errors in the component's implementation. It is true that this is a useful step in increasing the visibility of defects, and many bugs can be detected this

way. Checking postconditions alone, or even checking all specification-level conditions, does not guarantee that implementation-level defects will be spotted.

One can increase the defect-revealing capabilities of a violation checking wrapper by also checking implementation-level assertions [Edwards00], specifically *representation invariants*. An invariant property (or simply an *invariant*) for a class is some property that always holds true for newly created objects, and for any publicly exported method, if the property is true on entry it will also always be true upon completion of that method. There are two different varieties of class invariants: *abstract invariants* and *representation invariants* [Edwards97, Meyer97]. An abstract invariant captures properties of a class that are visible to the client; model-based specifications include such assertions. A representation invariant, on the other hand, captures properties of a class' internal representation--its private (or protected) data members. Representation invariants are not usually present in formal interface specifications, and are instead annotated within a component's implementation (if they are written down at all).

The invariant checking helper class follows a structure similar to that of the other checking helpers. Figure 10 shows the `VectorArrayImpl_InvChecker`. In this case, the checker is testing the representation invariant, instead of an abstract, client-visible invariant [Edwards97]. Since no formal behavioral description has been given for the Vector interface presented in Figure 4, it is not clear if the public contract includes an abstract invariant.

```
class VectorArrayImpl_InvCheck
{
    protected CheckListener listener;

    protected VectorArrayImpl_InvCheck( CheckListener cl )
    {
        listener = cl;
    }

    public void invariant( VectorArrayImpl vector, String method )
    {
        listener.notify(
                Check_Listener.CONCRETE_INVARIANT,
                "Vector_Array_Impl",
                method,
                   elementData       != null
                && elementCount      >= 0
                && elementCount      <= elementData.length
                && capacityIncrement >= 0
            );
    }
}
```

Figure 10. An invariant checking class.

In the case of the `VectorArrayImpl` class, the representation invariant captures mutual constraints between the data members: there must be an allocated array, the number of elements must be able to fit within this array, and the number of elements must be non-negative. Figure 10 also adds the condition that the `capacityIncrement` must be non-negative. This illustrates a design choice made by the `VectorArrayImpl` implementer. One can either require the capacityIncrement to be non-negative (and then assume it is so everywhere), or not require this property (and handle the case where it is not everywhere). The example `VectorArrayImpl` class here makes the former choice, while Sun's `java.util.Vector` implementation makes the latter.

An additional design choice that should be considered is whether or not the array slots at or past `elementCount` must contain `null` values, or whether they may refer to non-`null` objects. Because Java provides garbage collection, this design decision has relatively subtle consequences, possibly delaying reclamation of unreachable objects. In other languages, such a decision may be more important, since inconsistent treatment may lead to storage leaks. For the purposes of this example, the `VectorArrayImpl` class does not require the unused array slots to contain `null` values.

Figure 11 presents an alternative implementation of the invariant checking helper class. In this version, rather than combining all of the invariant conditions into one Boolean expression--and one `CheckListener` notification--each clause of the invariant condition is separated into a separate test. The alternate form of `CheckListener.notify()` is used so that a string describing the condition can be provided along with the result of the check. This version provides greater detail in its notification behavior, at the expense of less run-time efficiency.

```
class VectorArrayImpl_InvCheck
{
    protected CheckListener listener;

    protected VectorArrayImpl_InvCheck( CheckListener cl )
    {
        listener = cl;
    }
```

```
    public void invariant( VectorArrayImpl vector, String method )
    {
        listener.notify(
                Check_Listener.CONCRETE_INVARIANT,
                "Vector_Array_Impl",
                method,
                 elementData != null,
                "elementData != null"
            );
        listener.notify(
                Check_Listener.CONCRETE_INVARIANT,
                "Vector_Array_Impl",
                method,
                 elementCount >= 0,
                "elementCount >= 0"
            );
        listener.notify(
                Check_Listener.CONCRETE_INVARIANT,
                "Vector_Array_Impl",
                method,
                 elementCount <= elementData.length,
                "elementCount <= elementData.length"
            );
        listener.notify(
                Check_Listener.CONCRETE_INVARIANT,
                "Vector_Array_Impl",
                method,
                 capacityIncrement >= 0,
                "capacityIncrement >= 0"
            );
    }
}
```

Figure 11. An alternative invariant checking class.

3.6. Vector Factories

The idea of using a factory [Gamma95] to isolate client code from decisions about which particular concrete class is used when creating new vector objects is standard practice for many object-oriented developers. There are many common approaches to implementing such factories, often in conjunction with the singleton pattern. Any reasonable factory approach is viable here, whether decisions about which concrete classes to use are controlled by program statements, registry properties, a separate initialization file, or some other means.

```
public abstract class VectorFactory
{
    static private VectorFactory factory  = null;

    protected VectorFactory() {}

    static public instance()
    {
        if ( factory == null )
        {
            registerNewFactory( new VectorArrayImplFactory );
        }
        return factory;
    }

    abstract public Vector newVector();

    protected void registerNewFactory( VectorFactory vf )
    {
        factory = vf;
    }
}


public class VectorArrayImplFactory extends VectorFactory
{
    protected VectorArrayImplFactory() {}
```

```
    public Vector newVector()
    {
        return new VectorArrayImpl;
    }

    static public use()
    {
        registerNewFactory( new VectorArrayImplFactory );
    }
}


public class CheckingVectorFactory extends VectorFactory
{
    static private VectorArrayImpl_PreChecks  pre_checks;
    static private VectorArrayImpl_PostChecks post_checks;
    static private VectorArrayImpl_InvCheck   inv_check;

    protected CheckingVectorFactory( Check_Listener listener,
                                     boolean        check_inv,
                                     boolean        check_post)
    {
        listener = cl;
        pre_checks = new VectorArrayImpl_PreChecks( listener );
        post_checks = check_post ?
            new VectorArrayImpl_PostChecks( listener ) : null;
        inv_check = check_inv ?
            new VectorArrayImpl_InvCheck( listener ) : null;
    }

    public Vector newVector()
    {
        return new VectorArrayImpl_Wrapper( pre_checks, post_checks, inv_check );
    }

    static public use( Check_Listener cl, boolean post_check, boolean inv_check )
    {
        registerNewFactory( new CheckingVectorFactory );
    }
}
```

Figure 12. Factory support classes for creating `Vectors`.

Figure 12 shows one possible factory arrangement for creating vectors in the context of this example. The `VectorFactory` base class serves as a container for the singleton vector factory used to create new vector objects. This base provides the primary interface for client code that needs to create vectors:

```
    Vector my_vector =VectorFactory.instance().newVector();
```

Figure 12 also shows two concrete subclasses of `VectorFactory`. The `VectorArrayImplFactory` class creates "normal," unwrapped `VectorArrayImpl` objects. This is the "default" concrete factory used in the absence of any other actions in this example, although the component implementer has control over this choice.

The second concrete factory is the `CheckingVectorFactory`. As indicated by its name, this concrete factory creates wrapped vector implementations that perform run-time contract violation checking. The concrete factory subclasses provide a static `use()` method that can be called to "replace" the singleton instance currently being used to create new vectors with a new instance of the selected concrete factory. For example, to request unwrapped vectors in a given program, the following statement could be used in the `main()` method:

```
    VectorArrayImplFactory.use();
```

The static `use()` method provided by the `CheckingVectorFactory` takes additional parameters to control the behavior of checking wrappers. When this concrete factory is selected, the client can choose which specific `CheckListener` will be used, and whether or not invariant checking and postcondition checking are enabled or disabled. For example, if `RaiseCheckException` is a `CheckListener` implementation that prints a diagnostic message and raises an exception when any check fails:

```
    CheckingVectorFactory.use(
            new RaiseCheckException(),
            true,  // enable postcond. checks
```

```
            true   // enable invariant checks
        );
```

4. Issues in Using Wrappers

The violation detection wrapper strategy presented here provides a different approach to packaging and managing run-time assertions with a number of benefits. For programmers used to using conventional assertions, the wrapper approach also raises a number of questions. The wrapper approach presented here has been employed at earlier points during its evolution in the commercial development of a family of software products for over seven years [Hollingsworth00]. This collection of applications consists of approximately 250 components implemented in more than 100,000 lines of C++ code. The developers have cited the practice of using assertion-checking wrappers during unit, integration, and system testing as an important mechanism in achieving a high level of quality across all products in this application family. Despite such practical success, however, there are issues worthy of discussion in considering the wrapper-based approach.

The first critical observation about the approach is that it separates the assertion checks from the source code being checked. This is no surprise, since part of the motivation for the approach was the desire to separate checks from the underlying code. Unfortunately, this separation has negative consequences when the same information (how internal data members are managed, for example) is encoded in two different locations. Further, since many developers use assertions as a form of documentation as well as a run-time aid, removing assertions from the underlying code substantially reduces their utility in this direction.

The second critical observation is that the approach introduces extra development overhead to create and maintain the extra support classes needed. If practiced by hand, this strategy appears to provide only marginal benefits to the original component developer, while incurring at least some extra cost.

Both of these concerns are valid. Also, both are issues raised from the perspective of the original component developer--the only party to receive direct benefits from the presence of more conventional embedded assertions. The goal here, however, is to provide an effective way for the clients of reusable components (typically distributed in compiled form) to receive client-level benefits from run-time assertions, something they cannot easily obtain without having source code access under traditional approaches. Also, it is important to note that these concerns also focus on the textual location and management of assertions, rather than on the way checks are executed at run-time or the way they are distributed along with release versions of components to clients.

With careful planning, it is possible to achieve the best of both worlds: the benefits of behavioral contract assertions included in the original source code for locality of management and documentation, together with the wrapper-based deployment strategy. One approach is to adapt an existing behavioral contract framework, such as the Java Modeling Language (JML) [Leavens99]. JML allows one to use structured comments to provide a formal description of the behavior of a Java class directly embedded in the class source code. JML also provides the ability to automatically include run-time checking of such contract assertions (in most cases) in a manner similar to Eiffel. As has been noted before, this has the drawback of requiring source code in order to change decisions about the inclusion or exclusion of checks. The next research goal for this work is to extend JML to generate checking code in separate violation checking wrapper components. The majority of the wrapper support code discussed in Section 3 can be easily generated from the underlying component in a syntax-directed fashion--the only portion of the code requiring any creativity is the code in the checking helper classes that actually implements the assertions. Since JML already tackles the task of providing such checks (but in-line), transforming JML to place this code in separate wrapper components is a natural ideal. This provides a feasible solution to addressing the two biggest concerns about the wrapper strategy while still providing its benefits.

Another important issue with respect to assertion checking wrappers is the use of "internal" assertions within a method. The wrapper strategy only addresses client-visible, contract-level assertions rather than intermediate assertions about proper functioning within individual operations. However, developers are the primary users of such internal assertions, which are most helpful during development, testing, and later internal maintenance. Because such assertions do not necessarily have any relationship to the externally visible object state or to client-visible behavior, it is much less likely that they will be useful to the reusers of a fully-tested, released component implementation. As a result, it is likely that embedded assertions would still be used by the developer of a component before release, but be removed from released code, as is common practice. For similar reasons, assertions on non-public methods (usually helper methods) are not handled by wrappers. Since they are not visible to clients and are not part of the public contract, any *interface* violation detection scheme will not address them directly. Instead, assertions on such hidden, internal features are most appropriate during development, rather than after component release.

Another issue to consider is the fact that opening up access to the data members inside the wrapped class can be considered to violation encapsulation. For many developers, this is not a major issue. Elsewhere, other approaches to writing contract violation checks have been presented that address this concern [Edwards98]. Similarly, it is possible to write assertion checks in abstract, client-level terms rather than in the low-level terms of the wrapped component's internal implementation [Edwards98], although a full discussion of this approach is outside the scope of this article.

Finally, it is worth considering whether to use aggregation or inheritance to "wrap" checks around the methods of a class. Rather than using a separate wrapper class containing a reference to a `VectorArrayImpl`, for example, why not just create a subclass of `VectorArrayImpl` that overrides all public methods with implementations similar to those in Figure 6? Why not go further, and simply include the assertion checks directly in-line in such a class, rather than place them in separate helper classes? When only one concrete vector class is considered in isolation, these alternatives appear equally feasible. However, consider a family of concrete vector subclasses. In this case, if one creates a "checking wrapper" as a subclass of one of these, it becomes infeasible to share the checking code or the wrapping code with other vector subclasses. Further, if checks are included in-line in the wrapper class, it is difficult to extend the checking code, since adding additional precondition checks after the existing invariant and precondition checks (i.e., in the *middle* of the method shown in Figure 6) is not easy. With the proposed violation checking wrapper class arrangement shown in Figure 3, an entire tree of vector subclasses can be handled by creating a tree of each kind of checking helper class. Each checking helper subclass would only have to add

checks for the extra conditions required by its corresponding vector implementation. With care, a single wrapper could be used to manage the entire tree of vector implementations.

5. Related Work

The approach described here shares the same philosophy presented by Bertrand Meyer in describing design by contract [Meyer97]. Eiffel implements Meyer's strategy for run-time checking of contracts. Eiffel's assertion checking mechanism differs from the wrapper approach described here in several ways. First, assertions are embedded directly within the underlying component by the compiler, rather than being placed in a separate wrapper. Second, Eiffel provides a single Boolean expression slot for each assertion, which tends to discourage the inclusion of longer or more difficult to phrase constraints. Although one can always add new features to a class to implement such checks while keeping the assertions themselves small, this leads directly into serious disadvantages caused when potentially faulty methods on the wrapped object are used to check the health and conditions of that object.

JML [Leavens99, Leavens01] is similar to Eiffel in that assertions are directly embedded in component source and run-time checks of such assertions can be compiled into the corresponding component implementation. Unlike Eiffel, however, JML provides explicit separation between the abstract level of behavioral specification and the concrete details in a specific implementation. Assertions in JML can be phrased in abstract, client-level terms rather than in terms of implementation features.

Interestingly, Sun Microsystems has added an `assert` facility to Java 2 Platform version 1.4 [JDK1.4]. This facility is similar in spirit to `assert` macros in C and C++: the programmer can include an arbitrary Boolean expression in an assert statement. By using command line options, the developer can instruct the Java compiler to either insert code for checking assertions or to ignore assertions. As an added twist, when assertion checking code has been generated, it can be enabled or disabled through the Java Virtual Machine, either on the command line or through run-time method calls. This approach comes much closer to supporting assertions for the component reuser, instead of just the original developer. The approach described here allows per-object decisions about enabling checks, however, and also provides a clear evolutionary path to behavioral descriptions written in abstract terms rather than in implementation terms. Further, Java's `asserts` are tied into the language's exception mechanism; raising an exception is the sole notification action supported when an assertion fails. Sun's advice to developers reflects common practice, which fails to consider the value of assertion checking to component reusers and integrators in a commercial component marketplace: "Typically, assertion checking is enabled during program development and testing and disabled for deployment" [JDK1.4].

Sun's approach is a significant advancement over the wide variety of existing assertion utility classes available, most of which have been inspired by C++ `assert` macros. The Mozilla project's `Assert` class is a solid, publicly available example of this approach [Mozilla]. Such a utility class normally provides support for a standard response to failed assertions--raising a specific exception, for example. Support is also usually provided for enabling or disabling this action. Because Java does not provide for conditional compilation, some overhead is introduced even when assertions are disabled; completely removing this overhead necessitates recompilation of the checked component's source code, and in many cases modification to that source code as well.

The assertion checking approach described by Rosenblum [Rosenblum95] using the Annotation Pre-Processor (APP) is also related. APP allows assertions to be embedded in C programs and then combines separately provided code for checking those assertions to produce a "checking" build. Assertions can be assigned to different levels of importance to provide a simple means of including or excluding selected groups of assertions. Unfortunately, APP is for procedural programming in C and does not directly address object-oriented concerns.

Finally, more recent work on AspectJ also allows checking code to be separated from both clients and components [Kiczales]. Using aspect-oriented programming, one would create a separate aspect containing checking code and then choose whether or not to weave this crosscutting decision into a non-checking implementation at build time. Because aspect-oriented programming nicely embodies the decision to interpose code adornments such as run-time contract checks in a separate layer between the client and the underlying component, in many ways it is philosophically close to the approach advocated here. Unfortunately, the build-time focus of AspectJ would typically require source code to work effectively and has not been extended to the tailoring of reusable components distributed in binary form.

The checking wrapper approach was initially described as part of a larger framework for run-time behavioral contract checking [Edwards98] that is also related to formal specification, specification-based testing, and parameterized programming. It is possible to extend the violation checking wrapper approach to serve as the cornerstone that enables a comprehensive specification-based testing strategy [Edwards01].

6. Conclusions

Violation checking wrappers are applicable anywhere defensive checks are appropriate in an object-oriented design. Checking wrappers facilitate unit testing, integration testing, and debugging. Their critical benefit is that they provide an easy way to manage and control the externally visible contract checking features of a software component without requiring one to have access to its source code. This strategy recognizes the benefits provided by contract checking, particularly when component-based development approaches using independently developed or commercially available software parts are used.

By separating run-time checks from the underlying implementation code and promoting them to the level of a separate class, the attention and concerns of the programmer are refocused. Instead of being concerned about cluttering the base component with checks or how to fit complex checks into simple Boolean tests, the programmer can concentrate on writing more thorough checks that stand on their own. Using the factory pattern makes it simple to insert or remove checks while all client code and base component code remains unchanged and requires no recompilation. This allows a natural transition from including full checks when a new component is acquired or a new combination of components is being integrated, to gradual removal of checks as component compositions become more trusted. Further, by splitting different groups of checks into separate classes, one can use the same techniques to manage preconditions or postconditions

separately.

For these reasons, the wrapper-based approach to separating run-time checks provides the most important benefits of prior approaches while avoiding all of the earlier disadvantages. To obtain the advantages of using checks for documentation and keeping them "near" the implementation, it is possible to modify existing run-time check generation tools, such as JML, to deploy generated checks using wrappers. Further work is needed in this direction to obtain the best benefits from the wrapper approach together with the benefits of more traditional embedded assertion techniques.

Acknowledgments

This research is funded in part by NSF grant CCR-0113181. I also gratefully acknowledge the contributions to this work provided by Bruce W. Weide, Murali Sitaraman, and Joseph Hollingsworth, all of who have helped shape the violation checking wrapper approach. Thanks also go to Roy Tan and John Zaloudek, who helped perform some of the code development for this article.

References

[Edwards97]
Edwards, S.H. Representation inheritance: A safe form of "white box" code inheritance. *IEEE Trans. Software Engineering*, Feb. 1997; 23(2): 83-92.

[Edwards00]
Edwards, S.H. Black-box testing using flowgraphs: An experimental assessment of effectiveness and automation potential. *Software Testing, Verification and Reliability*, Dec. 2000; 10(4): 249-262.

[Edwards01]
Edwards, S.H. A framework for practical, automated black-box testing of component-based software. *Software Testing, Verification and Reliability*, June 2001; 11(2).

[Edwards98]
Edwards, S., Shakir, G., Sitaraman, M., Weide, B.W., and Hollingsworth, J. A framework for detecting interface violations in component-based software. In *Proc. 5th Int'l Conf. Software Reuse*, IEEE CS Press: Los Alamitos, CA, 1998, pp. 46-55.

[Gamma95]
Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Hollingsworth00]
Hollingsworth, J.E., Blankenship, L., and Weide, B.W. Experience report: Using RESOLVE/C++ for commercial software. In *Proc. ACM SIGSOFT 8th Int'l Symposium on the Foundations of Software Engineering*, pp. 11-19, ACM, San Diego, CA, Nov. 2000.

[Kiczales]
Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W.G. Aspect-oriented programming with AspectJ. Available on-line at http://www.aspectj.org.

[Leavens99]
Leavens, G.T., Baker, A.L., and Ruby, C. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds (eds.), *Behavioral Specifications of Businesses and Systems*, Chapter 12. Kluwer, 1999, pp. 175-188.

[Leavens01]
Leavens, G.T., Baker, A.L., and Ruby, C. Preliminary design of JML: A behavioral interface specification language for Java. Dept. of Computer Science, Iowa State University, TR #98-06p, Aug. 2001. Available on-line: <http://www.cs.iastate.edu/~leavens/JML.html>.

[Meyer92]
Meyer, B. Applying "design by contract." *Computer*, Oct. 1992; 25(10): 40-51.

[Meyer97]
Meyer, B. *Object-Oriented Software Construction, 2nd Edition*. Prentice Hall PTR: Upper Saddle River, New Jersey, 1997.

[Mozilla]
Mozilla.org. Utility classes for Java code. Available on-line: <http://www.mozilla.org/projects/blackwood/java-util/>.

[Parnas72]
Parnas, D.L. A technique for software module specification with examples. *Comm. ACM*, May 1972, pp. 330-336.

[Rosenblum95]
Rosenblum, D.S. A practical approach to programming with assertions. *IEEE Trans. Software Eng.*, Jan. 1995; 21(1): 19-31.

[JDK1.4]
Sun Microsystems. Assertion facility. Section in "Java™ 2 SDK, Standard Edition, version 1.4 Summary of New Features and

Enhancements." Available on-line: <http://java.sun.com/j2se/1.4/docs/guide/lang/assert.html>.

[Voas97]
Voas, J.M. Quality time: How assertions can increase test effectiveness. *IEEE Software*, Feb. 1997; 14(2): 118-122.

[Wing90]
Wing, J.M. A specifier's introduction to formal methods. *IEEE Computer*, Sept. 1990; 29(9): 8-24.

Specification and Verification of Performance Correctness

Joan Krone
Dept. Math. and Comp. Science
Denison University
Granville, OH 43023, USA
+1 740-587-6484
krone@denison.edu

William F. Ogden
Dept. Comp. & Info. Science
The Ohio State Universtiy
Columbus, OH 43210, USA
+1 614-292-5813
ogden@cis.ohio-state.edu

Murali Sitaraman
Dept. Comp. Science
Clemson University
Clemson, SC 29634, USA
+1 864-656-3444
murali@cs.clemson.edu

Abstract [1]

Component-based software engineering is concerned with predictability in both functional and performance behavior, though most formal techniques have typically focused their attention on the former. The objective of this paper is to present specification-based proof rules compositional or modular verification of performance in addition to functionality, addressing both time and space constraints. The modularity of the system makes it possible to verify performance correctness of a module or procedure locally, relative to the procedure itself. The proposed rules can be automated and are intended to serve as part of a system of rules that accommodate a language sufficiently powerful to support component-based, object-oriented software.

Categories and Subject Descriptors

D.2.13 [Software Engineering]: formal specification and verification of software performance.

General Terms

Verification, assertive language, formal specifications.

Keywords

Procedure calls, proof rules, time and space, while loop rule.

1. INTRODUCTION

Predictability is a fundamental goal of all engineering, including software engineering. To show that a program predictably provides specified functional behavior, a variety of ways to apply a system of proof rules to a program for proving functional correctness have been studied since Hoare's work. More recent efforts address the special challenge of modular reasoning for object oriented, component based software [1, 5, 8, 9, 12]. These systems depend on programmer-supplied assertions that serve as formal specifications for the functional behavior of the software. While correct functional behavior is critical to any software system, in order to achieve full predictability, we must ultimately address the issue of performance as well.

A program that carries out the right job, but takes longer than available time to complete is of limited value, especially in modern embedded systems. Similarly, a program that is functionally correct, but that requires more space than the system can provide is not useful either. Cheng, Clemens, and Woodside note the importance of the performance problem in their guest editorial on *Software and Performance* [21]:

> "Performance is a problem in many software development projects and anecdotal evidence suggests that it is one of the principal reasons behind cases where projects fail totally. There is a disconnect between techniques being developed for software analysis and design and the techniques that are available for performance analysis."

Measurement during execution (e.g., using run-time monitoring) is a common approach for analyzing performance of large-scale systems [21]. The objective of this paper is to present static analysis (and hence, a priori prediction) as an alternative to measurement. In particular, the focus is on *modular* or *compositional performance reasoning*: Reasoning about the (functionality and performance) behavior of a system using the (functionality and performance) *specifications* of the components of the system, without a need to examine or otherwise analyze the implementations of those components [17].

Compositionality is essential for all analysis, including time and space analysis, to scale up. To facilitate compositional performance reasoning, we have introduced notations for performance specifications elsewhere [18]. Given functionality and performance specifications (and other internal assertions such as invariants), the rest of this paper describes a proof system for modular verification. Section II sets up the framework to facilitate automated application of rules, using a simple example rule. Section III contains proof rules for verification of procedure bodies and procedure calls, involving possibly generic

objects with abstract models as parameters. Section IV contains an example to illustrate a variety of issues involved in formal verification. Section V has a discussion of related work and summary.

2. ELEMENTS OF THE PROOF SYSTEM

Though the underlying principles presented in this paper are language-independent and are applicable to any assertive language that includes syntactic slots for specifications and internal assertions, to make the ideas concrete we use the RESOLVE notation [15, 16]. RESOLVE is intended for predictable component-based software engineering and it includes notations for writing specifications of generic components that permit multiple realizations (implementations) of those components. It also includes notations for specifying time and space behaviors of an implementation. The implementations include programmer-supplied representation invariants, loop invariants, progress metrics, and other assertions depending on the structure.

The proof rules have been designed so that an automated clause generator can start at the end of a given assertive program and back over the code replacing the executable language constructs with assertions about the mathematical domain over which the program has been written. The clause generator produces a clause that is equivalent to the correctness of the given program. The clause can then be evaluated manually, automatically by a theorem prover, or by a combination to determine whether the clause is provable in the appropriate mathematical domain, and thereby whether the program is correct (with respect to its specification). To illustrate the ideas, we begin with a simple example. First we consider functional behavior and then address performance for the following piece of assertive code:

Assume x = 3;
x := x + 1;
Confirm x = 4;

Exactly how such an assertive code comes into place, given a specification and an implementation, is explained in Section III. In this code segment, the programmer has supplied a pre-condition indicated by the **Assume** keyword and a post-condition following the keyword **Confirm** with some (assertive) code in between. To prove the correctness of this segment, consider the following automatable proof rule for expression assignment:

C \ Code; **Evaluate** (exp); **Confirm** Outcome_Exp[x ⇝ **M_Exp**(exp)]

C \ Code; x := exp; **Confirm** Outcome_Exp;

In this rule, *C* on the left side of both the hypothesis and the conclusion stands for *Context* and it denotes the collection of whatever information is needed about the code in order to reason about its correctness. For example, the types of variables and the mathematical theories on which those types are based would be in the context.

In our example, the Outcome_Exp is "x = 4." The Code preceding the assignment is the assertion "**Assume** x = 3." In the assertive clauses, the 3 and 4 are the mathematical integers, while the assignment statement is performing an increment on a computer representation of an integer. (The use of mathematical integers in specifying computational Integer operations is documented in *Integer_Template* that specifies Integer objects and operations, and it is assumed to be in the context.)

Applying the proof rule on the example leads to the following assertive code:

Assume x = 3; **Evaluate**(x + 1); **Confirm** x + 1 = 4.

This is the result of substituting the expression "x + 1" for x, the meaning of [x ⇝ **M_Exp**(exp)]. **M_Exp** denotes putting in the mathematical expression that corresponds to the programming expression, thus keeping our assertions over mathematical entities, rather than programming ones. The rule for **Evaluate** that causes the expression to be evaluated by the verifier is given at the end of this subsection. Similarly, the verifier would simply continue backing through the rest of the code, applying appropriate proof rules, eliminating one more constructs in each step.

Now we augment the above rule to prove functional correctness, with performance-related assertions. Suppose we need to prove the correctness of the following assertive code:

Assume x = 3 $\wedge$ **Cum_Dur** = 0 $\wedge$ **Prior_Max_Aug** = 0 $\wedge$ **Cur_Aug** = 0;
x = x + 1;
Confirm x = 4 $\wedge$ **Cum_Dur** + 0.0 = $D_{:=}$ + D_{Int} + $\wedge$ Max(**Prior_Max_Aug**, **Cur_Aug** + 0) $\leq$ $S_{:=}{}^{2}$;

Here, $D_{:=}$ denotes the duration for expression assignment [3] (excluding the time to evaluate the expression itself). $S_{:=}$ denotes storage space requirement for expression assignment (excluding the storage space needed to evaluate the expression itself and the storage for variable declaration of x which is outside the above code). The units for time and space are assumed to be consistent, though we make no assumptions about the units themselves. The rest of the terms (whose need may not become fully clear until after the discussion of procedures in Section III) are explained in the context of the following rule for expression assignment:

C \ Code; **Evaluate**(exp); **Confirm** (Outcome_Exp $\wedge$ **Cum_Dur** + $D_{:=}$ + Sqnt_Dur_Exp $\leq$ Dur_Bd_Exp $\wedge$

Max(**Prior_Max_Aug**, **Cur_Aug** + $S_{:=}$ + Fut_Sup_Disp_Exp) $\leq$ Aug_Bd_Exp)) [x $\rightsquigarrow$ **M_Exp**(exp)];

C \ Code; x := exp; **Confirm** Outcome_Exp $\wedge$ **Cum_Dur** + Sqnt_Dur_Exp $\leq$ Dur_Bd_Exp $\wedge$

Max(**Prior_Max_Aug**, **Cur_Aug** + Fut_Sup_Disp_Exp) $\leq$ Aug_Bd_Exp;

The new rule includes everything needed for functional correctness, and also includes new clauses about time and space performance. In spite of past attempts in the literature, it is just not possible to develop rules for performance correctness independently of functional correctness, because in general, performance depends on values of variables (which come from analyzing functional behavior) [17, 18]. In the example and in the rule, terms in bold print are keywords and the terms ending with "_Exp" represent expressions to be supplied by the programmer and kept up to date by the verifier.

First we consider timing. The keyword **Cum_Dur** suggests cumulative duration. At the beginning of a program the cumulative duration would be zero. As the program executes, the duration increases as each construct requires some amount of time to complete. The programmer supplies an over all duration bound expression, noted by Dur_Bd_Exp. This is some expression over variables of the program that indicates an amount of time acceptable for the completion of the program. As the verifier automatically steps backward through the code, that expression gets updated with proper variable substitutions as the proof rules indicate.

For example, in the above rule, when the verifier steps backward over an assignment, the variable, "x," receiving the assignment is replaced by the mathematical form of the given expression, "exp," in all of the expressions included within the parentheses.

Sqnt_Dur_Exp stands for the subsequent duration expression, an expression for how much time the program will take starting at this point. This expression is updated also automatically by the verifier, along with other expressions in the rule.

The duration (timing) for a program is clearly an accumulative value, i.e., each new construct simply adds additional duration to what was already present. On the other hand, storage space is not a simple additive quantity. As a program executes, the declaration of new variables will cause sudden, possibly sharp, increases in amount of space needed by the program. At the end of any given block, depending on memory management, storage space for variables local to the block, may be returned to some common storage facility, causing a possibly sharp decrease in space.

The right operation for duration is addition and for storage it turns out to be taking the maximum over any given block. It is reasonable to assume that for any given program, there will be a certain amount of space needed for getting the program started. This will include the program code itself, since the code will reside in memory. Assuming real, rather than virtual memory, the code will take up a fixed amount of space throughout the execution. With this in mind, we think of some fixed amount of space for any given program that remains in use throughout the execution. Our rules are written to deal with the space that augments the fixed storage and increases and decreases as the program executes. **Prior_Max_Aug** stands for "prior maximum augmentation" of space. At the beginning of any program, the prior maximum will be zero, since only the fixed storage is in use. As the program executes, over each block, a maximum of storage for that block is taken to be the **Prior_Max_Aug**. At any point in the program, there will be a storage amount over the fixed storage. We call that the current augmentation of space, **Cur_Aug**. Of course, there will be some overall storage bound to represent what is acceptable. We call that the augmentation bound expression, Aug_Bd_Exp. Finally, just as there was an expression to represent how much additional time would be needed, there is an expression for how much storage (displacement) will be needed in the future, the future supplementary displacement expression, Fut_Sup_Disp_Exp.

We conclude this section with a proof rule for Evaluation.

C \ Code; **Confirm** Fnl_Outcome_Exp $\wedge$ **Cum_Dur** + **Eval_Dur**(Exp) + Sqnt_Dur_Exp $\leq$ Dur_Bd_Exp $\wedge$

Max(**Prior_Max_Aug**, **Cur_Aug** + Max(**Eval_Sup_Disp**(Exp), Fut_Max_Sup_Exp)) $\leq$ Aug_Bd_Exp;

C \ Code; **Evaluate**(Exp) **Confirm** Outcome_Exp ∧ **Cum_Dur** + Sqnt_Dur_Exp ≤ Dur_Bd_Exp ∧

Max(**Prior_Max_Aug**, **Cur_Aug** + Fut_Sup_Disp_Exp) ≤ Aug_Bd_Exp;

3. PROCEDURES

We examine a more complicated procedure construct in this section, having introduced basic terminology using the expression assignment proof rule. We present a rule for procedure declarations and one for procedure calls. These rules apply not only to ordinary code when all variables and types are previously defined, but to generic code as well, i.e., code written for variables that have not yet been tied to a particular type or value. This capability to handle generic code is critical for reusable, object-based components.

3.1 Procedure Declaration Rule

Associated with every procedure is a heading that includes the name, the parameter list, and assertions that describe both functional and performance behavior:

P_Heading:
Operation P(**updates** x: T);
requires P_Usg_Exp ⟨ x ⟩;
ensures P_Rslt_Exp ⟨ x, #x ⟩;
duration Dur_Exp ⟨ x, #x ⟩;
manip_disp M_D_Exp ⟨ x, #x ⟩;

This heading is a formal specification for procedure P. We use separate keywords **Operation** to denote the specification and **Procedure** to denote executable code that purports to implement an operation. We have included only one parameter on the argument list, but of course, if there were more, they would be treated according to whatever parameter mode were to be indicated. The **updates** mode means that the variable is to be updated, i.e., possibly changed during execution.

In the heading, the type T may be a type already pinned down in the program elsewhere, or it might represent a generic type that remains abstract at this point. The **requires** and **ensures** clauses are pre and post conditions respectively for the behavior of the operation, and the angle brackets hold arguments on which the clauses might be dependent. Due to page constraints, the rule does not include other potential dependencies such as on global variables.

Details of performance specification are given in [18]. **Duration** is the keyword for timing. Dur_Exp is a programmer-supplied expression that describes how much time the procedure may take. That expression may be given in terms of other procedures that p calls and it may be phrased in terms of the variables that the operation is designed to affect. We may need to refer both to the incoming value of *x* and to the resulting value of x in these clauses. We distinguish them by using #x for the value of x at the beginning of the procedure and x as the updated value when the procedure has completed. The last part of the Operation heading involves storage specification. Here, **manip_disp** (termed **trans_disp** in [18]) suggests manipulation displacement, i.e., how much space the procedure may manipulate as it executes.

Given the operation heading, we next consider a rule for a procedure declaration to implement an operation.

C ∪ {P_Heading} \ **Assume** P_Usg_Exp ∧ **Cur_Dur** = 0.0 ∧
Prior_Max_Aug = **Cur_Aug** = **Disp**(x);
P_Body;
Confirm P_Rslt_Exp ∧ **Cur_Dur** + 0.0 ≤ Dur_Exp ∧

Max(**Prior_Max_Aug**, **Cur_Aug** + 0) ≤ M_D_Exp;

C ∪ {P_Heading} \ Code; **Confirm** Outcome_Exp;

C \ P_Heading; **Procedure** P_Body; **end** P;
Code; **Confirm** Outcome_Exp;

As in the assignment rule, C stands for the context in which the procedure occurs. Note that P_Heading, the specification of Operation P, is added to the context making it possible for reasoning about the procedure to take place. The conclusion

line of the rule allows the procedure declaration to be made and followed by some code and a clause to confirm after the code.

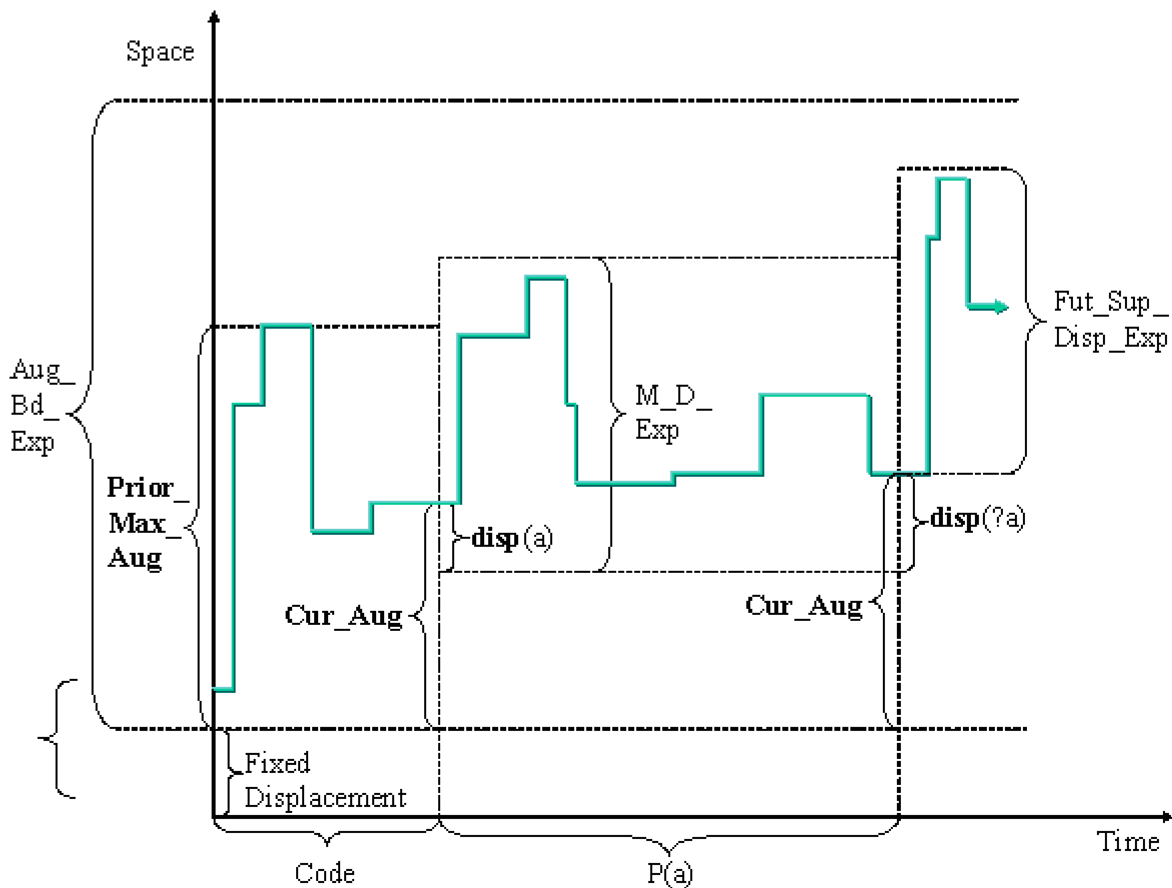
The hypotheses of the rule indicate that the procedure is to be examined abstractly, proving that no matter what value for the parameter is passed in, the result will satisfy both the functional and performance requirements.

The first hypothesis checks functional behavior by showing that if the **requires** clause is met, then the **ensures** clause is satisfied upon completion of the procedure body. For timing, we set the **Cum_Dur** to 0 thereby localizing the proof to just this procedure, avoiding the pitfall of having to consider the entire program when proving correctness for just this procedure. After the procedure body, we confirm that the **Cum_Dur** remains below Dur_Exp, the bound expression given in the specifications. It is assumed that the **Cum_Dur** acts like an auxiliary variable updated automatically at each step.

Finally, we address the storage requirements. Before the procedure body, we set the **Prior_Max_Aug** and the **Cur_Aug** both to be the amount of space required by the parameter, x. (Alternatively, the displacement of parameters at the beginning could be subtracted at the end.) This is necessary to retain the local nature of the proof process. The only concern that the procedure rule has about space is what the procedure uses above what has already been used in the past and what might be used in the future. After the body, the rule checks that the max over the stated values is within the specified bound.

3.2 Procedure Call Rule

A picture serves to motivate space-related assertions in the procedure call rule. The timing aspects of the rule are more straightforward and they are not shown in this picture.



Along the lower part of the picture the "fixed displacement" represents some amount of storage necessary for the program to run, an amount that does not vary throughout execution. The code itself is included in this fixed storage. Above the fixed storage the execution of the code requires a fluctuating amount of space, increasing when storage for new variables is allocated and decreasing when it is released.

The auxiliary variable, **Cur_Aug**, represents at any point what the current amount of storage is over and above the fixed storage. Note that the same variable appears twice on the picture,once at the place where a call to procedure P is made and

again at the point of completion of P. **Cur_Aug** has a value at every point in the program and is continually updated. Similarly, as the execution procedes, **Prior_Max_Aug** keeps track of the maximum storage used during any interval. In the picture at the point where the call P(a) is made, **Cur_Aug** is shown, as is **Prior_Max_Aug**. Of course, as the code execution progresses, the value for **Prior_Max_Aug** is updated whenever a new peak in storage use occurs.

Within the procedure body, some local variables may be declared. This augmented displacement is denoted in the figure by a spike in the line representing space allocation for the procedure code. The specifications of the procedure include M_D_Exp, an expression that limits the supplementary storage a procedure may use. The procedure must stay within that limit in order to be considered correct in terms of performance. As the picture shows, the M_D_Exp is an expression about only local variables and whatever parameters are passed in. These are the only variables under the control of the procedure and they are the only ones the procedure should need to consider for specification and verification purposes.

Disp is an operator that extracts the amount of storage for a given variable. This operator gets its value in the displacement clause given in an implementation of an object-oriented concept, and it is usually parameterized by the object's value [18]. At the point where the call P(a) is made the picture shows **Disp**(a), to denote that a's space allotment is part of the current augmentation displacement. Upon completion of the procedure call, the new value of a, shown as ?a may be different and may require a different amount of space from what its value needed at the time of the call. **Disp**(?a) is part of the current augmentation at the point of completion. Fut_Max_Sup_Exp, as noted before, describes a bound on the storage used by the remaining code, i.e., code following the current statement under consideration.

Given his explanation, the procedure call rule follows:

C $\cup$ {P_Heading} \ Code; **Confirm** P_Usg_Exp[x $\rightsquigarrow$ a] $\wedge$

$\forall$?a: **M_Exp**(T), **if** P_Rslt_Exp[#x $\rightsquigarrow$ a, x $\rightsquigarrow$?a] **then** Outcome_Exp[a $\rightsquigarrow$?a] $\wedge$

Cum_Dur + Dur_Exp[#x $\rightsquigarrow$ a, x $\rightsquigarrow$?a] + Sqnt_Dur_Exp[a $\rightsquigarrow$?a] $\leq$ Dur_Bd_Exp[a $\rightsquigarrow$?a] $\wedge$

Max(**Prior_Max_Aug**, **Cur_Aug**, Max(M_D_Exp[#x $\rightsquigarrow$ a, x $\rightsquigarrow$?a],

Disp(?a) + Fut_Sup_Disp_Exp[a $\rightsquigarrow$?a]) – **Disp**(a)) $\leq$ Aug _Bd_Exp[a $\rightsquigarrow$?a];

C $\cup$ {P_Heading} \ Code; P(a); **Confirm** Outcome_Exp $\wedge$ **Cum_Dur** + Sqnt_Dur_Exp $\leq$ Dur_Bd_Exp $\wedge$

Max(**Prior_Max_Aug**, **Cur_Aug** + Fut_Sup_Disp_Exp) $\leq$ Aug_Bd_Exp;

The heading for P is placed in the context, making available the specifications needed to carry out any proof. In the conclusion line, a call to p with parameter a is made at the point in the program following Code.

In modular reasoning, verification of this code that calls an operation P is based only on the specification of P. The functional behavior is addressed in the top line of the hypothesis part of the rule. To facilitate modular verification, at the point in the code where the call to P is made with parameter a, it is necessary to check that the **requires** clause, P_Usg_Exp with a replacing x holds. The second hypothesis, also about functional behavior, checks to see that if the procedure successfully completes, i.e., the **ensures** clause is met with the appropriate substitution of variables, then the assertion Outcome_Exp holds, again with the appropriate substitution of variables. These substitutions make it possible for the rules to talk about two distinct times, one at the point where a call to the procedure is made and one at the point of completion. The substitution of what variables need to appear at what points in the proof process avoids the need ever to introduce more than two points in the time line, thereby simplifying the process.

It is important to note here that the specification of Operation P may be relational, i.e., alternative outputs may result for the same input. Regardless of what value results for parameters after a call to P, the calling code must satisfy its obligations. This is the reason for the universal quantification of variable ?a in the rule.

The next hypothesis in the rule is about timing, and it checks, after variable substitution, that any result from the procedure will lead to satisfaction of specified time bounds for the client program. It is not surprising that any reasoning about time or space must be made in terms of the variables being manipulated, since their size and representation affect both.

Finally, the displacement hypothesis considers the maximum over several values. To understand this hypothesis, the picture helps by illustrating the prior maximum augmentation, current augmentation both at the point of the call and at the point of the return. The picture also shows the displacement for actual parameter a at the beginning of the procedure call and the

displacement of ?a at the end.

The displacement hypothesis involves a nested max situation. We consider the inner max first. Here we are taking the maximum over two items. The first is the expression from the procedure heading that identifies how much storage the procedure will need in terms of the local variables and the parameters. The second is the sum of the amount of space required by the final value of the updated parameter referred to as ?a and the amount of space for the rest of the program represented by Fut_Sup_Disp_Exp. From the second quantity we subtract the displacement of a, since it was accounted for in the current augmentation. Finally, we take the max over the two items and show that it remains within the overall bound.

The technique used in parameter passing naturally affects the performance behavior of a procedure call. In the rule, we have assumed a constant-time parameter passing method, such as swapping [3]. An additional degree of complication is introduced when an argument is repeated as a procedure call, because extra variables may be created to handle the situation. The present rule does not address this complexity

4. AN EXAMPLE

In this section, we present a more comprehensive example of a generic code segment, including appropriate expressions for describing time and space. In our example, we reproduce Stack_Template concept from [18], where a detailed explanation of the notation may be found:

```
Concept Stack_Template( type Entry;
                              evaluates Max_Depth: Integer);
          uses Std_Integer_Fac, String_Theory;
     requires Max_Depth > 0;

   Type_Family Stack  ⊆ Str(Entry);
     exemplar S;
     constraints |S| ≤ Max_Depth;
     initialization
          ensures S = Λ;

   Operation Push( alters E: Entry; updates S: Stack );
     requires |S| < Max_Depth;
     ensures S = ⟨#E⟩ ∘ #S;

   Operation Pop( replaces R: Entry; updates S: Stack );
     requires |S| > 0 ;
     ensures #S = ⟨R⟩ ∘ S;

   Operation Depth_of( restores S: Stack ): Integer;
     ensures Depth_of = ( |S| );

   Operation Rem_Capacity( restores S: Stack ): Integer;
     ensures Rem_Capacity = ( Max_Depth – |S| );

   Operation Clear( clears S: Stack );
end Stack_Template;
```

This specification is for a generic family of stacks whose entries are left to be supplied by clients and whose maximum depth is a parameter. It exports a family of stack types along with the typical operations on stacks. Any given stack type is modeled as a collection of strings over the given type *Entry* whose length is bounded by the *Max_Depth* parameter.

In order to promote both component reuse and the idea of multiple implementations for any given concept, our design guidelines include the recommendation that concepts should provide whatever operations are necessary to support whatever type is being exported and operations that allow a user to check whether or not a given operation should be called. In the stack example both *Push* and *Pop* must be present because those are the operations that define stack behavior. The *Depth_of*

and *Rem_Capacity* enable a client to find out whether or not it is alright to Push or to Pop. These are called primary operations.

Our guidelines suggest that secondary operations, ones that can be carried out -- efficiently -- using the primary ones, should be in an enhancement. An enhancement is a component that is written for a specific concept. It can use any of the exported types and operations provided in that concept. For example, we might write an enhancement to reverse a stack. In it would be an operation whose specifications indicate that whatever stack is passed into the procedure is supposed to be reversed. Given below is the functionality specification of such an enhancement:

Enhancement Flipping_Capability **for** Stack_Template;
  Operation Flip(**updates** S: Stack);
    ensures $S = \#S^{Rev}$;
end Flipping_Capability;

The advantage of writing this capability as an enhancement is that it is reusable, i.e., it will work for all Stack_Template realizations. For an example of a Stack_Template realization, a reader is referred to [18].

In our implementation, given below, we have included both the code (it is purely generic since any realization of the given stack concept may be used for the underlying stack type) and the performance specifications that deal with time and space.

Realization Obvious_F_C_Realiz **for** Stack_Template.Flipping_Capability;
  Duration Situation Normal: $\exists\ C_{Pu}, C_{Po}, C_{IE}, C_{EI}, C_{SIS}: \mathbb{R}^{>0} \ni$
    $C_{Pu} = \text{LUB}(\mathbf{Dur}_{Push}[\text{Entry}\times\text{Stack}])$ **and**
    $C_{Po} = \text{LUB}(\mathbf{Dur}_{Pop}[\text{Entry}\times\text{Stack}])$ **and**
    $C_{IE} = \text{LUB}(\mathbf{Dur}_{Is_Empty}[\text{Stack}])$ **and**
    $C_{EI} = \mathbf{Dur}_{Entry.Initialization}$ **and**
    $C_{SIS} + \text{Max_Depth} * C_{EI} = \mathbf{Dur}_{Stack.Initialization}$;
  Defn const $C_1: \mathbb{R}^{>0} = (C_{IE} + C_{Po} + C_{Pu})$;
  Defn const $C_2: \mathbb{R}^{>0} = (\mathbf{Dur}_{Call}(1) + C_{EI} + C_{SIS} + C_{IE} + C_{:=:})$;
  Defn const Cnts_Disp(S: Str(Entry)): $\mathbb{N}$ =
  $$\Big(\sum_{E:\,Entry} Occurs_Ct(E, S) * \mathbf{Disp}(E) \Big);$$

  Displacement Situation Normal: $\exists\ D_{SD}, D_{EID}: \mathbb{N} \ni$
    $D_{EID} = \mathbf{Disp}_{Entry.Init_Val}$ **and** $\forall$ S: Stack,
  $\mathbf{Disp}(S) = D_{SD} + D_{EID} * (\text{Max_Depth} - |S|) + \text{Cnts_Disp}(S)$
  and
    $\forall$ E: Entry, $\mathbf{Disp}(E) \geq D_{EID}$ **and**
  Is_Nominal($\mathbf{Mnp_Disp}_{Pop}$(E, S)) **and**
  Is_Nominal($\mathbf{Mnp_Disp}_{Push}$(E, S)) **and**
  Is_Nominal($\mathbf{Mnp_Disp}_{Is_Empty}$(S));

  Procedure Flip(**upd** S: Stack);
    duration Normal: $C_1 * |\#S| + \text{Max_Depth} * C_{EI} + C_2$;
    manip_disp Normal: $2 * D_{SD} + D_{EID} * (2 * \text{Max_Depth} + 1 - |@S|) + \text{Cnts_Disp}(@S)$;
    Var Next_Entry: Entry;
    Var S_Flipped: Stack;
    While $\neg$ Is_Empty(S)
      affecting S, S_Flipped, Next_Entry;
      maintaining $\#S = S_Flipped^{Rev} \circ S$ **and**
      Entry.**Is_Initial**(Next_Entry);
      decreasing $|S|$;

```
            elapsed_time Normal: C1*|S_Flipped|;
            max_manip_space 2*DSD + DEID*(2*Max_Depth
                                + 1 – |#S|) + Cnts_Disp( #S );
        do
                Pop( Next_Entry, S );
                Push( Next_Entry, S_Flipped );
        end;
        S :=: S_Flipped;
    end Flip;
end Obvious_F_C_Realiz;
```

In writing performance specifications, there is a trade-off between generality and simplicity. Given that the space/time usage of a call to every operation could depend on the input and outputs values of its parameters at the time of the call, a general version of performance specification can be quite complex. But we can simplify the situation, if we make some reasonable assumptions about the performance of reusable operations. While the performance specification language should be sufficiently expressive to handle all possibilities, in this paper, we present simplified performance expressions making a few assumptions. When the assumptions do not hold, the performance specifications do not apply.

There may a variety of ways in which time and space are handled, such as the straightforward allocation of space upon declaration and immediate return upon completion of a block as one method, and amortization as another. Here we use the term **Duration Situation** followed by Normal to indicate the former. A specification may also give performance behavior for more than one situation.

We provide constants that represent durations for each of the procedures that might be called, taking least upper bound when those durations might vary according to contents. For example, $\mathbf{Dur}_{Push}$ stands for the amount of time taken by a Push operation. Since that might vary depending on the particular value being pushed, the least upper bound is used to address that fact.

The way this approach allows the use of generic code is to have specifications that can be given in terms of the procedures they call. We think of initialization as a special procedure, one for each type, that is called when a variable is declared. For example, $\mathbf{Dur}_{Stack.Initialization}$ means the duration associated with the initialization of a stack. We do not know nor do we need to know what particular kind of stack will be used here, rather our specifications are completely generic, allowing the specific values to be filled in once a particular stack type has been designated.

All of the constants at the beginning of the realization are presented as convenience definitions so that the expressions written in the **duration** and **manip_disp** clauses will be shorter to read.

Just as we have identified what **duration** constants are needed for specifying the duration of the reversing procedure, we also set up definitions to make the storage (**manip_disp**) expression shorter to read. We can now see how the duration and manipulation displacement expressions associated with each procedure can be used when scaling up and using those procedures in a larger program.

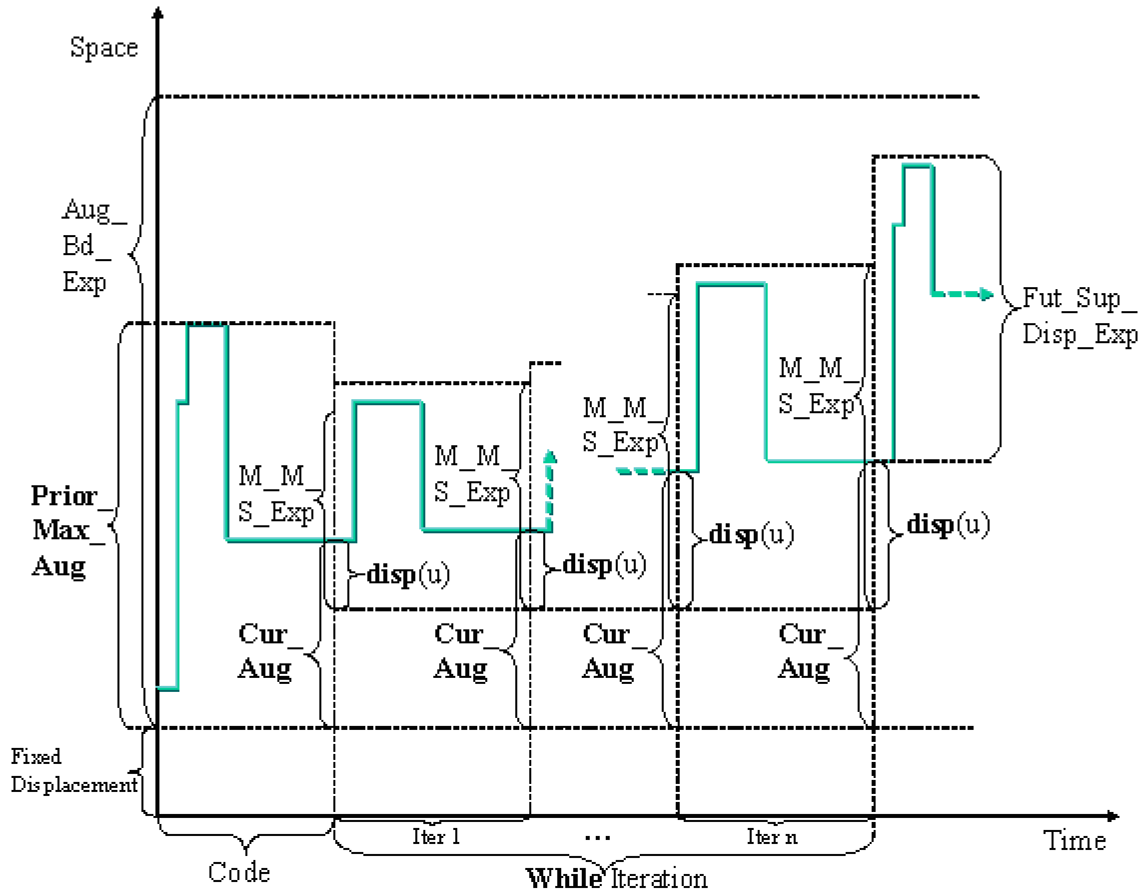
4.1 While Loop Rule

In verifying the correctness of the procedure, for the loop statement, the programmer supplies the following information:

- An **affecting** clause that lists variables that might be modified in the loop, allowing the verifier to assume that values of other variables in scope are invariant, i.e., not modified;
- A **maintaining** expression *Inv_Exp* that postulates an invariant for the loop;
- A **decreasing** expression *P_Exp* that serves as a progress metric to be used in showing that the loop terminates;
- An **elapsed time** expression *E_T_Exp* (for each situation assumption in the duration specification) to denote how much time has elapsed since the beginning of the loop; and
- A **max_manip_space** expression *M_M_S_Exp* that denotes the maximum space manipulated since the beginning of the loop in any iteration.

Developing a rule for **while** loops that will fit into our automated proof system and that will deal with both time and space presents some challenges different from those of procedures. Each procedure has a parameter list indicating which variables will be affected by the procedure body, and the parameter mode tells how they will be affected, but in the case of the loop, no such list exists. To compensate, we have an "**affecting**" keyword to be followed by a list of variables that may be changed by the body of the loop. There is a corresponding "**Change**" rule that takes care of making sure that the correct variable names are associated with each clause mentioned in the proof.

Also different from the procedure situation is the fact that a loop body may be executed multiple times, where a procedure body, unless recursive, is understood to execute only once. Our rule is written with two hypotheses, the first of which deals with each iteration of the loop, while the second hypothesis takes care of the case where the body of the loop is not executed because the Boolean expression controlling the loop, *B_Exp*, evaluates to *false*. A picture serves to illustrate the **while** construct rule.



In the first hypothesis of the rule, the variables following the **Change** keyword are those that may change during any iteration of the loop. All of the variables in bold are variables named by the proof system and present in all programs that deal with performance. **P_Val** is used to hold the value of *P_Exp,*, an expression supplied by a programmer to allow the proof system to check for loop termination. **P_Val** must decrease properly with each iteration.

Cum_Dur has a value that represents how much time has elapsed and increases with every step. **Prior_Max_Aug** holds the value of the maximum amount of space needed up to this point. This value is updated with each step, being replaced whenever a displacement larger than the current value occurs. **Cur_Aug** refers to the local displacement and may change at any step. We use the word "augmentation" to suggest amount of space beyond some fixed amount taken by the system on which this application is running.

Upon entering a **while** loop, the invariant, *Inv_Exp*, is assumed and the Boolean expression, *B_Exp*, is considered to be true having been evaluated as a mathematical expression. The current value of the progress metric, *P_Exp*, is assigned to the system variable, *P_Val*. The elapsed time expression, *E_T_Exp* is evaluated and assigned. To keep the proof concerns local, **Prior_Max_Aug** and **Cur_Aug** are set as the displacement of the variables that may change during any iteration, **Disp(u)**, in this case, since u has been designated as a **Change** variable. During any iteration, the body, *Body*, is executed, followed by an evaluation of the Boolean, *B_Exp*. Now we must show that the loop invariant is true and that the progress metric is strictly decreasing, hence establishing that the loop is progressing toward termination. We also must check that the amount of time taken remains within the specified allotment, *E_T_Exp*. With regard to space, we need to show that the maximum of what is currently being used and the prior maximum remains within the specified space allotment, *M_M_S_Exp*, maximum manipulation space for the local loop.

The second hypothesis deals with the situation that arises when the Boolean expression evaluates to false, thereby causing the loop body to be skipped. In this case it must be shown that whatever is supposed to be true upon completion of the loop

construct can be established by using the loop invariant and whatever other relevant clauses may be available in the assertive program from the code preceding the loop. Note that we take into consideration the evaluation of the Boolean expression that controls the loop. A rule for **Evaluate** follows.

Upon completion of the loop construct, regardless of whether there have been multiple iterations or none at all, it is necessary to check that the loop invariant is true and that the progress metric has a positive value, as does the maximum manipulation space expression.

Next we need to show that whatever value the changing variable(s), in this case *u*, may have, the loop invariant together with the negation of the Boolean expression are sufficient to establish the required functional outcome, *Fnl_Outcome_Exp*, and that the sum of the cumulative duration added to the elapsed time of the loop and the time it takes to evaluate the Boolean expression plus whatever time the code following the loop may require, *Sqnt_Dur_Exp*, will stay below the specified duration bound for the program, *Dur_Bd_Exp*.

Finally, we look at how we handle the proof that we have not used too much space. We take the maximum of three values and show that that maximum is less than the specified allotment of space. One of the values is the **Prior_Max_Aug**, the largest amount used up to this time. The second item we consider is the sum of the current augmentation, **Cur_Aug**, and the maximum allowed for the loop construct, *M_M_S_Exp*. The third value is found by using the expression that specifies the maximum that the code following the loop may take, Fut_Max_Sup_Exp, added to the difference between the displacement of *u* that it currently takes minus the displacement of the previous value of *u*.

```
C \ Code; Change u, P_Val, Cum_Dur, Prior_Max_Aug, Cur_Aug;
    Assume Inv_Exp ∧ M_Exp(B_Exp) ∧ P_Val = P_Exp ∧
 Cum_Dur = E_T_Exp ∧ Prior_Max_Aug = Cur_Aug = Disp(u);
                    Body; Evaluate(B_Exp);
    Confirm Inv_Exp ∧ P_Exp < P_Val ∧ Cum_Dur  ≤ E_T_Exp ∧
                    Max( Prior_Max_Aug, Cur_Aug ) ≤ M_M_S_Exp;

C \ Code; Evaluate(B_Exp); Confirm Inv_Exp ∧ E_T_Exp ≥ 0.0 ∧
M_M_S_Exp ≥ 0 ∧∀ ?u: M_Exp(T), ( if ¬ M_Exp(B_Exp) ∧ Inv_Exp
then ( Fnl_Outcome_Exp ∧ Cum_Dur + E_T_Exp + Eval_Dur( B_Exp )
+ Sqnt_Dur_Exp ≤ Dur_Bd_Exp) ∧
          Max( Prior_Max_Aug, Cur_Aug + Max(M_M_S_Exp,
            Fut_Max_Sup_Exp + Disp(u)) – Disp(??u) ) ≤ Aug_Bd_Exp )
                                          [ u ⇝ ?u][??u ⇝ u] );
____________________________________________________________
C \ Code; While B_Exp affecting u; maintaining Inv_Exp;
          decreasing P_Exp;
          elapsed_time E_T_Exp;
          max_manip_space M_M_S_Exp; do Body end;
Confirm Fnl_Outcome_Exp ∧
          Cum_Dur + Sqnt_Dur_Exp ≤ Dur_Bd_Exp ∧
          Max(Prior_Max_Aug, Cur_Aug + Fut_Max_Sup_Exp) ≤
                                          Aug_Bd_Exp;
```

In this short version of the paper, we have omitted discussion of several important issues. We have not explained how the system can accommodate dynamic and/or global memory management, though the framework allows for those complications. Finally, the non-trivial aspects of a framework within which to discuss the soundness and completeness of the proof system need to be presented.

5. RELATED WORK AND SUMMARY

The importance of performance considerations in component-based software engineering is well documented [7, 19, 20, 21]. Designers of languages and developers of object-based component libraries have considered alternative implementations providing performance trade-offs, including parameterization for performance [2]. While these and other advances in object-based computing continue to change the nature of programming languages, formal techniques for static performance analysis have restricted their attention to real-time and concurrency aspects [6, 10, 11, 20].

Hehner and Reddy are among the first to consider formalization of space (including dynamic allocation) [4, 13]. Reddy's work is essentially a precursor to the contents of this paper, and its focus is on performance specification. The proof system for time and (maximum) space analysis outlined in [4] is similar to the elements of our proof system given in section 2 of this paper. Both systems are intended for automation. In verification of recursive procedures and loops, we expect a time remaining clause to be supplied by a programmer, though the need for the clause is not made obvious in the examples in Hehner's paper. The main differences are that we address issues of generic data abstraction and specification-based modular performance reasoning. This becomes clear, for example, by observing the role of the displacement functions in the procedure call rule in Section 3.

This paper complements our earlier paper on performance specification in explaining how performance can be analyzed formally and in a modular fashion. To have an analytical method for performance prediction, i.e., to determine a priori if and when a system will fail due to space/time limits, is a basic need for predictable (software) engineering. Clearly, performance specification and analysis are complicated activities, even when compounding issues such as concurrency and compiler optimization are factored out. Bringing these results into practice will require considerable education and sophisticated tools. More importantly, current language and software design techniques that focus on functional flexibility need to be re-evaluated with attention to predictable performance.

ACKNOWLEDGMENTS

It is a pleasure to acknowledge the contributions of members of the Reusable Software Research Groups at Clemson University and The Ohio State University. We would especially like to thank Greg Kulczycki, A. L. N. Reddy, and Bruce Weide. We also gratefully acknowledge financial support from the National Science Foundation under grants CCR-0081596 and CCR-0113181, and from the Defense Advanced Research Projects Agency under project number DAAH04-96-1-0419 monitored by the U.S. Army Research Office.

REFERENCES

1. Ernst, G. W., Hookway, R. J., and Ogden, W. F., "Modular Verification of Data Abstractions with Shared Realizations", *IEEE Transactions on Software Engineering 20,* 4, April 1994, 288-307.

2. *Generic Programming*, eds. M. Jazayeri, R. G. K. Loos, and D. R. Musser, LNCS 1766, Springer, 2000.

3. Harms, D.E., and Weide, B.W., "Copying and Swapping: Influences on the Design of Reusable Software Components," *IEEE Transactions on Software Engineering,* Vol. 17, No. 5, May 1991, pp. 424-435.

4. Hehner, E. C. R., "Formalization of Time and Space," *Formal Aspects of Computing*, Springer-Verlag, 1999, pp. 6-18.

5. Heym, W.D. *Computer Program Verification: Improvements for Human Reasoning.* Ph.D. Dissertation, Department of Computer and Information Science, The Ohio State University, Columbus, OH, 1995.

6. Hooman, J., *Specification and Compositional Verification of Real-Time Systems*, LNCS 558, Springer-Verlag, New York, 1991.

7. Jones, R., Preface, *Proceedings of the International Symposium on Memory Management, ACM SIGPLAN Notices 34,* No. 3, March 1999, pp. iv-v.

8. Leavens, G., "Modular Specification and Verification of Object-Oriented Programs", *IEEE Software,* Vol. 8, No. 4, July 1991, pp. 72-80.

9. Leino, K. R. M., *Toward Reliable Modular Programs*, Ph. D. Thesis, California Institute of Technology, 1995.

10. Liu, Y. A. and Gomez, G., "Automatic Accurate Time-Bound Analysis for High-Level Languages," *Procs. ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, LNCS 1474, Springer-Verlag, 1998.

11. Lynch, N. and Vaandrager, F., "Forward and backward simulations-Part II: Timing-Based Systems," *Information and Computation*, 121(2), September 1995, 214-233.

12. Muller, P. and Poetzsch-Heffter, A., "Modular Specification and Verification Techniques for Object-Oriented Software Components," in *Foundations of Component-Based Systems*, Eds. G. T. Leavens and M. Sitaraman, Cambridge University Press, 2000.

13. Reddy, A. L. N., *Formalization of Storage Considerations in Software Design,* Ph.D. Dissertation, Department of Computer Science and Electrical Engineering, West Virginia University, Morgantown, WV, 1999.

14. Schmidt, H. W. and Chen, J. Reasoning About Concurrent Objects. In *Proceedings of the Asia-Pacific Software Engineering Conference*, IEEE, Brisbane, Australia, 1995, 86-95.

15. Sitaraman, M., and Weide, B.W., eds. Component-based software using RESOLVE. *ACM Software Eng. Notes 19,4* (1994), 21-67.

16. Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B. W., Long, T. J., Bucci, P., Heym, W., Pike, S., and Hollingsworth, J. E., "Reasoning About Software-Component Behavior," *Procs. Sixth Int. Conf. on Software Reuse*, LNCS 1844, Springer-Verlag, 2000, 266-283.

17. Sitaraman, M., "Compositional Performance Reasoning," *Procs. Fourth ICSE Workshop on Component-Based Software Engineering: Component-Certification and System Prediction*, Toronto, CA, May 2001.

18. Sitaraman, M., Krone, J., Kulczycki, G., Ogden, W. F., and Reddy, A. L. N., "Performance Specification of Software Components," *ACM SIGSOFT Symposium on Software Reuse*, May 2001.

19. Szyperski, *C., Component Software: Beyond Object-Oriented Programming,* Addison-Wesley, 1998.

20. Special issue on Real-Time Specification and Verification, *IEEE Trans. on Software Engineeering*, September 1992.

21. Special section: Workshop on Software and Performance, Eds., A. M. K. Cheng, P. Clemens, and M. Woodside, *IEEE Trans. on Software Engineeering*, November/December 2000.

Endnotes

1 An earlier version of this paper appeared in the Proceedings of the 2001 OOPSLA Workshop on Specification & Verification of Component-Based Systems.

2 We have added terms “+ 0.0” and “+ 0” in the expressions here so that it is easy to match the syntactic structure of the rule given next.

3 In RESOLVE, the right hand side of an assignment statement is restricted to be an expression. In particular, x := y is not allowed on variables of arbitrary types. For copying y to x, the assignment statement needs to be x := Replica(y). This interpretation is implicit for (easily) replicable objects such as Integers for programming convenience. This is what justifies the time analysis in the present rule. To move the value of y to x efficiently on all objects large and small, and without introducing aliasing, RESOLVE supports swapping (denoted by “:=:”) as the built-in data movement operation on all objects [3].

The Pragmatics of Integrative Programming

Larry Latour, Ling Huang, and Tom Wheeler
Software Engineering Laboratory, Dept. of Computer Science
University of Maine
222 Neville Hall
Orono, Maine 04469 USA

{larry.latour,ling.huang,tom.wheeler}.umit.maine.edu
Phone: +1 207 581 2260
Fax: +1 207 581 4977
URL: http://www.umcs.maine.edu/~pbrick

Abstract

We say that Interface "Pragmatics" are the unstated assumptions about the way a module interface and it's implementation is used, but that still must be understood in order for us to properly use that module. Such pragmatics can range from simple restrictions placed on the module to complex architectural assumptions about how the module can and must be used. Program integration is the process of combining modules in such a way that inconsistencies between module assumptions are avoided, and the use of pragmatic information can go a long way toward supporting this process. We look specifically at formalizing pragmatic information through the capture and analysis of patterns of use, first by using regular expressions to capture interface patterns, and then by using CSP (Communicating Sequential Processes) [Hoare85] to capture and analyze interaction patterns between using and providing modules.

Keywords

Program composition, program integration, interface pragmatics, regular expressions, CSP

Paper Category: technical paper
Emphasis: research

1. Introduction

Experience teaches us that the cornerstone of good software engineering practice is the concept of separation of concerns. This, along with it's companion support mechanisms of information hiding and modularization, drives the program development and understanding process, allowing us to manage the complexity of a large system. Each concern is encapsulated and modeled with an interface, and these interfaces are then combined into a working system using one of a variety of existing composition techniques.

Unfortunately, problems arise when systems are recomposed in this manner. We argue that many of these problems are due to the incomplete information typically provided by interfaces.

According to David Parnas [Parnas77], an interface is the set of "Assumptions that a using module makes of a providing module" and "a providing module makes of using module". But we need to analyze and elaborate such assumptions. They are not captured by just the types and accessing operations, as is usually taken for granted. There is more to it than that. The code that is written to deal with unstated assumptions of ordering, multiple module interactions, obligations, etc. of the operations shows that there is something else that is needed besides the operation definitions and the function of the module(s) being written. The goal of this research is to explore what this "something else" is.

If a module (or an object of an ADT) is thought of as defining a language then the thought pattern of "syntax, semantics, pragmatics" can be applied to module interfaces:

- Pragmatics Layer
- Syntax Layer
- Semantics Layer (Model)

Syntax is handled in programming languages by the structure of access routines. Semantics is the meaning of each of these access routines. But what are pragmatics? This concept is talked about with respect to natural languages, but never with respect to abstract module interfaces.

For our purposes pragmatics are the unstated assumptions about way an interface is used, but that still must be understood in order to properly use that interface. In natural language understanding, they are best shown by an example - "I'm beside myself". Pragmatics make things more natural and smooth in natural language, but actually seem to be necessary in formal language.

The following is a definition of Semantics and Pragmatics, taken from the Natural Language section of an online AI course [Cawsey94]:

> "… semantics and pragmatics, are concerned with getting at the meaning of a sentence. In the first stage (semantics) a partial representation of the meaning is obtained based on the possible syntactic structure(s) of the sentence, and on the meanings of the words in that sentence. In the second stage, the meaning is elaborated based on contextual and world knowledge."

Interface pragmatics can range from simple restrictions place on abstract data type operations to complex architectural assumptions about how an interface can and must be used.

2. The Problem

Often mismatch problems occur between modules, and the process of composing multiple modules can cause tangled code. A good case study illustrating some of these mismatches is described in [Garlen95]. In addition, design decisions that are usually thought of as being in the domain of the using module are actually dictated by the often unstated or poorly stated assumptions of the providing module.

3. The Position

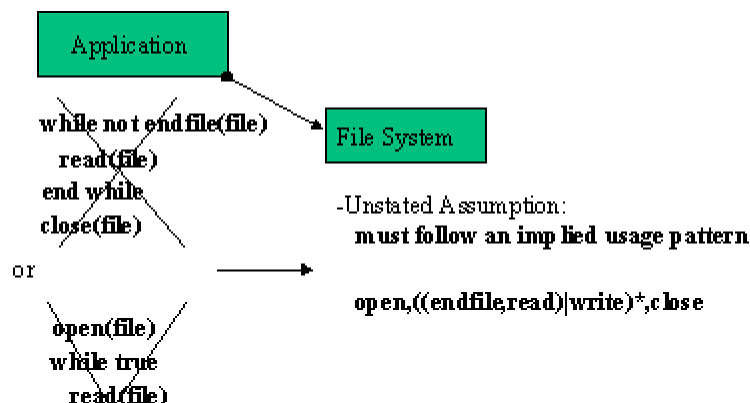
If we make an effort to enumerate and formalize unstated assumptions about the use of a module, we might be able to use the results to:

- better understand the meaning of a component's syntax, semantics, **and** pragmatics.
- better understand the process of system integration.
- better design and verify the correctness of the using module.
- better analyze and verify the correctness of the system.

4. Justification

We've developed a number of examples using both regular expressions and CSP (Communicating Sequential Processes) [Hoare85]. It seems that this technique shows promise for formalizing and analyzing patterns of use, and as a result provides more information to the user of a component about how to properly integrate that component into a system.

Let's begin with a simple file interface, with operations open, read, write, close, and endfile. As straightforward as this abstraction is, there are still ordering restrictions that must be obeyed in order for a using program to work properly. For example, typically open must be used before reading or writing, the file "must" be closed (either explicitly or implicitly), and endfile must be checked before reading. We can represent these ordering restrictions with a regular expression, and then show three simple using programs, one that matches the expression, and two that don't.



```
  if endfile (file) exit loop
 end while
 close(file)
or
 open(file)
  while not endfile(file)
    read(file)
  end while
close(file)
```

Figure 1

What we typically consider as the interface here is really a "low level" interface - the specification of each independent operation in the interface. What we're missing are the generic, "higher level" patterns of use of the combined operations, or what we'll call "high level" interfaces. These generic patterns can either be derived from restrictions based on the interface specification of the operations or on implied restrictions based upon an implementation of the operations. For example, in figure 1 above, the regular expression describing the pattern of use of the file operations can be derived as a theorem from the specifications of the file interface operations, whereas in other cases patterns of use might be semantically correct but more or less efficient dependent on the implementation choices of the operations.

Consider the following simple example of an iterator class:

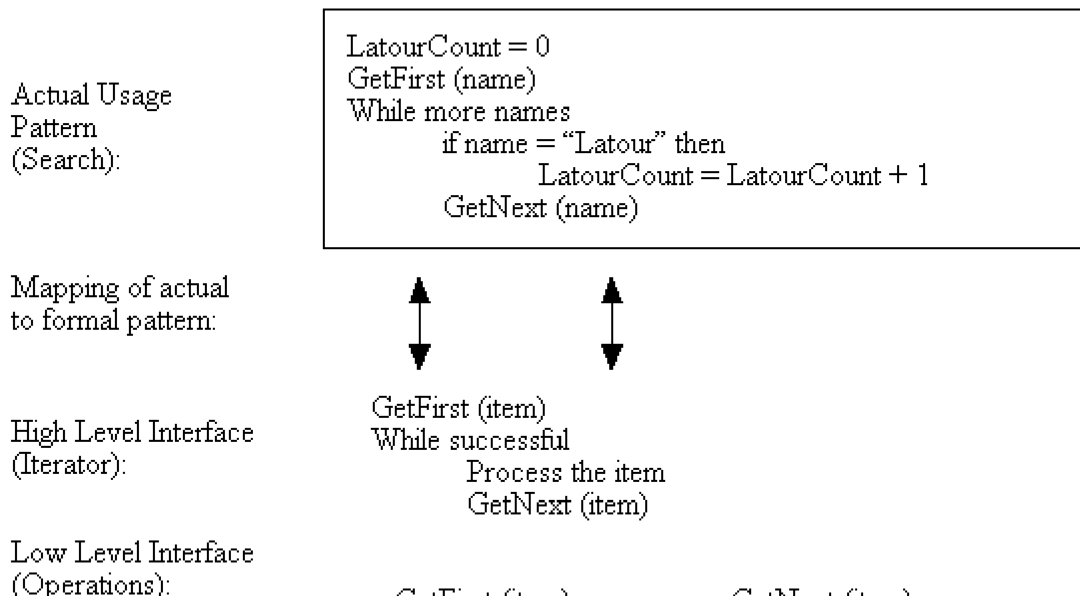
```
Class iterator
  GetFirst (item)
  GetNext (item)
```

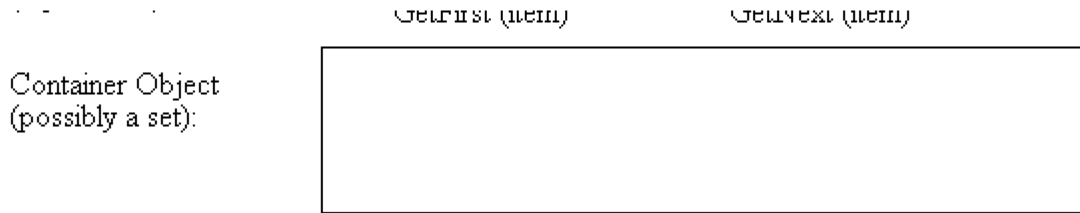
In the case of this iterator, the forced pattern of use, or the high level interface of the iterator, is:

```
GetFirst (item)
While successful
   Process the item
   GetNext (item)
```

It is these patterns of use that caused us to specifically partition the low level interface into getFirst and getNext to begin with.

While we typically say low level interfaces are **composed** into a higher level module, we might better say that high level interfaces are **integrated** into that higher level module. The term "integration" better suggests that a good deal of important activity is going on during the integration process. It is the instantiation and integration of high level interfaces that is the value added in the higher level module.





Module Low Level and High Level Interfaces

Figure 2

Part of the reason that programmers tend to practice bottom-up programming (as opposed to purely top-down functional decomposition) is that they have developed and refined skills for integrating high level interfaces into a blending of concerns in a using module. But a number of problems arise with higher level interface integration that work against the programmer:

1. There is no explicit intermediate step of instantiating a generic "formal" high level interface with an "actual" high level usage pattern. The programmer implicitly moves directly to an integrated pattern that instantiates two or more high level interfaces together in an integrated pattern. **Our Solution:** there needs to be both a formal language for describing a high level interface and an explicit instantiation of this interface to an actual usage pattern before the step of integrating high level patterns together

2. Many times usage patterns from high level interfaces "tangle" together in a way that leads to undue code complexity and subsequent confusion. Gregor Kiczales's Aspect Oriented Programming [Kiczales97] also addresses this type of problem. **Our Solution:** the tangling rules between usage patterns derived from high level interfaces need to be formalized and analyzed using an integration language.

3. Often high level interfaces are incompatible and need to be redesigned. Dave Garlen's Architectural Mismatch paper [Garlen95] discusses the ramifications of this. **Our solution:** this incompatibility needs to be expressed in terms of mutual exclusion restrictions between elements of the offending usage patterns, derived from offending underlying high level interfaces.

One of the mindsets that contribute to this problem is that we tend to view systems concerns on a single level. For example, if we separate concerns in a system using OO methods, we might come up with the decomposition in figure 3:

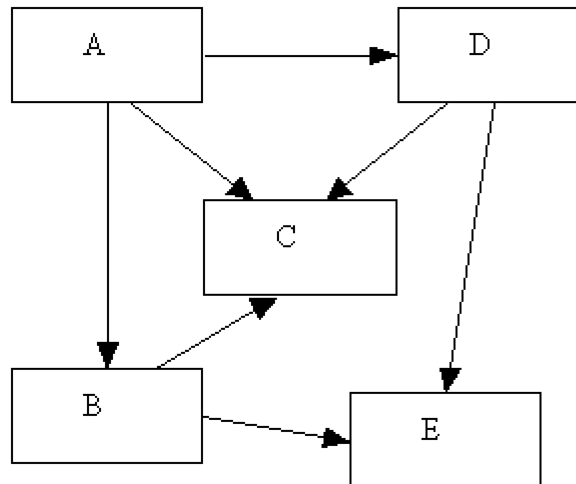
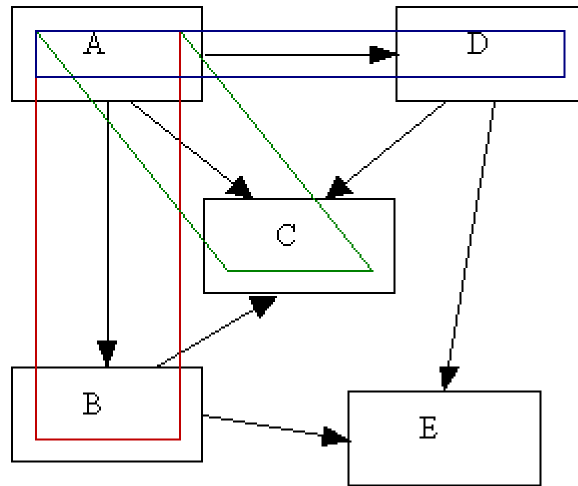


Figure 3

The arrows represent the "uses" relation, i.e., which objects call which operations of which other objects.

Unfortunately, a concern is not completely abstracted away in an object. It is replaced by not only its low level interface, but

more critically, its high level interface. A using module has to deal with the restrictions place on it by the high level interfaces of each of the objects it uses. The concerns of integrating these high level interfaces are meta-concerns, and tend to be more global/integral in nature. For example, consider the concerns of modules B,C, and D integrated in A in figure 4.



B (Red), C(Green), and D (Blue) Concerns Integrated in A

Figure 4

A more complex example is the module interface of a FIFO TCPIP network "Connection". Typically it only provides specifications of individual operations, duch as:

Socket: define the type of communication protocol
Bind : assign a name to a socket
Listen :notice of willingness to receive a connection
Accept : wait for an actual connection
Close : close a connection
Send : send data across a connection
Receive: receive data across a connection

The patterns of use /restrictions on the TCP/IP operations can be described informally as:

- Socket must be called before bind
- Bind must be called before listen
- If called, listen must be called before accept
- Accept must be called before send/recv
- Close must be called after accept and before next accept

From these informal sematics we can extract a regular expression into the high level interface of

Socket,Bind,[listen],(accept,(send|recv)*,close)*

All "legal" programs must match this regular expression pattern, including the following simple client program:

```
Int sockfd,newsockfd;
If (sockfd=socket(…)<0)
      err_sys("socket error");
If (bind(sockfd,5) <0)
     err_sys("socket error");
For(;;) {
      newsockfd=accept(sockfd,…);/*blocks*/
      if (newsockfd<0)
           err_sys("socket error");
```

doit(newsockfd); /* process request with sends/recieves */
close(newsockfd);}

Note that the actually using pattern of the TCP/IP operations in this case does match the regular expression. Note also that we used an informal argument to derive the regular expression pattern of use from the informal specifications of each of the TCP/IP operations. This is one of those cases where the pattern of use is a theorem that can be proven from the specification of the individual TCP/IP operations, and we can thus benefit from a more formal specification of these operations.

A number of efforts have been made to formalize these high level usage patterns of modules and their interconnections, and an interesting body of work of great interest to us is being done by Allen, Garlen, and Ivers [Allen98]. Specifically, we borrow a number of concepts from their work extending CSP (Communicating Sequential Processes) to deal with architectural patterns. Their resulting formal language, Wright, allows for the specification and analysis of module interactions by formalizing and verifying the interaction patterns.

We utilize their ideas in a slightly different way. Our premise is that we can use CSP processes to define both the formal and actual patterns of use of a module interface. We can then use formal techniques to verify that the "actual parameter" using pattern is consistent with the "formal parameter" providing pattern of the high level module interface. One can view the formal and actual usage patterns as two communicating sequential processes that must properly synchronize in order to make progress. We can also view the two usage patterns as formal expressions whose consistency might be checked with a formal tool such as FDR (Failures, Divergence, Refinement) [FDR97].

Comparison with Garlan's Wright Approach
Ex. TCP/IP Client-Server System

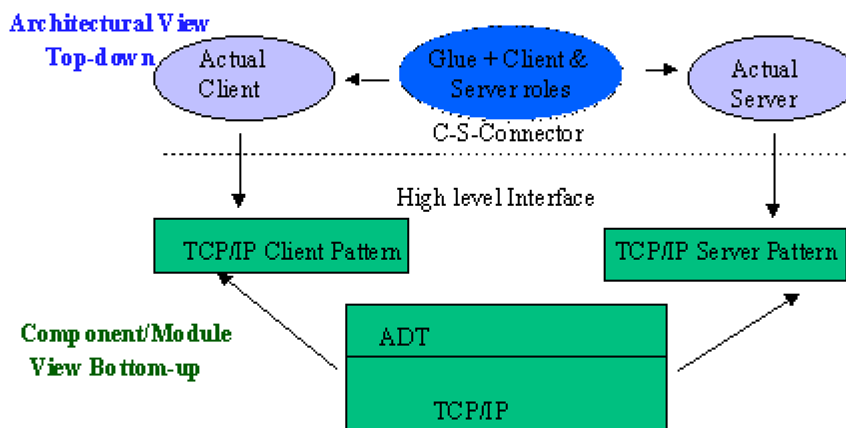


figure 5

From **figure 5** we can see that Allen, Garlen, and Ivers take an architectural point of view, defining client and server roles and then "connecting" them together with a glue event process, all written in CSP. Our view is to provide formal patterns of use in the high level interface of the TCP/IP module. In both cases the actual client and server are verified for correctness, but in our view correctness has already been established through the verification of the TCP/IP specification. Note that the TCP/IP module interface (**figure 5**) provides two patterns of use, one for a simple client and one for a simple server. These patterns are assumed to be correct when used by the actual client and server modules.

A simple server formal usage pattern might look like:

Server_Pattern= (CreatSocket→Bind→Listen→Session)
Session =(Accept→Interaction→Close→Session)
Interaction=(Receive→Send→Interaction)

The above CSP expressions can be thought of either as a single sequential process or as an event pattern, the way we would like to view them. The CSP process x -> P says "execute x and then behave like process P. So, the expression "Server_Pattern" begins by executing CreateSocket and then behaving like the remainder of the expression. The remaining expression in turn executes Bind and then behaves like Listen. When executed in tandem with another CSP process, like

operations must be executed together in order to collectively make progress. We will see how this plays out when the formal CSP event pattern is "executed" together with the actual pattern derived from the actual server program described earlier..

Here is what a simple client formal usage pattern might look like:

```
Client_Pattern= (CreatSocket→Session)
Session =(Open→Interaction→Close →Session)
Interaction=(Send→Receive→Interaction)
```

From the legal program we can derive an actual usage pattern in CSP:

```
Server_Actual_Program= Socket → Bind → ForLoop
ForLoop = Accept → DoIt → Close → ForLoop
DoIt = ...
```

Now, to verify that this actual pattern "matches" the formal client usage pattern, we symbolically execute the formal and actual CSP event patterns. If they are consistent then they will "synchronize" and both always "make progress".

Here is the synchronized "execution":

```
Combined_Pattern= CreatSocket(formal)/Socket(actual) →
    Bind(formal)/Bind(actual) → Listen(formal) → Session(formal)/ForLoop(actual)
Session(formal)/ForLoop(actual) = Accept(formal)/Accept(actual)
  → Interaction(formal)/DoIt(actual) → Close(formal)/Close(actual)
  → Session(formal)/ForLoop(actual)
```

As for our earlier example, the formal and actual server and client patterns can actually be proven as a theorem of the underlying "low level" TCP/IP interface. This is a subject for further study - the formulation and proof of higher level event patterns from interface specifications. But what about those event patterns that are not derived explicitly from interface restrictions, but that influence how the interface is designed or its implementation chosen? For example, when using graph algorithms we typically consider their relationship with the interface and implementation of their underlying graph abstract data type. In this case we can think of the algorithm and its underlying graph services existing symbiotically in a system. One is not thought of as "coming before" the other. We can actually say that the collection of application event patterns influence ones understanding of the underlying services they use. In this case, patterns of use of the using program, while they are not strictly part of the high level interface of the providing module, are part of the pragmatics of that module. In order to truly understand the meaning of the providing module, one needs to understand the collection of applications that cause it to be the way it is.

5. Related Work

The origins of this work come from a paper by Garlen, Allen, and Ockerbloom, "Architectural Mismatch: Why reuse is so hard" [Garlen95]. While that paper enumerates a number of fascinating issues concerning the mismatch between COTS components in a complex system development project, it ultimately raises more questions than answers. A very nice followup paper by Allen,Garlen, and Ivers, "Formal Modeling and Analysis of the HLA Component Integration Standard" [Allen98] gave us the idea of using CSP [Hoare85] to model the relationship between formal interface descriptions and actual interface usage. We borrowed their idea of using event patterns to model component connectors from a "top-down", architectural perspective in order to model interface obligations from a "bottom-up", component point of view. In both cases event patterns are used to verify the correct interconnection between to components, but in our case we're more interested in verifying that a using program uses the services of a providing module correctly.

In order to analyze the "correctness" of their module interconnections, Allen, Garlen, and Ivers use a CSP model checker called FDR [FDR297]. Although we haven't explored formal analysis to any great depth, this looks like a promising direction to explore.

6. Conclusion

Our premise is that the problem with wide-spread reuse of components (or the lack of it) is that unstated assumptions about the use of a component and its implementation often get in the way of system integration. Much as Garlen found in his "Architectural Mismatch..." paper [Garlen95], we see that unstated component assumptions, or pragmatics, are in many cases mismatched and need to be reimplemented or translated to solve the problem. Even if mismatch doesn't occur, many design decisions normally considered in the realm of the using program designer are actually obligations that are inherited from the modules used in that implementation - "often there ain't but more than one way to implement a module".

We propose that each component consist of both a low level, traditional, interface, and a high level interface consisting of those pragmatics that can be formalized and analyzed. Patterns of use are one of the formal objects of this high level interface. Furthermore, some of these patterns are properties of the traditional interface, whereas other patterns might be user level patterns that have influenced that traditional interface to provide the services it provides.

We looked at a number of formalisms for capturing interface patterns of use, specifically mentioning here regular expressions and CSP. CSP seems to be a good candidate for capturing patterns of use because (1) it is a formal notation, (2) it can be symbolically executed to show that a using and providing module are consistent, and (3) theorem provers can be used to verify correctness.

We would like to compare our notion of interface pragmatics to the techniques other researchers have used to try to formalize "higher level" architectural information, especially from the point of view of a component. It seems to us that the exploration of this "stuff in the ether floating above a component" is what will help us to get a better handle on system integration issues.

References

[Allen98]

R.J. Allen, D. Garlen, and J. Ivers. Formal Modeling and Analysis of the HLA Component Integration Standard. *Proceedings of of the Sixth International Symposium on the Foundations of Software Engineering (FSE-6),* 1998

[Cawsey94]

A. Cawsey. Databases and Artificial Intelligence 3 Artificial Intelligence Segment. Online course: http://www.cee.hw.ac.uk/~alison/ai3notes/section2_7_4.html 1994.

[FDR97]

Failures, Divergence, Refinement: FDR2 User Manual. Formal Systems (Europe) Ltd., Oxford, England, version 2.22 edition, October, 1997.

[Garlen95]

D. Garlen, R. Allen, and J. Ockerbloom. Architectural Mismatch: Why reuse is so hard. IEEE Software, November, 1995.

[Hoare85]

C.A.R. Hoare. Communicating Sequential Processes. Prentice Hall, 1985.

[Kiczales97]

G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J. Loingtier, J. Irwin. Aspect-Oriented Programming. Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997.

[Parnas77]

D.L. Parnas. Use of Abstract Interfaces in the Development of Software for Embedded Computer Systems, NRL Report 8047, 3 June 1977.

Design Issues Toward an Object Oriented RESOLVE

Roy Patrick Tan
Dept. of Computer Science
Virginia Tech
660 McBryde Hall, MS 0106
Blacksburg, VA 24061 USA

rtan@vt.edu

Abstract

Object-oriented programming is the most popular software development paradigm today. As such, the development of object- oriented support in RESOLVE may benefit the general programmer. As a first step, this paper surveys the various issues that must be considered in the creation of an object-oriented version of RESOLVE.

Keywords

Object-oriented design, RESOLVE, language issues

Paper Category: technical paper
Emphasis: research

1 Introduction

Object oriented programming is probably the most popular software development paradigm today. Although component-based, RESOLVE is not an object-oriented language. An object- oriented version of RESOLVE may benefit the programmer by giving a better mapping between the RESOLVE discipline and popular modern object-oriented languages, and making RESOLVE more accessible to those who are familiar with object-oriented software development.

To redesign RESOLVE as an object oriented language, careful consideration must be given to all issues regarding this transition. We need to explore issues surrounding questions such as:

- What are classes, and do they relate to types and modules in RESOLVE?
- Should an OO RESOLVE use reference or value semantics?
- What is inheritance, and how is it used?
- How should RESOLVE implement specification inheritance properly?
- How should an OO RESOLVE support implementation inheritance?
- What about multiple inheritance, should we support it fully, partially, or not at all?
- How will object-orientation affect type checking?
- How will exceptions be handled?

Fortunately, many people have traveled similar roads before, and we can learn their experiences. This paper presents a survey of pertinent issues that might arise in evolving RESOLVE to become an object-oriented language.

2. Classes, Types and Objects

2.1 What are Classes Anyway?

Classes are the primary level of data abstraction in mainstream object-oriented languages. In many object-oriented languages, classes are used to define two things at the same time: an encapsulation boundary (a module), and a type [Meyer98]. Typically, classes contain both the definition of a data structure, and functions or procedures (methods) related to the data structure. Thus a class defines a boundary for encapsulation -- similar to a RESOLVE module. Because every class typically defines only one data structure, the term class in most OO languages is also equivalent to the term type.

The addition of inheritance adds a new dimension: A class or type can be thought of either as one specific type, or as that specific type plus all other subtypes derived from it. Ada distinguishes between these two uses: a specific type T is what is properly called a "type" in Ada. In contrast, a "class" means "the union of all types in the tree of derived types rooted at T" [Wheeler97].

While RESOLVE currently does not fully support inheritance in the object-oriented sense, RESOLVE has in common with Eiffel and Ada a built in support for genericity. Parameterized types are often also called generic types (or generic classes), to distinguish them from regular types or classes.

2.2 Should RESOLVE Modules be Classes?

Because the class is the basic level of encapsulation in normal object-oriented languages, there is no distinction in most of such languages between type and module. That is, a class as viewed in a language like Java is like a RESOLVE module that exports only one type, and both module and type are called the same thing.

To implement OO-RESOLVE, we might want to follow the mainstream OO languages and restrict modules to export one type. In fact, in [Sitaraman94], it is already mentioned that every specification normally exports only one type, as it is usually possible to decouple two types into two separate components. However, it is also hinted that there are situations where having more than one type per component is preferable.

It is also possible to retain the module/type distinction, and allow multiple types (classes) to be defined within a module. Ada 95, for example, allows the definition of more than one type within a package.

2.3 Objects and Variables: Value or Reference?

Value versus reference semantics refer to what happens when one variable is assigned to another or is passed as a parameter. Scalar types in C and Java use value semantics assignments and parameter passing are always done by copying the contents of one variable to another. Copying is necessarily expensive when passing large amounts of data, so languages with value semantics like C may require the programmer to explicitly pass pointers instead.

Many object-oriented languages have reference semantics. Variables in a language with reference semantics hold only a reference to an object, rather than the object itself. In a sense, variables always contain pointers, and only the pointers get copied through assignment or parameter passing. With reference semantics, in a variableÕs lifetime, a variable can refer to more than one object. Another consequence of reference semantics is that two variables may refer to the same object at the same time.

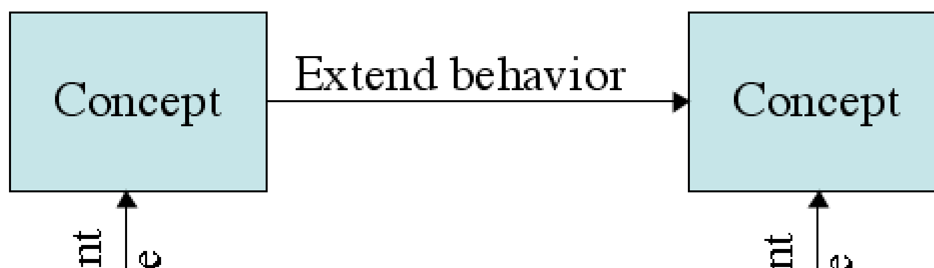
One unique feature of RESOLVE is that its semantics are neither value semantics nor reference semantics. The primary method of data-transfer in RESOLVE is swapping. Because swapping allows programmers to think in terms of value semantics regardless of the actual implementation, the use of swapping in RESOLVE solves a number of problems that assignment (both by reference and by value) has [Harms91].

A related issue is object identity. Some language experts believe that for a language to be fully object oriented, there must be a mechanism for determining between two variables whether they pertain to two different objects or not. In the case of RESOLVE, it shares an implicit mechanism with languages with value semantics: at any point in time, two different variables will always refer to two different objects. This is potentially a problem, since it is merging the concepts of name and identity [Khoshafian86].

3 Inheritance

3.1 Many Kinds of Inheritance

Inheritance is at the heart of object-oriented programming languages. Perhaps one reason that object-oriented programming is popular is that it mimics strongly how people think about everyday objects. That is, one can describe an object by how different it is from another, a kind of programming by difference. Inheritance however, is used in many different, often conflicting ways [Edwards93]. One problem with inheritance is that it is a single mechanism that encodes different kinds of component relationships [Edwards97]. In fact, Meyer defines a taxonomy of 12 different uses of inheritance, but fails to see how splitting the single inheritance mechanism into several mechanisms would help [Meyer98].



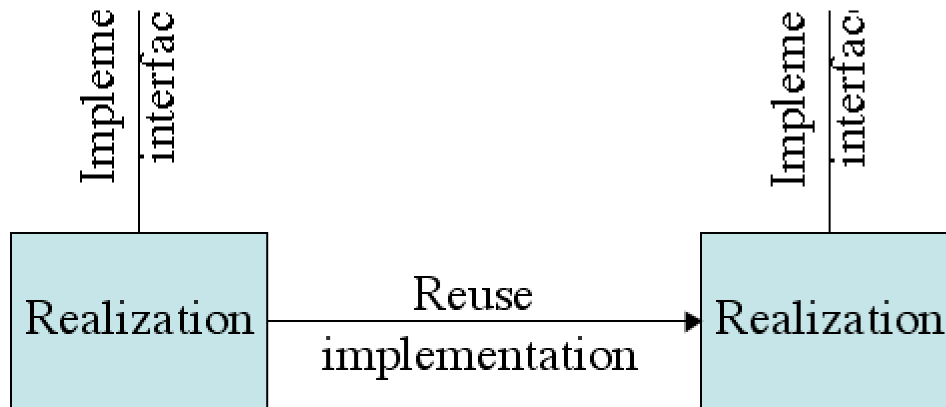


Figure 1. Some different uses of inheritance

In typical OO systems, the inheritance mechanism usually makes a class inherit two potentially separable things. The first is interface inheritance where a subclass declares that it inherits all the method signatures of the superclass. The other type of inheritance is implementation inheritance. This is where a subclass not only inherits the interface of its superclass, it also inherits the code that implements the interface. Smalltalk derived languages such as C++ and Java share the same feature that implementation and interface inheritance are coupled together. However, RESOLVE need not follow their example, since there is a strong distinction in RESOLVE between interface (concepts) and implementation (realizations). It is thus possible to think of interface inheritance as inheritance between concepts (and from concepts to realizations), and implementation inheritance as inheritance between realizations.

3.2 Relationships Between Components

One important issue in inheritance is how derived types are related to their parent types. In strongly typed languages, it is commonly held that the "is a" relationship should hold: the derived type "is a" parent type. Under one interpretation, this relationship means that a variable declared as type T should be able to hold any object of the type T, or any other types derived from T. This is called the "Liskov substitution principle" [Liskov88].

The Liskov substitution principle is especially important in a specification language like RESOLVE. This is because if the "is a" relationship is to be enforced, it means that a componentÕs specifications must also hold for any component that derives from it. That is, to enforce the "is a" relationship the derived component must be a behavioral subtype of the parent [Liskov94]. However, inheritance is typically used for more than just behavioral subtyping.

In [Edwards97], three general component relationships for which inheritance is (sometimes) used are outlined: implements, uses, and extends. RESOLVE has some support for these component relationships Ð realizations implement concepts and use other components, and enhancements extend other concepts. However, these three relations are supported only in a limited, non object- oriented fashion, i.e. in extension, new operations can be specified, but no additional specification can be made for existing operations. Similarly, with the uses relationship, there is no possibility of reusing publicly inaccessible (i.e. private) data structures.

3.3 Specification Inheritance

One way to enforce behavioral subtyping is through specification inheritance. That is, if a concept B is a behavioral subtype of another concept A, B does not need to respecify its pre- and post-conditions and invariants. Instead only new constraints are added to B, then a few rules are applied to get the total constraint. Dhara and Leavens describe a method of specification inheritance [Dhara97].

In general, preconditions can be weakened, but postconditions can only be made stronger. So if component B is a behavioral subtype of A, then the precondition of a method in B is composed of a disjunction of the precondition defined at A, and the precondition defined at B. However, the postcondition of a method in B is a conjunction of two implications. The first is that the precondition defined in A implies the postcondition defined in A. The second is that the precondition defined in B implies the postcondition defined in B. Invariants are conjoined [Dhara97].

[Dhara97] also defines a form of weak behavioral subtyping, in which, with proper restrictions on the language, history constraints of the supertype need not be enforced. This allows, for example, mutable arrays to be a behavioral subtype of immutable arrays.

3.4 Contravariance and Covariance

A subset of the above problem of enforcing behavioral subtyping is the question of having covariant or contravariant parameter passing modes. The contravariance vs. covariance debate has been heatedly discussed between the Sather and Eiffel languages. The Eiffel FAQ gives a pretty clear explanation of the issues involved:

> Consider the following situation: we have two classes PARENT and CHILD. CHILD inherits from PARENT, and redefines PARENT's feature 'foo'.
>
> ```
> class PARENT
> feature
>     foo (arg: A) is ...
> end
>
> class CHILD
> inherit PARENT redefine foo end
> feature
>     foo (arg: B) is ...
> end
> ```
>
> The question is: what restrictions are placed on the type of argument to 'foo', that is 'A' and 'B'? (If they are the same, there is no problem.)
>
> Here are two possibilities:
>
> 1. B must be a child of A (the covariant rule - so named because in the child class the types of arguments in redefined routines are children of types in the parent's routine, so the inheritance "varies" for both in the same direction)
> 2. B must be a parent of A (the contravariant rule)

Eiffel uses the covariant rule. However, to be fully compliant with the Liskov principle, recall that for any variable of type T, the variable can hold any of TÕs derived classes. So if in the above example, we have a variable of type PARENT that holds an object of type CHILD, we must be able to call foo with an argument of type A. This is not possible if B is a derived type of A.

Sather uses the contravariant rule, that is all parameter types (aside from self) are contravariant, and the return value is covariant [Szyperski93]. Since the procedure interface is really a part of the behavioral specification of a component, this scheme is better for RESOLVE. More generally, in-mode parameter types should be contravariant, out-mode parameter types should be covariant (the return value being a special case of an out-mode parameter), and in/out mode parameter types must be invariant.

3.5 Implementation Inheritance

A second use of inheritance is inheritance between realizations. In many cases, implementation and behavioral inheritance are tightly coupled. However, some have argued that behavioral subtyping should be separate from implementation inheritance. For example, a doubly-ended queue can be a behavioral subtype of stack, but it is more convenient for stack to inherit its implementation from doubly-ended queue [Snyder86].

RESOLVE's strong separation of concepts from realizations may provide an opportunity to decouple implementation inheritance from behavioral subtyping. This leads to many more questions such as whether it is necessary to put restrictions on which realizations other realizations can inherit from.

Currently, RESOLVE enhancements provide some form of implementation inheritance by allowing the enhancement to inherit all the public interface of the original module, as well as the implementation for it. However, all new functionality must be in terms of the publicly callable functions of the original component. This type of inheritance is called public inheritance, or black-box inheritance.

There may be cases, however, when private or white-box inheritance is desirable. Perhaps some the underlying data structure of a realization might be useful for another component reusing it. Java and C++ implement this with the protected modifier, allowing subclasses to access otherwise restricted data members or methods. Edwards presents a case for this type of inheritance with the example of extending a list type with a swap_rights operator. In this case, the performance of the swap_rights operator is significantly improved if white-box reuse is allowed. The paper also describes a method of representation inheritance whereby white-box inheritance can be allowed while retaining formally specified properties [Edwards96].

3.6 Multiple Inheritance

Another important issue is multiple inheritance. Sometimes, a mechanism is needed whereby a class is derived from two classes instead of one. There are several approaches taken by different languages. Some languages like Smalltalk disallow multiple inheritance altogether, sacrificing flexibility for simplicity. Other languages like C++ and Eiffel allow multiple inheritance, which brings problems of its own, such as naming conflicts and the diamond problem discussed below.

Other approaches allow a limited form of multiple inheritance. Java, for example, allows classes to inherit implementation from only one class. However, using Java interfaces, a class may derive from and implement multiple interfaces. Another alternative to multiple inheritance are mix-ins, in which common methods across multiple classes can be defined.

3.6.1 The Diamond Problem

The classic example for the diamond problem involves the types Person, Student, Teacher, and Teaching_assistant. The Student and Teacher classes both derive from Person. The Teaching_assistant class, however, derives from both student and teacher. Several problems arise from this arrangement.

For common fields inherited from Person, should there be only one copy of field, or two? If a method is redefined in either the Student or the Teacher classes, which method does Teaching_assistant use? The solution lies in renaming, for example, common fields are shared by default but if replication is necessary, then the fields are renamed, so no naming clashes occur. [Meyer98] offers a thorough discussion of the problem of repeated inheritance, and offers solutions such as undefining one or more inherited methods.

3.6.2 Mix-ins

Frequently, a set of operations are applicable to a wide set of classes. Take for example any class that implements a "compare" method, such that when A.compare(B) is called, the method returns a value that distinguishes whether A is less than B, A equals B, or A is greater than B. Clearly an additional set of operations (greater_than, less_than, greater_than_equal_to, etc.) can be defined that will work in essentially the same way regardless of the class that implements the compare method. Mix-ins allow these methods to be defined in a separate module, and inherited when appropriate.

A mixin can be thought of as an abstract subclass that can be applied to may superclasses [Bracha90], or as specifying an extension, but deferring what type the extension is for. In C++, a mixin class can be implemented by having the superclass as the parameter of a template. ie: [Smaragdakis98]

```
//C++ mixin example
template <class SuperClass>

class Mixin : SuperClass {
    ... /* mixin body */
}
```

The question of how this approach might be useful for RESOLVE may be an interesting area to explore.

4 Other Issues

4.1 Runtime Type Issues and Casting

It will have to be decided whether it is necessary for RESOLVE to check for type errors at runtime or not. Variables are dynamically typed in languages such as Smalltalk and thus type errors must always be checked at runtime. Other languages such as Eiffel and Sather try to catch as much type errors at compile time. However, even in these languages there still are ways to abuse the type system, and thus runtime type checking is still done.

One of the most common reasons for runtime type errors in languages such as Java is downward casting. In turn, downward casts are rampant in Java because of the lack of genericity. For example, container types in Java all store items of the root Object type. When extracting an item from a container therefore, a Java program must subsequently cast the item down from Object to whatever the original type of the item is.

RESOLVE has ample facilities for generics, and thus will not require downward casts in this sense. However, RESOLVEÕs swapping semantics requires a facility for downward casts. For example, if A is a variable of type Parent and B is a variable of type Child, then what happens when we swap them? The upward cast from B to A is not a problem, but the downward cast from A to B, must be handled such that type errors can be statically checked. Downcasting may also be needed when there is a legitimate need to store objects of sibling types in a container.

An alternative to swap as data movement may be assignment by consumption. In the above example, we can do A := B. Where after the assignment, A contains the old value of B, and B is reinitialized. This method retains some of the qualities of

swapping, but removes the necessity of a downward cast. There might be implications for memory management though, since the old value of A is "lost".

One possible method for doing a safe downcast is to use an approach inspired by ALGOL type unions [McGettrick78]. Recall the Ada notion of a class C as a union of all types in the type hierarchy rooted at C. Thus, C can be treated as a type union, and so to "downcast" C into one of its subtypes, we can use a variation of the ALGOL case statement. For example, we have a type Bag, with subtypes List and Stack. We can write something similar to:

```
case bag_var in
    (List) x: //do something with x
    (Stack) y: //do something with y
end case
```

This can ensure that the program is statically type-checkable, but perhaps making the progammer work a little bit more.

4.2 Exceptions

Most mainstream object oriented languages have some exception mechanism for cases where a method fails because of an exceptional condition. Exceptions are any unusual event that may require special processing. Exceptions may be used to detect errors due to precondition violations (such as the divide operator throwing a division-by-zero exception when the divisor is 0). However, exceptions are also used for unusual events that are not necessarily precondition violations. One example is I/O failure. I/O failures may not be expected in the normal course of processing, but might need special processing if they do occur [Sebesta02]. If RESOLVE is to have exception handling, it must be able to specify these exceptional conditions.

One formal specification language that supports exceptions is the Java Modeling Language (JML). JML works with exceptions by having a specification for normal behavior, and another specification for exceptional behavior. A correct implementation of a component must be able to satisfy both behaviors. That is if the precondition for the normal behavior applies, the implementation must satisfy the specification for the normal behavior, and if the precondition for the exceptional behavior applies, then the implementation must satisfy the specification for the exceptional behavior. [Leavens99]

5 Related Work

There are several formal specification languages that have been adapted for use in object- oriented programming languages. JML is a specification language tailored for Java, it comes from the Larch family of specification languages [Leavens99]. Object-Z and Z++ are some of many object-oriented extensions of the Z specification language.

VDM++ similarly extends VDM. Interestingly, VDM++ separates representation inheritance and method inheritance -- one can actually choose which superclass methods are inherited by the subclass. VDM++ also introduces the interesting construct called traces, which specifies the order in which methods may be called [Durr92].

An object-oriented RESOLVE would be different from other OO specification languages in the same way RESOLVE is different from non-OO specification languages. It is both a specification language and an implementation language. As such, it is in a unique position to disallow unsafe object-oriented development practices at the level of the implementation language.

Eiffel is an object-oriented language with facilities for generic types. EiffelÕs design by contract is provides a mechanism for invariant and pre- and post-condition checking. There are, however limitations to its type system, for example, the covariant/contravariant issue above. Sather is closely related to Eiffel, but is contravariant, and has a distinction between "subtype" (behavior) and "subclass" (implementation) inheritance. Both languages suffer from having the assertion as part of the implementation language, and thus discouraging the use of assertions that might be computationally expensive.

6 Conclusion

This paper surveys the many design issues to be considered for designing an object-oriented version of RESOLVE. While several issues remain contentious, the experiences of many people can guide the design process. Of course not all of the solutions deal exactly within the framework and unique perspective of RESOLVE.

While meant to touch on the important topics, the issues presented by this paper were not meant to be comprehensive. There are sure to be other problems and issues that need to be confronted as the new language is fleshed out. Taking up the endeavor of designing and implementing an object-oriented language based on RESOLVE promises yield many interesting research topics.

Acknowledgments

This research is funded in part by NSF grant CCR-0113181.

Bibliography

[Bracha90]
Gilad Bracha and William Cook. Mixin-based inheritance. In Proc. of the Joint ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications and the European Conference on Object-Oriented Programming, Oct 1990.

[Dhara97]
Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. Proceedings of the 18th International Conference on Software Engineering (ICSE'18), Berlin, Germany, Mar 1996.

[Durr92]
E. H. Durr and J. van Katwijk. VDM++ --- a formal specification language for object- oriented designs. In Computer Systems and Software Engineering, Proceedings of CompEuro'92, pages 214--219. IEEE Computer Society Press, 1992.

[Edwards93]
Stephen H. Edwards. Inheritance: One mechanism, many conflicting uses. In Proceedings of the Sixth Annual Workshop on Software Reuse, Nov 1993

[Edwards96]
Stephen H. Edwards. Representation inheritance: A safe form of "white box" code inheritance. In Proceedings of the Fourth International Conference on Software Reuse, IEEE Computer Society Press, April, 1996

[Edwards97]
Stephen H. Edwards, David S. Gibson, Bruce W. Weide, and Sergey Zhupanov. Software component relationships. In Proceedings of the Eighth Annual Workshop on Software Reuse, March, 1997.

[Harms91]
Douglas E. Harms and Bruce W. Weide, Copying and Swapping: Influences on the Design of Reusable Software Components. IEEE Transactions on Software Engineering, 17, 5, May 1991.

[Khoshafian86]
Setrag N. Khoshafian and George P. Copeland. Object Identity. In Object-oriented programming systems, languages and applications: (OOPSLA) conference proceedings, SiIGPLAN notices, New York, 1986.

[Leavens99]
Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A Notation for Detailed Design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds (editors), Behavioral Specifications of Businesses and Systems, chapter 12, Kluwer, 1999.

[Liskov88]
Barbara Liskov, Data Abstraction and Hierarchy. SIGPLAN Notices 23,5, May 1988.

[Liskov94]
Barbara Liskov and Jeanette M. Wing. A Behavioral Notion of Subtyping. ACM Trans. Programming Languages and Systems, 16(6):1811-1841, November 1994.

[McGettrick78]
Andrew D. McGettrick. Algol68 A first and second course. Cambridge University Press, 1978.

[Meyer98]
Bertrand Meyer, Object-Oriented Software Construction, Prentice Hall, 1998

[Sebesta02]
Robert W. Sebesta. Concepts of Programming Languages, Fifth Edition, Addison- Wesley, 2002.

[Sitaraman94]
Murali Sitaraman and Bruce W. Weide, editors. Special feature: Component- based software using RESOLVE. Software Engineering Notes, 19(4):21-67, Oct 1994.

[Smaragdakis98]

Yannis Smaragdakis and Don Batory. "Mixin-Based Programming in C++", University of Texas at Austin CS Tech. Report 98-27.

[Snyder86]
Alan Snyder, Encapsulation and inheritance in object-oriented programming. ACM SIGPLAN Notices 21, Nov 1986

[Wheeler97]
David A. Wheeler, Ada 95: The Lovelace Tutorial, Springer Verlag, 1997

Why Swapping?

Bruce W. Weide
Computer and Information Science
The Ohio State University
Columbus, OH 43210-1277 USA

weide.1@osu.edu
Phone: +1 614 292 1517
Fax: +1 614 292 2911
URL: www.cis.ohio-state.edu/~weide

Scott M. Pike
Computer and Information Science
The Ohio State University
Columbus, OH 43210-1277 USA

pike@cis.ohio-state.edu
Phone: +1 614 292 8234
Fax: +1 614 292 2911
URL: www.cis.ohio-state.edu/~pike

Wayne D. Heym
Computer and Information Science
The Ohio State University
Columbus, OH 43210-1277 USA

w.heym@ieee.org
Phone: +1 614 297 6803
Fax: +1 614 292 2911
URL: www.cis.ohio-state.edu/~heym

Abstract

An analysis of the pros and cons of all options available for the built-in data movement operator in imperative languages shows that the swap operator is the best choice, while the assignment operator is the worst.

Keywords

Assignment statement, copy, data movement, parameter passing, swap, swapping paradigm

Paper Category: technical paper
Emphasis: research

1. Introduction

For years we have advocated swapping (i.e., exchanging values) as an alternative to assignment (i.e., copying either references or values) as the built-in operator for data movement in imperative programs [Harms91, Hollingsworth00]. But these are not the only possible choices, as both we and others (e.g., [Minsky96]) have noted. Is swapping really the best way to move data between variables?

The contribution of this paper is that it maps out *all* rational options for a data movement operator and a set of criteria through which they can be compared, and thereby explains two important conclusions:

- swapping is the best possible data movement operator; and
- assignment is the worst possible data movement operator.

We hope this will help satisfy both other researchers and students who are sometimes mystified by this important feature of the RESOLVE language and discipline, and ask, "Why swapping?"

We begin in Section 2 with a brief description of the data movement problem and our previously recommended solution: swapping. This section is adapted from the appendices of [Hollingsworth00, Weide01]. In Section 3 we map out the entire solution space for the data movement question and suggest several evaluation criteria to distinguish among solution points. Section 4 uses this setup to show why swap dominates all other possible data movement operators.

2. The Data Movement Question and the Swapping Answer

The issue of how to achieve movement of data between variables is a technical problem that needs to be faced by all designers of imperative programming languages, by all software engineering disciplines that concern themselves with component-level design details, and therefore by all software engineers. We pose the problem as the *data movement question*:

> How do you make some variable (say, y) get the value of another variable (say, x)?

Before answering this question, many people legitimately ask another: Why you would ever *want* to "make some variable (say, y) get the value of another variable (say, x)". In other words, if you need the value of x at some point in a program, why not just use x directly, rather than first making y get that value and then using y? There are a few important reasons, including:

1. Parameters and returned results from function calls must be transmitted between callers and callees. For example, the

value of a formal parameter (say, y) must get the value of a corresponding actual parameter (say, x) when an operation declared as "P(y)" is called as "P(x);".
2. Sometimes you need to remember a value for future use (e.g., an intermediate result, checkpoint data, etc.) or for separate use and possible independent modification in two or more places in your code.
3. In an iteration, the code of the loop body must eventually reestablish the loop invariant. This sometimes involves making the value of a variable (say, y) that denotes some important quantity at the beginning of each iteration of the loop body, get the value of a corresponding variable (say, x) at the end of the prior iteration of the loop body.
4. When storing something into a "collection", the code implementing the collection must make some variable (say, y) in the container get the value of another variable (say, x) that is to be stored there.

The obvious answer to the original question (which is implicit in case 1 above, and explicit in cases 2, 3, and 4) is that you use the assignment operator:

```
    y := x;
```

This is the answer to the data movement question in all popular imperative languages, e.g., C, Fortran, Java, C++, C#, etc. (All these languages daftly use "=" rather than ":=" as the notation for assignment, but that's another story entirely.) Operationally, at the instruction-set level, assignment simply copies the data stored in location x into location y, replacing the data previously stored in location y. This is something that all computer hardware is designed to do efficiently, so it's no surprise to see a direct high-level-language manifestation of such an important low-level instruction. The formal semantic connection between assignment and parameter passing [Landin66] has been termed "the correspondence principle" and has even been suggested as a "design guide" for programming languages [Tennent77].

Everything about the assignment operator is not quite so simple, though. The operator has two related interpretations to modern high-level language programmers, depending on whether x and y are variables of a *value type* or variables of a *reference type*. If x and y are variables of a value type, e.g., int, then the copying leaves x and y independent of each other, so all subsequent changes to x do not affect y, and vice versa. If x and y are variables of a reference type, then the references x and y subsequently can be changed independently, i.e., without affecting each other. But the objects referenced by x and y are not independent because subsequent method calls that change the object referenced by x also change the object referenced by y, and vice versa. This difference in effect between copying values and copying references is evident in the two common parameter-passing mechanisms of call-by-value (actually, this is more properly termed call-by-value-copying) and call-by-reference (call-by-reference-copying).

An additional implication in the case of reference types is that the assignment operator for references creates an *alias*, which (if observable) dramatically complicates formal specification [Weide01] and modular reasoning about program behavior [Hogg91]. Note also that when reference types are involved, after assignment there is generally one more reference to some object and one less reference to another object, so there are storage management implications that either the programmer in a language such as C++, or the garbage collection mechanism in a language such as Java or C#, is obliged to deal with.

The question of the "observability" of aliasing arises because an object that is referenced might be either *mutable* (capable of having its value changed) or *immutable* (having a fixed value upon creation). The idea is that no reasoning problem arises from aliases to an immutable object because that object's value can't be changed through any of those references; you might as well have had one copy of the object for each reference, and there's no way to tell that you don't. In other words, a reference type where the referrent is immutable can be thought of as a value type. There may or may not be performance and storage management differences associated with the distinction, depending on the referrent type, but there are no reasoning differences.

Despite the potential for reasoning problems, both the assignment operator and the value-reference type dichotomy have been codified into modern commercial software technologies, including C++, Java, and the .NET framework. This is simply terrible from the software engineering standpoint. One reason is that the programmer now must be aware that variables of some types have ordinary values while variables of other types hold references to objects (it's the objects that have the values). For parameterized components this creates a special problem. Inside a component that is parameterized by a type Item, there is no way to know before instantiation time whether an assignment of one Item to another will assign a value or a reference. Of course, this can be "fixed" as it is in Java, by introducing otherwise-redundant reference types to immutable objects, such as Integer, that correspond to value types such as int. Actual template parameters can then be limited to reference types; but so much for parsimony. The technique known in .NET as "boxing" is a cosmetic improvement in terms of syntax but fails to remove the value-reference dichotomy or the need to understand its ramifications for program behavior.

In summary, then, going the direction of making everything a reference rather than a value only exacerbates the complications for specification and modular reasoning that are caused by potentially aliased references. It is true that the reasoning problems created by aliasing could, in principle, ultimately be avoided by requiring that *all* reference types should refer to immutable objects. This would be tantamount to mandating pure functional programming in a nominally imperative language, and to our knowledge no one is seriously proposing this as the light at the end of the tunnel on the

moving-toward-references track.

Facing up to this unfortunate situation, a decade ago we wrote about a different approach to data movement that was originally proposed and used by Bill Ogden in the early 1980s: the *swap operator* [Harms91]. The idea is to make a language's built-in data movement operator not assignment but swap, so the answer to the original data movement question becomes:

```
y :=: x;
```

This means that the values of x and y are exchanged. For instance, if x = -42 and y = 97 before swapping, then x = 97 and y = -42 afterward. Swapping x and y is, crucially, safe with respect to modular reasoning. Moreover, it is also efficient in execution because the compiler can (for data representations larger than some threshhold) simply use one level of hidden indirection and swap pointers to the representation data structures for x and y in order to effect the logical swapping of x and y. And--surprisingly to most people--the resulting *swapping paradigm* for software design and implementation demands remarkably few changes in how most programmers write imperative programs [Hollingsworth00].

3. The Solution Space and Some Evaluation Criteria

We are often asked about interactions between the swapping paradigm and other programming language and software engineering issues, such as the assignment of function results to variables, how parameter passing works, what happens in a case where you really need two copies of a value, etc. These issues are discussed in [Harms91, Hollingsworth00]. One question we are often asked that is not discussed elsewhere, however, is whether swapping is the *only* way to address the data movement question that is both efficient and safe with respect to modular reasoning. Here we explain why the answer is "no"--but why the swapping solution dominates all the alternatives in other dimensions, and why the assignment operator is the worst of all possible solutions.

First, we need to enumerate all the ways we might define a (binary) data movement operator. Let's use "<-" to denote this operator--in English we might pronounce it "gets"--so by definition the answer to the data movement question is:

```
y <- x;
```

The key to identifying and classifying all possible *meanings* for <- is to note that <- is required to leave the new y with the old value of x in order to solve the stated problem; but there is complete flexibility in how <- selects a new value for x. However, there are really only five possibilities for the new value of x after the execution of "y <- x;":

- x has no value, i.e., it is undefined;
- x has a unique, statically specified value of its type;
- x has some value of its type, but this is arbitrarily chosen from among a statically specified set of two or more values;
- x has its own old value; or
- x has y's old value.

These exhaust the possibilities that make any sense. There are only two particular values in sight as the problem is stated: the old value of x and the old value of y. Other than these, we can select the new value of x without regard to the old values of x and y, from among all the values of its type, in one of three ways: we can give x no value (the "undefined" option), give it one particular value, or give it one value chosen arbitrarily (non-deterministically) from among two or more values. The last case could be further partitioned in obvious ways, but it turns out in the end that there is no point in doing so. Non-deterministic choice could also be replaced by probabilistic choice of some kind, but again it turns out that this does not affect any of our conclusions. Hence we do not further discuss these variants.

The above five possibilities apply both for values and references. First, suppose the type of x and y is int, a value type. If beforehand, y = 21 and x = 13, then after "y <- x;" we must have y = 13. But we could leave x undefined (i.e., unusable in subsequent computations until it somehow gets a value of type int); we could leave it with a particular known value (e.g., a natural choice for type int might be 0); we could leave it with one of a number of known values (e.g., any int value, or any even value, or any prime value, or either 1 or 69); we could leave it with its old value (i.e., 13); or we could leave it with y's old value (i.e., 21).

Now suppose the type of x and y is a reference to objects of type T. If beforehand, y = ref-to-B and x = ref-to-A (where A and B are of type T), then after "y <- x;" we must have y = ref-to-A. But we could leave x undefined (i.e., unusable in subsequent computations until it somehow gets a value of a reference to type T); we could leave it with a particular known value (e.g., a natural choice for all reference types might be null); we could leave it with one of a number of known values (e.g., ref-to-O where O is any object of type T, or ref-to-O where O is any object of type T for which some predicate P(O) holds); we could leave it with its old value (i.e., ref-to-A); or we could leave it with y's old value (i.e., ref-to-B). Technically, there are many more possibilities now that references are in play! If T is a value type, then there are five additional cases, one

corresponding to each of the possibilities for value types. We could leave x = ref-to-Z (where Z is a new object whose value is undefined), or x = ref-to-Z (where Z is a new object whose value is some statically specified value of type T), and so on through x = ref-to-Z (where Z is a new object whose value equals B). And if T is a reference to a value type then we can iterate the same idea one level deeper, and so on. Doing this turns out not to affect any of our conclusions, other than to make them even more obvious because reasoning about behavior gets ever more complex and/or performance ever more dismal for each of the possible data movement operators except--you guessed it--the swap operator.

Before commencing an analysis of all the possible data movement operators, we need to list the criteria that might bear on the choice of a "best" one. The most important is that the solution should not break modular reasoning about software behavior, because this is required for any scalable software engineering discipline [Weide95]. Other than this there are no absolute requirements, but different solutions might be better or worse in terms of other important qualities. In Section 4, we consider the following desiderata that are obviously of general interest and that could differ from one data movement operator to another.

- **Efficiency**: How much time does <- take to execute? We consider faster to be better.
- **Ease of storage management**: How much complication does <- introduce into storage management? We consider less complication to be better, not only because it is easier to understand but because it is also likely to mean better efficiency in terms of overall execution time.
- **Ease of specification and reasoning**: How much complication does <- add to the understanding of component specifications and their use in modular reasoning by component clients? We consider less complication to be better.
- **Maximization of client knowledge**: How much information does the client know about the program state after <- executes? We consider more information to be better, in the sense that if all other things are equal then a more-deterministic program is easier to reason about than a less-deterministic one--and in some cases it can be more efficient, too, if the client can use the additional information to advantage.

4. Results

In the following summary table, we use a yes-no (Y-**N**) scale for support of modular reasoning, and a good-deficient (G-**D**) scale for each of the other factors. For each of the criteria, there are two entries: one for what happens with value types (val), the other for reference types (ref).

	New value of x after "y <- x;"	**Supports modular reasoning?**	**Efficiency of <-**	**Ease of storage management**	**Ease of specification and reasoning**	**Maximization of client knowledge**
1	undefined	val: Y ref: Y	val: G ref: G	val: G ref: G	val: **D** ref: **D**	val: G ref: G
2	unique statically specified value of its type	val: Y ref: Y	val: **D** ref: G	val: G ref: G	val: G ref: G	val: G ref: G
3	some value of its type, arbitrarily chosen from among a statically specified set of two or more possible values	val: Y ref: Y	val: G ref: G	val: G ref: G	val: G ref: G	val: **D** ref: **D**
4 (assignment operator)	x's old value	val: Y ref: **N**	val: **D** ref: G	val: G ref: **D**	val: G ref: **D**	val: G ref: G
5 (swap operator)	y's old value	val: Y ref: Y	val: G ref: G	val: G ref: G	val: G ref: G	val: G ref: G

Looking across the rows, we find deficiencies in all but #5 (i.e., the swap operator). Notably, the row with the most difficulties is #4, the one where x's new value equals its old value (i.e., the assignment operator). All the other options could be considered better approaches than the assignment operator, which, in a nasty quirk of fate, has been hardwired into every commercial software technology.

Let's examine, row by row, the most significant table entries for each of the possible solutions to the data movement problem:

1. Leaving x undefined is an acceptable solution, but it would be better if it didn't require the introduction of a new value for each type, i.e., "undefined", into all specifications and into the reasoning process. The additional

complication that this adds can be seen by considering the minor mess caused by allowing null values for reference variables. Suddenly, preconditions start looking like:

```
    p /= null and ...
```

and/or postconditions start looking like

```
    if #p = null then ...
        else ...
```

The same style arises when variables are allowed to be undefined. It is possible to make the problem disappear syntactically simply by making it an implicit proof obligation at each point in the program that, say, none of the variables involved in an expression or a call is undefined (except possibly y in a situation like "y <- ...;"). But it remains incumbent upon the programmer to add this to his/her informal reasoning about program behavior; it's not merely a formality needed in program proofs. Reasoning is easier if every variable always has a legal value of its type, i.e., if it is always "defined", *even if* its value at some point is arbitrary or not uniquely specified.

2. Leaving x with a unique statically specified value of its type is also a reasonably good approach. The only issue concerns the efficiency of <- for value types; for reference types there is no problem because null is a perfectly reasonable default value. Value types with small representations also cause no problem as they are quickly and efficiently constructed. The issue arises with value types that have inherently large representations. For example, consider what happens if x and y are value-type arrays of 1000 elements each. If x's new value must be such an array, constructing it is expensive for any normal representation of arrays. Similarly, y's old value must be reclaimed, and this is also likely to be expensive if the array entries require non-trivial finalization (e.g., they are references and the dereferenced objects have to be reclaimed, either immediately or by a garbage collector). It might be possible to implement this operator using lazy initialization and/or lazy finalization in an attempt to mitigate the damage to efficiency, but then the implementation becomes more complex than both swapping and the next solution without any obvious advantages to warrant it.
3. Leaving x with some (i.e., an arbitrary) value of its type is also a reasonably good approach. The reason it doesn't suffer from the same problems as the previous solution is that one correct implementation of it is to specify the set of alternatives as containing all values of the type, and then swap the values of x and y! The only question, then, is why you wouldn't *want* to tell the client the new value of x in order to maximize the client's knowledge of the values of the program's variables.
4. The assignment operator is acceptable only if it is restricted to value types. There is still an issue even in this case: efficiency. The problem arises when copying value types with possibly large data representations, as users of the C++ STL are advised to do [Musser01]. Reasoning problems arise only when assignment is used to copy references/pointers. Other papers (e.g., [Harms91, Hogg92, Weide95]) have already examined the many problems this causes. Indeed, the assignment operator is the only data movement approach where support for modular reasoning is an issue because it alone introduces aliased references. All the other solutions are alias-free, which also simplifies their storage management requirements compared to assignment.
5. According to the table, the swap operator has no deficiencies.

A careful reader may wonder whether the criteria were "rigged" to produce this outcome. Doesn't swapping have any problems? An early review suggested that programmers who are used to assignment might have to learn a new paradigm of programming in order to use swapping [Hogg92]. However, subsequent experience with swapping (both in the classroom and in building commercial software) suggests that not much changes for the programmer except the quality of the resulting software [Hollingsworth00].

One other potential problem with swapping has been suggested [Minsky96]: Because of its symmetry, swapping doesn't mesh well with the inherently asymmetric notion of subtyping in object-oriented languages. Suppose x is of type S and y is of type T, where S is a *behavioral* subtype [Leavens90] of T. (The actual objection to swapping on the grounds of its interaction with subtyping fails to note that if we're not talking about behavioral subtypes, then not even assignment should be allowed.) The claimed problem is that OO programmers want to be able to assign x to y in this case, but not vice versa--so swapping cannot be permitted.

We do not give a complete analysis of the issue here, but just explain one part of our response. Indeed, swapping cannot be permitted where the variables' types don't match, if the situation involves an *explicit* swap statement. However, the first and by far most common use of data movement is for parameter passing, and here swapping can be used even in the presence of subtyping. The declared type of the actual parameter (i.e., in the example, S) must be a subtype of the declared type (i.e., T) of the formal parameter. If it is, then for purposes of swapping to pass this parameter for this call, you simply consider the type of the formal parameter to be the same as the type of the actual parameter (i.e., S). Now the actual and formal are swappable. This doesn't affect the soundness of the separate reasoning about the correctness of the method body because that reasoning uses T as the type of the formal, and by assumption S is a behavioral subtype of T.

Even if incompatibility with subtyping were a significant objection to swapping, any of the first three "transfer" or "move" operators would still be better than the assignment operator, and they are not subject to the symmetry objection. The data movement operators of rows #2 and #3 are the best choices for those who prefer asymmetry in their data movement operator in the presence of subtyping.

5. Conclusion

Since the assignment operator was introduced at a time when languages had only value types with small representations, it was a perfectly good data movement operator for its day. With the advent of languages having user-defined types and reference types, though, it has become sub-optimal for general use; i.e., it should not be *the* built-in data movement operator in modern languages. But assignment is deeply woven into the fabric of computing for most software engineers, whose early computing education typically involved programming only with built-in value types having small representations. Languages could still allow assignment statements in such situations.

Swapping has many advantages over assignment as the built-in data movement operator that is available for *every* type. Most importantly, it doesn't interfere with modular reasoning. It is efficiently implementable for all value and reference types, regardless of the sizes of their data representations. It dramatically simplifies storage management because there is only one reference to any object; hence, the clean-up discipline is that you reclaim resources when a variable goes out of scope. Adoption of the swap operator also simplifies specification and reasoning because it unifies values and references in a fundamental way: There is no logical difference to the client between swapping two object references and swapping the values of the referenced objects. (There was a hint of this property in Section 3, where we mentioned without justification that only the swap operator does not get progressively messier to deal with in the face of references to references to ...) This means that introduction of the swap operator in place of the assignment operator--and in this case in preference to the other three possible data movement operators as well--facilitates a move toward a uniform value semantics. This is *precisely the opposite direction* taken by Java and .NET. Some details and consequences of such a move are discussed in [Weide01]. (Of course, this is one of the main features of the RESOLVE language and discipline.)

If you use any commercial software technology, then you face the data movement question all the time. You might not know it because the assignment operator is so ingrained in all of us that to many people it seems inseparable from the very idea of imperative-language programming. But it has been demonstrated [Hollingsworth00] that:

- there is nothing sacred about the assignment operator;
- after replacing the assignment operator with the swap operator, you can still readily build "real" commercial software systems;
- it is possible to reason modularly about the behavior of the resulting software; so,
- software designed based on the swap operator is more reliable and more easily maintainable than software designed using the assignment operator for data movement.

Not only is nothing sacred about the assignment operator, it is the *worst* choice among all possible data movement operators. Swapping dominates assignment when data movement operators are compared along several important dimensions. In fact, swapping dominates all the other possibilities as well.

Acknowledgments

This work is supported in part by the National Science Foundation under grant number CCR-0081596, and in part by a grant from Lucent Technologies.

References

[Harms91]
Harms, D.E., and Weide, B.W., "Copying and Swapping: Influences on the Design of Reusable Software Components", *IEEE Transactions on Software Engineering 17*, 5 (1991), 424-435.

[Hogg92]
Hogg, J., Lea, D., Holt, R., Wills, A., and de Champeaux, D., "The Geneva Convention on the Treatment of Object Aliasing", *ACM SIGPLAN OOPS Messenger 3*, 2 (Apr. 1992), 11-16; http://gee.cs.oswego.edu/dl/aliasing/aliasing.html, viewed 8 May 2002.

[Hollingsworth00]
Hollingsworth, J.E., Blankenship, L., and Weide, B.W., "Experience Report: Using RESOLVE/C++ for Commercial Software", *Proceedings of the ACM SIGSOFT Eighth International Symposium on the Foundations of Software Engineering*, 2000, ACM Press, 11-19.

[Landin66]
Landin, P.J., "The Next 700 Programming Languages", *Communications of the ACM 9*, 3 (Mar. 1966) 157-166.

[Leavens90]
Leavens, G.T., and Weihl, W.E., "Reasoning About Object-Oriented Programs That Use Subtypes", *Proceedings OOPSLA '90/SIGPLAN Notices 25*, 10 (Oct. 1990), 212-223.

[Minsky96]
Minsky, N.H., "Towards Alias-Free Pointers", *Proceedings ECOOP '96, LNCS 1098*, Springer-Verlag, New York, 1996, 189-209.

[Musser01]
Musser, D.R., Derge, G.J., and Saini, A., *STL Tutorial and Reference Guide, Second Edition*, Addison-Wesley, 2001.

[Tennent77]
Tennent, R.D., "Language Design Methods Based on Semantic Principles", *Acta Infomatica 8*, 2 (1977), 97-112.

[Weide01]
Weide, B.W., and Heym, W.D., "Specification and Verification with References", *Proceedings OOPSLA Workshop on Specification and Verification of Component-Based Systems*, ACM, October 2001; http://www.cs.iastate.edu/~leavens/SAVCBS/papers-2001 viewed 8 May 2002.

[Weide95]
Weide, B.W., Heym, W.D., and Hollingsworth, J.E., "Reverse Engineering of Legacy Code Exposed", *Proceedings 17th International Conference on Software Engineering*, ACM Press, 1995, 327-331.

Integration and Conceptual Modeling

Thomas J Wheeler
University of Maine
Department of Computer Science
Orono, ME 04469
wheeler@umcs.maine.edu

Abstract

It is becomming increasingly common that research efforts, and the development of systems to support them, are being undertaken by multidisciplinary teams. Life science multidisciplinary research has become the norm, rather than an innovation. There are several reasons for this trend. Insights from several points of view provide a richer understanding of issues and more opportunities for solutions. Also insights in one discipline often come from thought patterns from another discipline. In many disciplines the research in part(s) of the domain has reached the stage where exploring issues and advances in adjoining parts, and in the interaction of parts, is warrented. In hierarchical systems, especially life, research at individual levels is different in kind from research at others, and integration across levels has become possible and desirable.

Multidisciplinary research presents a marvelous opportunity, but also creates serious problems. Merging of the disciplines' conceptualizations must occur, at least in the (separate)minds of the collaborators. To be effective, merging must leverage the expertise of individual discipline members, as well as that of general purpose designers.

Systems that support this type of research, are complex systems, with significant semantic mismatch problems. The increasing use of computer databases for organizing disparit research results in data integrations problems. Models for each database or data source are designed independently, in accordance with a domain's conceptual model. These models are further specialized to a particular research effort, then encoded using general purpose data models. The independence of development and the differing cultures of the fields, cause incompatibilities between models and programming interfaces. The notation's general purpose nature loses(filters) insights and intuition from domains' natural illustrations and explanations of key models.

Within the software engineering strategy focussed on a designer's perspective of development, there are three classes of concerns that must be addressed in creating a system: developing a substantial understanding of a problem and its domain, designing a system concept, and architecting and realizing that system. This paper explores a mechanism and a methodology for all but realization, based on integration of multiple disciplines' models. It distills the inherent structure of each model, blends models to create the structure for the integrated domain and creates views of this blended structure for each participating discipline
.
The approach has four aspects. "Natural" graphic depictions and explanations are integrated with general purpose models. The underlying structure of the natural models is extracted by analysis of the metaphorical underpinnings of those models. Models are blended using the character of one to underlie semantics taken from others. A framework for visualization of the blended domain is created using the natural depictions, explanations and underlying metaphors.

This technique provides a framework for understanding, organizing and supporting interdisciplinary work. It improves the conceptual modeling process by integrating more domain intuition and insight into the process. We will illustrate the mechanisms and a methodology for use with excerpts from interdisciplinary projects in molecular biology and ecology.

Keywords

Integration, Model, Model Blending, Natural Graphic, Multidiscipline, Multidisciplinart Research

Paper Category: technical paper
Emphasis: research

1. Introduction

Research by multidiscipline teams provides insights that are richer than any single discipline can have, but introduces cross-disciplinary semantic problems. In addition, designing the complex systems needed to support multidisciplinary research surfaces a number of core informatics systems problems. The issues at the heart of these problems concern development of a unified conceptual model for the research, and integration of heterogeneous parts of systems developed by different disciplines. The first needs a strategy for merging individual discipline's models into a coherent, multiple discipline model. It must preserve the insights of the individual disciplines, and create a framework for insights in the (new) merged discipline. The second must use general purpose formal models and discipline specific natural models. Integration requires general purpose (i.e. domainless) models, languages, and notation. Scientific depth and insight require leveraging the notations and thought patterns natural to specific disciplines. The notations and thought patterns natural to specific disciplines produce heterogeneous representations, even when encoded in general purpose notations.

Within the software engineering strategy focussed on a designer's perspective of development, there are three classes of concerns that must be addressed in creating a system[Guttag,Horning]. First, designers need to develop a substantial understanding of a problem to be solved, the domain in which it resides and the community that works in that domain. Second, they must design a system concept, by a creative process whose success appears to be based on insights into the central concepts and activities of the domain and its community. Third, they need to architect and realize that system. The process of developing the understanding and the insight is called system/domain analysis, the development of the system concept and the architecture of the system are called design and realization is called implementation. Intuition and insight in the analysis/design process come from discipline specific understanding and are captured using formal notations. This perspective on development has modeling as its central focus.

A number concepts that have emerged in cognitive science over the past decade can help with the domain understanding and system concept design concerns. First, understanding the structure and conceptual metaphor basis of human cognitive models provides a framework for understanding and design. Models, on which understanding and design are based, are captured in a more natural way. Second, there is a cognitive process for developing new meaning for concepts, when they are placed in different contexts. In the new analysis of this process, conceptual model "blends" provide a basis for developing new meaning. An analog of this can support cross discipline and multidiscipline envisioning, which these type of systems need to exhibit. Third, explanation and understanding are related, with explanation reifying an understanding, while from the other direction, explanation develops understanding. This interplay of explanation and understanding provides leveraging during analysis. And fourth, the categorization concept of radial categories leads to a useful characterization of "natural" depictions used in explanations. This characterization places constraints on the amount and type of abstraction used in models derived from those explanations, providing guidelines for analysis.

The interoperation of heterogeneous data types requires transformation of data from its original representation to a standard representation (in a data warehouse architecture) or to a usage representation (in a federated architecture). The main technology developers addressing these types of issues come from the Computer Science/ General Purpose Modeling communities [e.g. Roth, Davidson]. Valid integration, however, depends on the expertise of scientific curators, understanding the source(s), and scientists who design and perform virtual experiments and analyses with the resulting merged information [Bult].

Central Ideas

Models underpin system design, perceptual interfaces, and programming interfaces to data sources, analysis programs and other subsystems. The idea here is that development of a unified conceptual model for the research be based on analysis of explanations and accompanying natural depictions. Their capture and analysis is based on analogy to the development of "natural" cognitive models and the blending of models, by an individual, during creative insight.

To address architecture level concerns, general purpose formal and natural modeling languages must be combined. General purpose modeling languages and notations are needed as a basis for automation, but because they are general purpose, they must abstract away any discipline specific intuition and insight. General purpose modeling languages and notations are also formal. But, Scientists understand and explain concepts and issues in their field using notation and language natural to their discipline. Depth and insight require the notations and thought patterns natural to specific disciplines. The models emerging from their explanations and depictions need be combined with general purpose modeling languages and notations, such as XML and UML. The general purpose, formal models are structured by the natural notations for more valid models. The natural notation models are integrated into the general purpose notation models, to retain the discipline's insight into the domain.

The aim of this effort is to develop a mechanism and a methodology for integration of separate discipline’s models, based on distilling the inherent structure of each model, blending them to create the structure for the integrated domain and creating views of this blended structure for each participating discipline. This paper results from work on a number of life sciences projects, taking a systems' biology approach, which is naturally interdisciplinary, in areas from genomics to ecology. It looks

at this type of system from a point of view which uses component based architectures, providing data integration through interfaces. The focus of this work is in integrating ideas about cognitive models and model blending from cognitive science, into a model based development process .

2. The Problem

Background

It is becomming increasingly common that research efforts, and the development of systems to support them, are being undertaken by multidisciplinary teams. In the life sciences for instance, multidisciplinary research has become the norm, rather than an innovation. There are several reasons for this trend. The first is that insights from several points of view provide a richer understanding of issues and more opportunities for problem solutions. A related reason is that an insight in one discipline often comes from a thought pattern from another discipline. Third, in many disciplines, research in some parts has reached the stage where exploring issues and advances in adjoining parts, and in the interaction of parts, is warrented. Lastly, in hierarchical systems, especially life, research at individual levels is different in kind from research at others, and integration across levels has become possible and desirable.

While multidisciplinary team based research presents a marvelous opportunity, it also creates serious problems. In multidisciplinary research, blending of the disciplines' conceptualizations must occur, at least in the (separate)minds of the collaborators, but also in the resulting or supporting systems.

Researchers in the life sciences are increasingly taking on a multidisciplinary character, using the system's biology approach to understanding issues from molecular biology to ecology. They are finding that integrative issues create a barrier to progress in molecular biology[Paton] and that a complex system approach is essential in ecological systems[Wu,Marceau]. The multidisciplinary character of system's biology is changing the landscape of life science research.

Problem

Development of the complex systems needed to support multidisciplinary research surfaces a number of core informatics systems issues at two levels. At the System Level, development of systems to support different disciplines needs to address the different thought patterns and cultures of the disciplines. Conceptual models underpin the design, understanding and use of systems; presenting a model effectively to users from different disciplines must be in terms fitting that discipline's model of the domain. At the Architecture/Component Level, development of parts of these systems by different disciplines, leads to heterogeneity and distribution problems in data sets, processing strategy, information presentation, and analysis. Heterogeneity leads to conceptual model ("impedance") mismatch, at semantic and pragmatic levels, between different parts of the system.

Conceptual model mismatch problems exhibits themselves in system level problems such as (mis)interpretation of results displayed by a system, difficulty in development using software from another discipline, and difficulty in a multidisciplinary team developing complementary and integrated understandings of others' concepts. These problems come about because the designer's conceptual model creates the character of the system and its components, and that character is usually difficult to understand from the (different) point of view natural to the user.

At the architecture level, significant mismatch problem comes from the increasing use of computer databases for organizing research and its results leads to data integrations problems for multidisciplinary research/systems. The data model for each database, or other data source, is designed independently, in accordance with a domain's conceptual model, specialized to a particular research effort, then encoded using general purpose data models. Because of the independence of development and the differing cultures of the fields, incompatibilities occur between models and at programming interfaces. Because of the general purpose nature of the notation for encoding the data models, the insight and intuition in each domain's natural illustrations and explanations of its key models is lost.

A related problem occurs in the interplay of formal and natural notations and thought patterns. Models that underlie the interfaces of systems and subsystems, start in notations of, and are framed in terms of thought patterns of, specific disciplines; but must be encoded in terms of general purpose formal notations. The integration or translation that occurs at system and subsystem interfaces requires the use of general purpose languages, models and notation; but scientific depth requires leveraging the notations and thought patterns of specific disciplines.

Motivation

There is considerable evidence that people remember, understand and use concepts, systems, etc. better when they have a memorable structure[Lakoff, Pinkler, Chomsky, MacEachern]. Understanding seems to take place in terms of metaphor based structures[Lakoff, Johnson, Mandler]. Organizing takes place through placing concepts into categories and structures which enriches concepts and connects them to other related and helpful concepts[Fauconnier], making them easier to

understand while making them more powerful and useful. Humans seem to organize concepts by logical, imagistic and procedural patterns [MacEachern]; by categorization[Lakoff], grouping things that are similar in some way, and by fitting them into conceptual structures to provide larger concepts; classifying them as generalizations and specializations of some naturally understood "basic level" categories.

Everyday existence, thinking about and using familiar, commonplace objects and concepts is organized effectively by perceptual images and metaphor based mental models of the objects and their context [Fauconnier, Lakoff, Johnson, Mandler], so that humans can naturally deal with their everyday environment. These perceptions and concepts provide an organized understanding of the everyday environment. When one wants to function with similar ease and facility in an artificially created environment such as system development or interdisciplinary research, where one cannot have a naturally constructed framework in which to reason and act, one has to consciously create the organized understanding necessary for natural and effective action.

Software engineering has been a search for organization patterns for differing types of system concerns. Creating explicit organized representations for work products has provided guidance in organizing work as well as useful structuring of the results[Parnas, Guttag,Horning]. The effort reported on here is an effort to find and develop a set of organization patterns for system development in the situation where the concerns are complex and dissimilar in some way, and have different cultures, and the development participants, and their products, have to interact.

3. The Position - Approach

Designers need to develop a substantial understanding of a problem to be solved(or an opportunity to be taken advantage of)[Guttag,Horning], the domain in which it resides and the community that works in that domain. They must also develop a system concept, by a creative process whose success appears to be based on insights into the central concepts and activities of the domain and its community, developed while building that understanding. Only after these do they need to architect and realize that system.

Our approach is based on using results from cognitive science research, within a framework provided by software engineering, to provide a basis for system design, by capturing the models on which the design is based in a more natural way. This is done by explicit analysis of the natural models of each discipline, capturing the essence of each in a formal model which is then used for system design. The structure and conceptual metaphors used in explanations of the discipline's concepts are captured and analysed for use in formal models in the system design.

We also combine the use of formal and natural models in developing a cross-discipline or multidiscipline understanding of particular domains. This is based on an analysis of the cognitive process which develops new meaning for a concept when placed in a different context, using blended cognitive models and metaphorical mappings. An analog of this process can provide a basis for supporting the cross discipline and multidiscipline envisioning which these systms need to support.

The concept of radial categories characterizing "natural" depictions is used in analysing explanations, and developing the abstraction used in models derived from those explanations. "Natural" semantic support for creative insight in multidiscipline research is provided by the emergent structure of these blended cognitive models and metaphorical mappings of meanings in the user's discipline.The models developed are used to design the system, the perceptual interfaces provided by systems, and the abstract interfaces to data sources, analysis programs and other subsystems.

Criteria

The criteria by which this work is judged address three areas:
1. The models and interfaces to software components must be valid with respect to scientific experiments. The models must accurately reflect the concepts used in the design of experiments. The computer data must reflect the results of the scientific experiments by the disciplines creating the data. The interfaces, both program and perceptual, must portray the data, and inferences from it, in accordance with the models of the disciplines.
2. The models and interfaces must structurally, semantically, and pragmatically conform to each discipline's conceptual models and the blended models. The models and interfaces must address the thought patterns and activities of each discipline.

3. The models and interfaces to software components must resonate with the intuitions and insights of each discipline and support development of new multidiscipline intiutions and creation of new multidiscipline insights.

4. Elaboration

Mechanism and Methodology

This paper develops a mechanism and a methodology for integration of separate discipline models, based on the process of distilling the inherent structure of each model and blending models to create the structure for the integrated domain. It

addresses issues at the terminology, semantics, pragmatic and activity levels.

The mechanism consists of two parts; (1) analysis of idealized cognitive models and metaphorical semantics; and (2) synthesis supporting creativity, using cognitive model blending. Recent results show that people's (e.g. scientist's and developer's) cognitive models appear to be based on idealized abstract cognitive models (Idealized Cognitive Models (ICM's)[Lakoff/Johnson], Schemata[MacEachern], Conceptual Structures[Jackendorf]) and structural mappings among their elements [Fauconnier]. Some of these are learned at an early age, common to people in general, and unconciously applied. Others, specific to their (specialized) domain, are learned as an an adult, shared among the members of the discipline, and are skills, whose unconcious application is because of training and experience[MacEachern], or are conciously applied[Fauconnier].

Cognitive Models: Metaphors and Maps

It appears that we develop an understanding of something, or domain, by forming a perceptual image based[Lakoff, Marr] conceptual model [Norman], or knowledge scheme[MacEa, Pink] of it, which gives it form[Alexander], allowing us to mentally simulate[Fauconnier] properties and activities[Norman] of the thing or in the domain, and thus infer properties and behavior. The mental images that we form have a visual or kinesthetic character[Lakoff] allowing them to be directly understood[Lakoff] and to be spatially manipulated in the mind[Marr, MacEacern] so that relationships and interactions can be inferred. Use, activity and interaction of objects, relationships and concepts in a domain make sense via the structure of the mental models. They do this in terms of the functionality and constraints they afford[Norman], the mappings of these aspects of objects with those of other understood objects, and metaphors[Johnson] applied to them to give them meaning.

Metaphors provide a basis for compositional semantics of the natural and abstract world. Analysis of language use shows [Lakoff,Johnson, Lakoff&Johnson] the pervasiveness of conceptual metaphors for both primary concepts and for their composition. Primary metaphors become part of our cognitive unconscious automatically, beginning in infancy[Lakoff&Johnson] providing experiential semantics for abstract concepts and activities. Complex concepts are structured by structural metaphors and mappings[Johnson] to included or associated primary or complex metaphors. As an example of metaphorical semantics from molecular biology (Figure 1) consider the following sentence: "a strand of DNA consists of pairs of bases" where "strand" is metaphorically a path(or line) and "base pairs" are at the positions of steps(or points) along the path.

Mappings of various kinds between cognitive models appear to be at the heart of what we mean when we say we understand some concept[Fauc]. Projection mappings use the structure, and vocabulary, of one domain to understand some other domain. Function mappings structure correspondences, organizing the knowledge in a field. Schema mappings structure situations transfering concepts into new contexts. In the example in figure 1, there is a projection map from the domian of paths, a primary metaphor learned in infancy from (probably) crawling and/or actual observation of different things, animate and inanimate, moving along different paths.

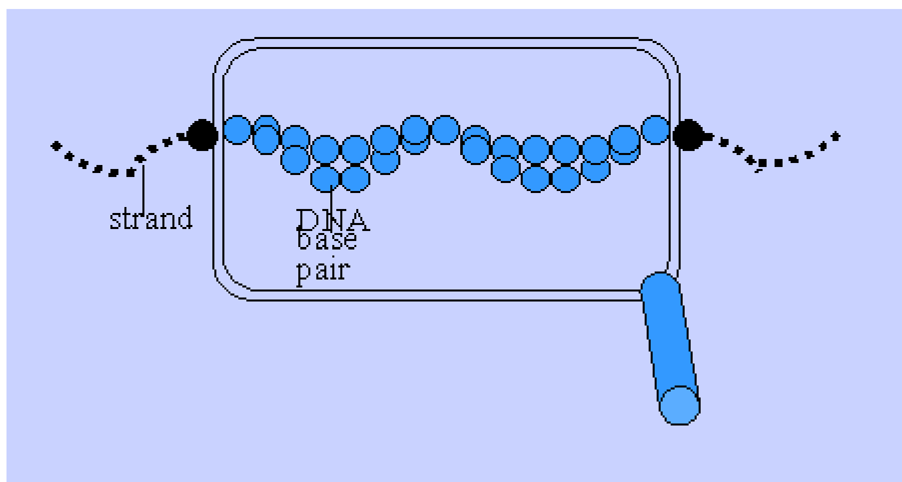


Figure 1

Cognitive Model Blending

Cognitive model blends use the structure and behavior of one conceptual model, or domain, (the source) to create insight about another concept, domain or situation (the target). Cognitive models can be pictured as structures composed of concepts as its components. The concepts are not atomic, but rather are either metaphorically structured entities or other complexes.

The components and the model have complex and open ended semantics, based on the activities they (metaphorically) participate in, the situations they (metaphorically) are found in and the behavior and character they (metaphorically) exhibit. Thus any conceptual model has a discernible structure and an enormous amount of emergent properties. The emergent properties come into the mind when the model is "run".

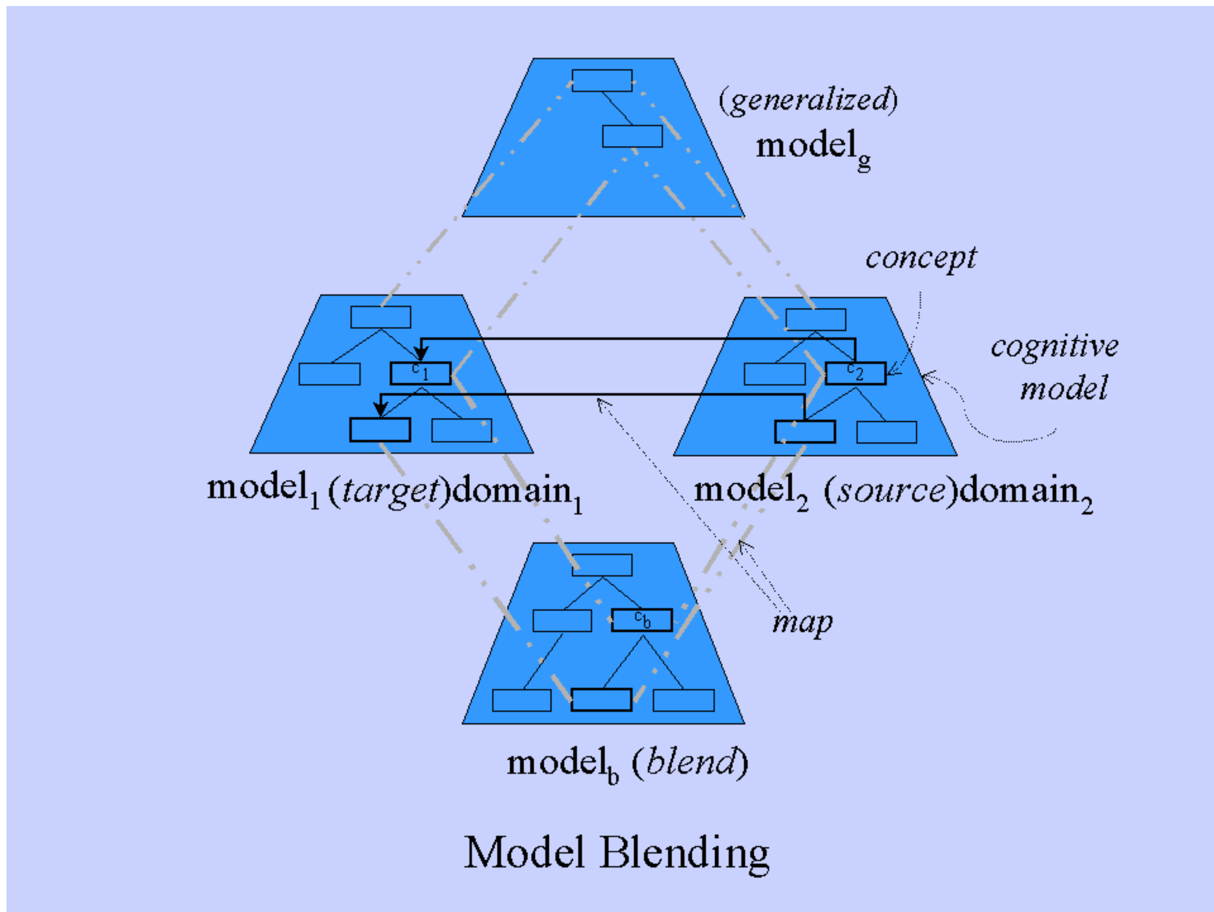


Figure 2

In a conceptual model blend(Figure 2), a person (say a scientist from domain 1) is trying to develop a conceptual model (model1 in Figure 2) of some subject matter. Another person (say a scientist from domain 2) explains the subject matter from her point of view, using a model (model2 in Figure 2) and terminology from her domain. There are some aspects of domain 2 and domain 1 which have a common, abstract semantic basis (modelg in Figure 2) and these serve to provide some abstract semantic anchors between the two people's concepts. But some of the concepts in model1 and model2 have the same metaphorical basis underpinning them (modelb in Figure 2), allowing(causing) the models to form a blend. The first person can understand model2 in the context of domain1 by use of the blend (modelb) and the vocabulary of model2.

Providing a view of a model, from one domain, in terms of a model in another domain is done by a similar technique. The semantics of the second domain are overlain on the information from the first domain. The interpretation in the second domain uses that domains thought patterns, expanded to include data from the first. We refer to this process as model morphing.

As an elaboration of example of metaphorical semantics from molecular biology above (Figure 1) consider the following further sentence: "The DNA 'zipper' (another metaphor) must attach itself to the gene in an area a certain distance unstream from the area to be 'unzipped', for transcription to take place another certain distance downstream". (Figure 3).

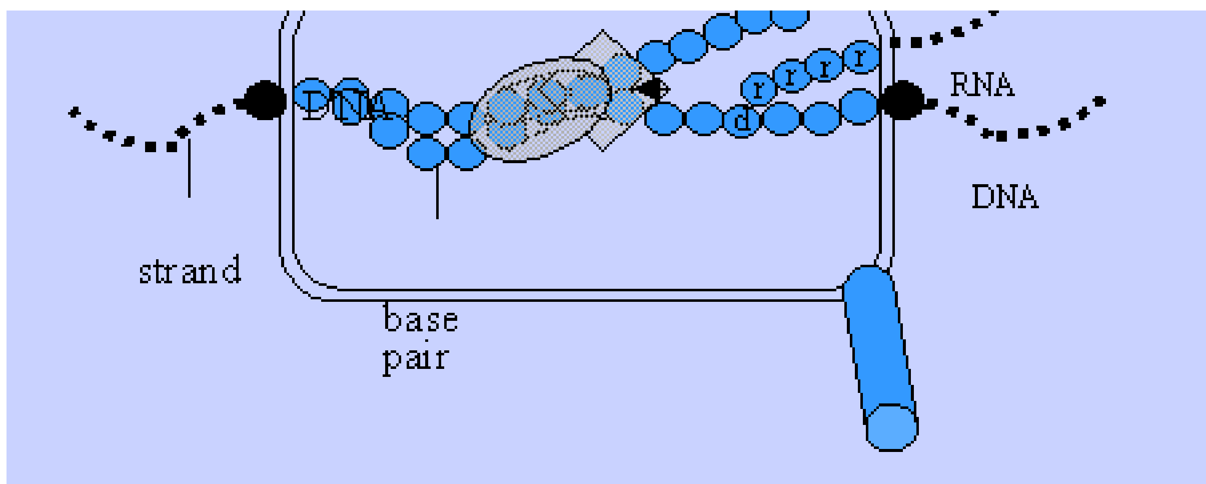


Figure 3

Here the geography metaphor is used to explain (and model) the process of transcription. The geography model is (something like) a map of some terrain with a number of paths, with the DNA path being specialized to a stream flowing downhill in a valley. The DNA and the stream have the same shape. In an area on the map, "upstream" of the start of a distributary (overlain by the unzipping metaphor) the unzipping occurrs. Following that (i.e. downstream from that place) transcription can take place, modeled as a distributary.

The methodology

The methodology has four aspects. The first consists of integrating the “natural” graphic depictions and explanations each discipline makes of its core concepts, with the general purpose models of their systems. The second extracts the underlying structure of the natural models based on analysis of the metaphorical underpinnings of the models. The third creates a blend of the models’ structures, using the character of one to underlie the semantics composed from elements from other models mapped onto slots of the core model. The fourth creates a framework for visualization of the blended domain by creating natural depictions and explanations from the blended structure of the blended domain.

Intuition and insight in the analysis process come from discipline specific understanding. This comes from direct or indirect experience in the domain. In systems which are primarily the product of an individual, the understanding comes from working in the field. In systems developed by a team, the understanding must be shared, through informal(conversations) or formal (meetings) verbal/visual interactions or documented representations; preferably all of these. The technique we describe here provides a framework for developing this understanding.

5. Example

A research project to look into the development of a database for the Genome Spatial Information System (GenoSIS) Project required an integrated genomic-spatial data model, which formalizes genomics(a computer analog of DNA molecular biology) along with metric, topological, and metrically uncertain properties and relationships among genome features. Such a genome spatial data model facilitates the powerful spatial reasoning and inferences that are part of spatial information science and thereby allows biologists to ask questions about the contextual and organizational significance of the spatial arrangement of genome features. These functional capabilities should, in turn, aid in the automation of repetitive analytical tasks associated with the mapping of genome features and drive the discovery of biologically significant aspects of genome organization and function.

We begin the analysis by attempting to characterize the biological processes we hope to model. We characterize the models and thought patterns in the domain both informally by working with the different disciplines and listening to their explanations; and formally by use of the mechanisms described in this paper. We formally characterize the models and thought patterns in the domain in two ways: by considering the natural graphics that are used within the domain among practitioners and by constructing a lexicon or ontology of the concepts which are essential in the domain. With these tools we develop a conceptual model, which can be formalized as the data model.

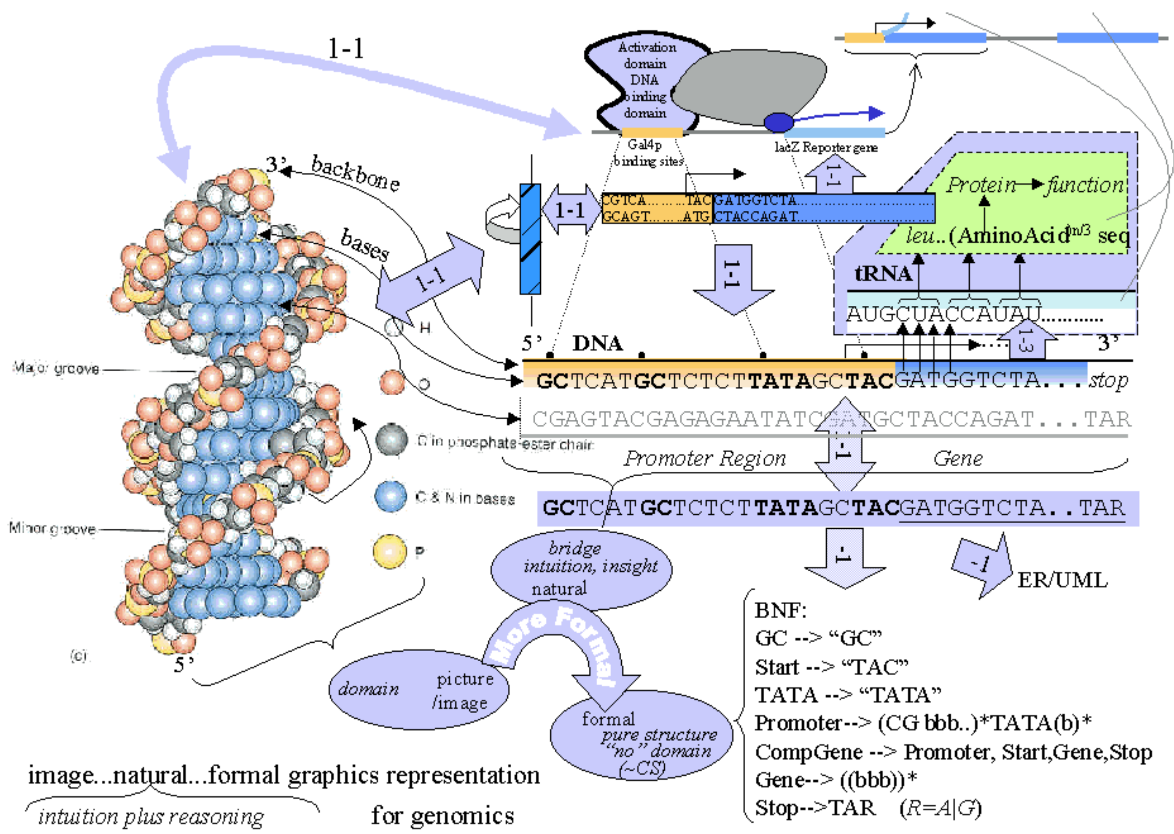
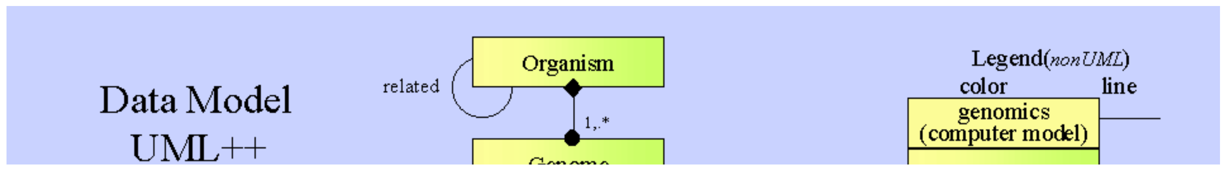


Figure 4

Some "natural" graphic depictions of the biological processes of interest to us are shown in Figure 4. First, in the lower left, a picture-like image grounds the conceptualization with a real(istic) image. There are a number of natural maps (natural isomorphisms) from that image. It is mapped onto a spirally wound tube, which is then unwound to produce a depiction as a ribbon with the 5' to 3' molecule strand on top. There are two further natural mappings, the upper one showing a simplified straight line depiction, with supplementary colored segments, and the other showing a blowup making sequence of the individual bases apparent. These natural depictions are used to illustrate explanations of the primary concepts in genomics.

The depictions and the accompanying explanations are part of the raw material for the analyses described above. Another part of the raw material is an analysis of the vocabulary in the explanations and from gossaries or ontologies. As an example of an explanation is as follows:
"An **Organism** is the largest *category* for our purposes here. We wish to compare **different organisms** in some analyses. Each organism *has* one or more **Genomes**. A genome *is made up of* one or more **Chromosomes**. The genome *contains* many **Features**, which we define to be recognizable functional elements. A feature *may be* simple or composite, that is, composed of other features making up a **Feature Set**. The **genome** and any **feature** within it *are sequences of* **Base Pairs**. The **base pair** sequence is the raw primary output of **genome** sequencing efforts. Features are determined by *applying* a number of **algorithms**, e.g. pattern matching, *to the sequence*. We indicate the **Start** and **Stop positions** *of a* **feature** *as determined by* the algorithm used to locate the **feature**. Since DNA is double stranded, for any feature on DNA we indicate which **Strand** contains the feature and *how far along* the strand it starts. Biologists interested in comparing organisms seek ... "

In this explanation, **words denoting objects(concepts)** in the model are boldfaced. Words *signaling the use of a metaphor* are italicized, and words useful in guiding the modeling are underlined, for instance "contains" signals the *container* metaphor, **Strand** and *how far along* signal the *path* metaphor specialized to a strand, while **genome** and any feature within it signal the structure of a **genome sequence**.



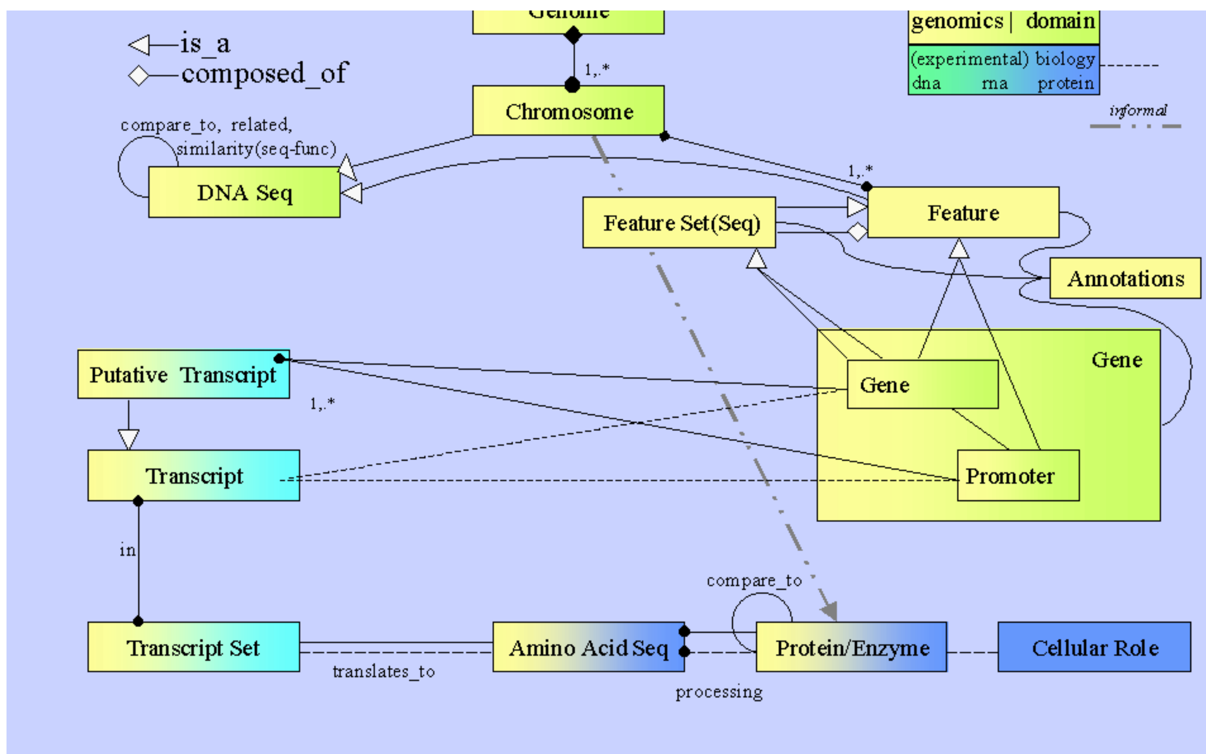


Figure 5

The UML model developed from the analysis is shown in Figure 5 (color coded to highlight the biological science parts and the conputer, genomics parts).

A part of the formal Abstract Interace for the database, using this model would look like:
Using an (object oriented) "XML++ " (;-)) syntax :

```
<!ELEMENT feature    <-- (2) -->
 (feature_type,start_coordinate,end_coordinate,strand
  feature_name,feature_symbol?,comment?,time_stamp?,
  transcript*)>    <--!Semantics:Structure-->
       <--!Metaphor:Part-Whole-->
......

<!ELEMENT gene1 is_a feature (annotation_list) >
 <--!Semantics:is_a = Structure Addition-->
 <--!Semantics:(..) = Structure-->
```

Here, the structure is given by an XML Element definition (instead of BNF), the formal semantics is given by (a formal model from) a collection of formal models, and the "natural semantic basis is given by (a conceptual metaphor from) a collection of (ground or complex) conceptual metaphors.

The automation of the semantics would be accomplished by pattern matching at the formal model and the conceptual metaphor levels.

6. Related Work

This work is a relatively new endeavor, applying research work in cognitive science to software engineering This effort is, in some sense, the development of an explicit "organized understanding" of the thinking, communication, activities, and documentation of a research or development project, in an attempt to provide a prescriptive approach to separating, and

addressing, the concerns of such a project.

As this project is an outgrowth of software engineering research, its core concepts are about design/research organization; characterizing, making explicit and managing the work products of the design/research efforts; developing prescriptive methods for separating the concerns of the effort, and addressing interaction, interface and interoperation issues between disciplines and multi-domain software (sub)systems.

This work is related to some of the work in the reuse community [WISR, & ?] which addresses conceptual underpinnings or reuse and interoperability [Wileden, Porter, Simos, Capilla, Kiczales, Latour]. It is related to the software architecture community[Garlan&Shaw,] whose work is one of the major sources of organizing, and working at, the Abstract Implementation level in the model presented below. It is also related to, but addresses a different aspect of collaboration than the computer supported cooperative work community[CSCW, ECSCW] who focus mainly on computed mediated interaction, whereas we focus on the perceptual, cognitive and (human) communication aspects of the problem.

7. Conclusion

This paper presented a mechanism and a methodology for integration of separate discipline's models into a multi-discipline model and the development of program component interfaces to heterogeneous infirmation which conform to these models. The mechanism distills the inherent structure of each model by explicating the metaphors underlying the discipline's explanations and natural graphic depictions of its core concepts. Integration is by model blending to create the structure for the integrated domain. Presentation, both perceptually and programatically is accomplished by creating views of this blended structure for each participating discipline. The focus of this work is in integrating ideas about cognitive models and model blending from cognitive science, into a model based development process .

The technique creates models and interfaces to software components that are valid with respect to scientific experiments. They accurately reflect the concepts used in the design of experiments by capturing them from the most accurate and insightful representations available. They are structured and given semantics in terms of models isomorphic to those apparent in the minds of discipline members.

The models and interfaces structurally, semantically, and pragmatically conform to each discipline's conceptual models because they capture the essence of the discipline's explanations[Tuffte]. The multi-discipline models are blended by the same mechanisms used by discipline members. The models and interfaces use the thought patterns and activities of each discipline.

Because they capture the essence of the discipline's explanations, the models and interfaces to software components should resonate with the intuitions of each discipline. Because they are blended by the same mechanisms used by discipline member they should support development of new multi-discipline intiutions and creation of new multidiscipline insights.

References

C. Alexander, "Notes on the Synthesis of Form" Penguin Books, 1982

F. Belz, D. Suthers, and T. Wheeler, "Architecture Abstraction Hierarchy - Reference Model," IEEE Learning Technology (P1484) Guideline (P1484.1), 1997.

F. P. Brooks, The Mythical Man-Month, Reading, MA: Addison Wesley, 1975, 1996.

C. Bult, et.al. "Mouse Genome Informatics in a New Age of Biological Inquiry" Bio-Informatics and Biomedical Engineering (BIBE2000) Arlington VA Nov. 2000

N. Chomsky, "Linguistics and Adjacent Fields: A Personal View," The Chomskyan Turn,(A. Kasher, ed.), New York: Blackwell, 1991.

G. M. Copper, The Cell: A Molecular Approach, Washington, D.C.: ASM Press, 1997.

S. Davidson, "BioKlesli: a Digital Library for Biomedical Research" Intl. J. Digit. Lib. 1(1) 1997

G. Fauconnier, "Mappings in Thought and Language" Cambridge Univ. Press 1997

C. Gallistel, Organization of Learning, Cambridge, MA: MIT Press, 1993

D. Garlan, R. Allen, and J. Ockerbloom, "Architectural mismatch, or, why it's hard to build systems out of existing parts,"

17th International Conference on Software Engineering, ICSE 95, April 1995.

J. Guttag, J. Horning "Formal Specification as a Design Tool"Formal Specification Case Studies MIT Press 1989

D. Hester, D. Parnas, and D. Utter, "Using Documentation as a Software Design Medium," Bell System Technical Journal, V60, 1981.

R. Jackendorf, "Cognitive Architecture of Language" MIT Press, 1984

G. Kiczales, "Aspect-Oriented Programming," Eighth Annual Workshop on Software Reuse, March 1997.

G. Lakoff, Women, Fire, and Dangerous Things-What Categories Reveal About the Mind, Chicago: University of Chicago Press, 1987.

G. Lakoff and M. Johnson, Philosophy in the Flesh-The Embodied Mind and Its Challenge to Western Thought, New York: Basic Books, 1999.

L. Latour, T. J. Wheeler,and B. Frakes, "Descriptive and predictive aspects of the 3C's model: SETA1 working group summary," First Symposium on Environments and Tools for Ada, Ada Letters, XI, 3, (Spring 1991).

J. Mandler "Preverbal Representation and Language" In Language and Space Bloom, Peterson, Nadel, Garrett Eds. MIT Press 1996

D. Marr, Vision: A Computational Investigation into the Human Representation and Processing of Visual Information, San Francisco: W.H. Freeman, 1982.

A.L. MacEachren, How Maps Work-Representation, Visualization, and Design, New York: The Guilford Press, 1995.

D. Norman "The Design of Everyday Things" Penguin 1986

N. Paton et.al. "Conceptual Modelling of Genomic Information"Bioinformatics V16,no.6 2000

S. Pinkler, The Language Instinct: How the Mind Creates Language, New York: William Morrow and Co., 1994.

M. I. Posner, ed., Foundations of Cognitive Science, Cambridge, MA: MIT Press, 1996.

M. Roth, F. Ozcan, L. Haas, "Don't Scrap it, Wrap it, A Wrapper Architecture for Legacy Data Sources" In Proc. VLDB Athens Greece Aug. 1997

M. Shaw and D. Garlan, Software Architecture: Perspectives on an Emerging Discipline, Upper Saddle River, NJ: Prentice Hall, 1996.
SIGSOFT, Fifth Symposium on Software Reusability, May 1999.

M. A. Simos, "Domain Envisioning: A Lightweight, Incremental Approach to Getting a Company Started with Systematic Reuse," Ninth Annual Workshop on Software Reuse, January 1999.

J. F. Sowa, Knowledge Representation-Logical, Philosophical, and Computational Foundations, Cambridge, MA: Brooks/Cole, 2000.

R. E. Slavin, Cooperative learning: Theory, research, and practice. Englewood Cliffs, NJ: Prentice-Hall, 1990.
Spatial-Genomics Project, University of Maine

E. R. Tufte, Envisioning Information, Cheshire, CT: Graphics Press, 1990.

T. J. Wheeler and J. Richardson "A Two Layered Interfacing Architecture," Journal of Standards & Interfaces, v.13, Elsevier-North Holland, 1991.

T. J. Wheeler, "Object Database Interface," DARPA Open Object Oriented Database Workshop, Dallas, Tx., 1992.

T. Wheeler, M. Dolan, and J. Richardson, A Framework for Interdisciplinary Collaboration Univ. of Maine CS Report, 2000

L. Wong, "Kleisli, its Exchange Format, Support Tools, and an Application in Protein Interaction Extraction" Bio-Informatics and Biomedical Engineering (BIBE2000) Arlington VA Nov. 2000

J. Wu, D. Marceau, "Modeling Complex Ecological Systems: an Introduction" Ecological Modelling 2002

Appendix

- Example Document Type Definition:

```
<!DOCTYPE organisms[
<!ELEMENT organisms (organism*)>
                                                        <!-- !Model: Set -->
                                              <!-- !Semantics:Collection -->

<!-- ***************************************************************  -->

<!ELEMENT organism (kingdom,genus,species,subtype?,
                           common_name,comment?,genome+)>
                                                                <!-- (1) -->
                                                    <!-- !Model:Structure-->
                                  <!-- !Semantics:Whole/Part Construction-->
<!ATTLIST organism      id ID #REQUIRED
                                               <!-- !Model:Unique Nat Num-->
                                       <!-- !Semantics:(Source-Path)-Goal-->

                        Name (#PCDATA)>
                                                         <!-- Model:Name -->
                                                   <!-- !Semantics:Symbol-->
<!ELEMENT kingdom (#PCDATA)>
                                                        <!-- !Model:Name -->
                                                   <!-- !Semantics:Symbol-->
<!ELEMENT genus (#PCDATA)>
                                                        <!-- !Model:Name -->
<!ELEMENT species (#PCDATA)>
                                                        <!-- !Model:Name -->
<!ELEMENT subtype (#PCDATA)>
                                                        <!-- !Model:Name -->
<!ELEMENT common_name (#PCDATA)>
                                                        <!-- !Model:Name -->
<!ELEMENT comment (#PCDATA)>
                                                        <!-- !Model:ch*  -->
                                          <!-- !Semantics:Points_on_Line-->

<!-- ***************************************************************  -->

<!ELEMENT feature (feature_type,start_coordinate,
        end_coordinate,strand,feature_name,feature_symbol?,
        comment?,time_stamp?,transcript*)>
                                                               <!-- (2) -->
                                                   <!-- !Model:Structure-->
                                            <!-- !Semantics:Construction-->
<!ATTLIST feature id ID #REQUIRED>
                                            ` <!-- !Model:Unique Nat Num-->
                                      <!-- !Semantics:(Source-Path)-Goal-->
<!ATTLIST feature idref IDREF #REQUIRED>
                                                        <!-- !Model:REF-->
                                      <!-- !Semantics:Source-(Path-Goal)-->

<!ELEMENT feature_type (#PCDATA)>
                                                       <!-- !Model:Name -->
                                                  <!-- !Semantics:Symbol-->
<!ELEMENT start_coordinate (#PCDATA)>
                                                    <!-- !Model:Nat Num -->
                                        <!-- !Semantics:Position_on_Line-->
<!ELEMENT end_coordinate (#PCDATA)>
                                                    <!-- !Model:Nat Num -->
                                        <!-- !Semantics:Position_on_Line-->
<!ELEMENT strand (#PCDATA)>
                                      <!-- !Model:Name(="plus","minus") -->
<!ELEMENT feature_name (#PCDATA)>
                                                       <!-- !Model:Name -->
```

```
<!ELEMENT feature_symbol (#PCDATA)>
                                                       <!-- !Model:Name -->
<!ELEMENT DNA_SEQUENCE (#PCDATA)>
                                                       <!-- !Model:ch*  -->
                                          <!-- !Semantics:Points_on_Line-->
<!ELEMENT comment (#PCDATA)> )>
                                                       <!-- !Model:ch*  -->
<!ELEMENT time_stamp (#PCDATA)>
                                                       <!-- !Model:Time -->
```

```
<!-- !Semantics:Points_on_Line-->

<!-- ***************************************************************  -->

<!ELEMENT gene1 is_a feature (transcript,annotation_list) >   <!-- (3) -->
                                   <!-- !Model: is_a = SubType (& deRef)-->
          <!-- !Semantics:Additional Construction & (Source-Path)-Goal -->
                                          <!-- !Model: (..) = Structure -->
                                           <!-- !Semantics:Construction -->

<!ELEMENT transcript (protein|enzyme) >
                                                      <!-- !Model:Union -->
                                                  <!-- !Semantics:Choice-->
<!ELEMENT protein(sequence_length,amino_acid_sequence) >
                                              <!-- !Model:Structure-->
                                             <!-- !Semantics:Construction-->
<!ELEMENT sequence_length  (#PCDATA)> )>
                                                    <!-- !Model:Nat Num -->
                                             <!-- !Semantics:Line_Segment-->
<!ELEMENT amino_acid_sequence>
                                                        <!-- !Model:ch*  -->
                                           <!-- !Semantics:Points_on_Line-->
<!ELEMENT annotation_list (annotation)*>
                                                    <!-- !Model: annot* -->
                                           <!-- !Semantics:Points_on_Line-->
<!ELEMENT annotation(annot_type,annot_val)>
                                                  <!-- !Model:Structure-->
                                 <!-- !Semantics:Part/Whole Construction-->
<!ELEMENT annot_type (#PCDATA)>
                                                      <!-- !Model:Name -->
                                                 <!-- !Semantics:Symbol-->
<!ELEMENT annot_val (#PCDATA)>
                                                      <!-- !Model:ch*  -->
                                          <!-- !Semantics:Points_on_Line-->

<!-- ***************************************************************  -->

<!ELEMENT promoter is_a feature (annotation_list) >
                                                              <!-- (4) -->
                                     <!-- !Model: is_a = SubType & deRef-->
          <!-- !Semantics:Additional Construction & (Source-Path)-Goal -->
                                          <!-- !Model: (..) = Structure -->
                                           <!-- !Semantics:Construction -->

<!ELEMENT annotation_list (annotation)*>
                                                   <!-- !Model: annot* -->
                                          <!-- !Semantics:Points_on_Line-->
<!ELEMENT annotation(annot_type,annot_val)>
                                                  <!-- !Model:Structure-->
                                            <!-- !Semantics:Construction-->

<!ELEMENT annot_type (#PCDATA)>
                                                      <!-- !Model:Name -->
                                                 <!-- !Semantics:Symbol-->
<!ELEMENT annot_val (#PCDATA)>                        <!-- !Model:ch*  -->
                                          <!-- !Semantics:Points_on_Line-->
```

```
<!-- ***************************************************************  -->

<!ELEMENT gene2 is_a feature view_of(promoter, gene1) >
                                                               <!-- (5) -->
                                    <!-- !Model: is_a = SubType & deRef-->
          <!-- !Semantics:Additional Construction & (Source-Path)-Goal -->
                                            <!-- !Model:view_of = View -->
                                            <!-- !Semantics:Surface_of -->
                                         <!-- !Model: (..) = Structure -->
                                          <!-- !Semantics:Construction -->

]>
```

```
<!-- ***************************************************************  -->
```

```
<!-- ***************************************************************  -->
```

```
<!-- INSTANCES:  -->
<!-- Eukaryota_Rodentia_Mus_musculus_GALT -->
                                                  <!--  dtd(1) -->
<ORGANISM Name=Mus musculus>
<KINGDOM>               Eukaryota               </KINGDOM>
<GENUS>                 Rodentia                </GENUS>
<SPECIES>               Mus musculus            </SPECIES>
<strain>                B6/CGAFIJ               </strain>
<db_xref>               taxon:10090             </db_xref>
<sex>                   female                  </sex>
<tissue_type>   liver                           </tissue_type>
<COMMON_NAME>   House Mouse                     </COMMON_NAME>
<annotation>    This reference sequence was provided by
the Mouse Genome database (MGD).                </annotation>
<CHROMOSOME>
<CHROMOSOME_NUMER>              4               </CHROMOSOME_NUMBER>
<CHROMOSOME_NAME>               chromosome 4    </CHROMOSOME_NAME>
<CHROMOSOME_STRUCTURE>  linear                  </CHROMOSOME_STRUCTURE>
<STRAND>                                plus    </STRAND>
<Symbol>                                GALT    <Symbol>
<Feature_Name> galactose-1-phosphate uridyl transferase </Feature_Name>
<cM_Position>                   19.9                    </cM_Position>
<MGI_Accession_ID>              M:96265                 <MGI_Accession_ID>
<FEATURE>
<FEATURE_TYPE>          source                  </FEATURE_TYPE>
<START_COORDINATE>      1                       </START_COORDINATE>
<END_COORDINATE>        13731                   </END_COORDINATE>
<FEATURE_NAME>          GALT                    </FEATURE_NAME>
<DNA_SEQUENCE>
         1 ttcagggtgg gtgggcgggg ggagacatgg aatggggcgc tcaccttgtg taccttaggt
        61 caattcgtgt ggcctcacgt cgcatagcga cgcgatcctg agcagcgcca cgaggcttca
       121 gaggcggacc gatggcagcg accttccggg cgagcgaaca ccagcatatt cgctacaacc
       181 cgctccagga cgagtgggtg ttagtgtcgg ctcatcgcat gaagcggccc tggcaaggac
       241 aagtggagcc ccagcttctg aagacagtgc cccgccacga cccactcaac cctctgtgtc
       301 ccggggccac acgagctaat ggggaggtga atccccacta tgatggtacc tttctgtttg
       361 acaatgactt cccggctctg cagcccgatg ctccggatcc aggacccagt gaccaccctc
       421 tcttccgagc agaggccgcc agaggagttt gtaaggtcat gtgcttccac ccctggtcgg
       481 atgtgacgct gccactcatg tctgtccctg agatccgagc tgtcatcgat gcatgggcct
       541 cagtcacaga ggagctgggt gcccagtacc cttgggtgca gatctttgaa aataaaggag
       601 ccatgatggg ctgttctaac ccccatcccc actgccaggt ttgggctagc agcttcctgc
       661 cagatatcgc ccagcgtgaa gagcgatccc agcagaccta tcacagccag catggaaaac
       721 ctttgttatt ggaatatggt caccaagagc tcctcaggaa ggaacgtctg gtcctaacca
       781 gtgagcactg gatagttctg gtccccttct gggcagtgtg gcctttccag acacttctgc
       841 tgccccggcg gcacgtgcgg cggctacctg agctgaaccc cgctgagcgt gatctcgcct
       901 ccatcatgaa gaagctcttg accaagtacg acaatctatt tgagacatcc tttccctact
       961 ccatgggctg gcatggggct cccacgggat taaagactgg agccacctgt gaccactggc
      1021 agctccacgc ccactactac cccccacttc tgcgatccgc aactgtccgg aagttcatgg
```

```
     1081 ttggaccgtg tacactggca gctcacgccc actacctacc cccacttctc ggatccgcaa
     1141 ctgtctatga aatgcttgcc caggcccagc gtgacctcac tcccgaacag gccccagaaa
     1201 gattaagggc gcttcccgag gtacactatt gcctggcgca gaaagacaag gaaacggcag
     1261 gatcaccatt gcttgactgt gaccacatca gggccttgaa tctttgtacc tgacagacct
     1321 gggacctgga gttcgggcag atgtgacatc aataaaactg cgtctcacat ttt
</DNA_SEQUENCE>
</FEATURE>

<!-- ***************************************************************  -->

                                                                         <--  dtd(3) -->
<GENE1>
<FEATURE_TYPE>          gene                    </FEATURE_TYPE>
<START_COORDINATE>      132                     </START_COORDINATE>
<END_COORDINATE>        1313                    </END_COORDINATE>
<Feature_Name>          GALT                    </Feature_Name>
<DNA_SEQUENCE>
      121            atggcagcg accttccggg cgagcgaaca ccagcatatt cgctacaacc
      181 cgctccagga cgagtgggtg ttagtgtcgg ctcatcgcat gaagcggccc tggcaaggac
      241 aagtggagcc ccagcttctg aagacagtgc cccgccacga cccactcaac cctctgtgtc
      301 ccgggggccac acgagctaat ggggaggtga atccccacta tgatggtacc tttctgtttg
      361 acaatgactt cccggctctg cagcccgatg ctccggatcc aggacccagt gaccaccctc
      421 tcttccgagc agaggccgcc agaggagttt gtaaggtcat gtgcttccac ccctggtcgg
      481 atgtgacgct gccactcatg tctgtccctg agatccgagc tgtcatcgat gcatgggcct
      541 cagtcacaga ggagctgggt gcccagtacc cttgggtgca gatctttgaa aataaaggag
      601 ccatgatggg ctgttctaac ccccatcccc actgccaggt ttgggctagc agcttcctgc
      661 cagatatcgc ccagcgtgaa gagcgatccc agcagaccta tcacagccag catggaaaac
      721 ctttgttatt ggaatatggt caccaagagc tcctcaggaa ggaacgtctg gtcctaacca
      781 gtgagcactg gatagttctg gtccccttct gggcagtgtg gcctttccag acacttctgc
      841 tgccccggcg gcacgtgcgg cggctacctg agctgaaccc cgctgagcgt gatctcgcct
      901 ccatcatgaa gaagctcttg accaagtacg acaatctatt tgagacatcc tttccctact
      961 ccatgggctg gcatggggct cccacgggat taaagactgg agccacctgt gaccactggc
     1021 agctccacgc ccactactac cccccacttc tgcgatccgc aactgtccgg aagttcatgg
     1081 ttggaccgtg tacactggca gctcacgccc actacctacc cccacttctc ggatccgcaa
     1141 ctgtctatga aatgcttgcc caggcccagc gtgacctcac tcccgaacag gccccagaaa
     1201 gattaagggc gcttcccgag gtacactatt gcctggcgca gaaagacaag gaaacggcag
     1261 gatcaccatt gcttgactgt gaccacatca gggccttgaa tctttgtacc tga
</DNA_SEQUENCE>
<TRANSCRIPT>
<PROTEIN>
<protein_id>    AAA37658.1"                     </protein_id>
<db_xref>               GI:193422"              </db_xref>
<SEQUENCE_LENGTH>               109             </SEQUENCE_LENGTH>
<AMINO_ACID_SEQUENCE>
MAATFRASEHQHIRYNPLQDEWVLVSAHRMKRPWQGQVEPQLLKTVPRHDPLNPLCPG
ATRANGEVNPHYDGTFLFDNDFPALQPDAPDPGPSDHPLFRAEAARGVCKVMCFHPWS
DVTLPLMSVPEIRAVIDAWASVTEELGAQYPWVQIFENKGAMMGCSNPHPHCQVWASS
FLPDIAQREERSQQTYHSQHGKPLLLEYGHQELLRKERLVLTSEHWIVLVPFWAVWPF
QTLLLPRRHVRRLPELNPAERDLASIMKKLLTKYDNLFETSFPYSMGWHGAPTGLKTG
ATCDHWQLHAHYYPPLLRSATVRKFMVGPCTLAAHAHYLPPLLGSATVYEMLAQAQRD
LTPEQAPERLRALPEVHYCLAQKDKETAGSPLLDCDHIRALNLCT
</AMINO_ACID_SEQUENCE>
</PROTEIN>
</GENE1>



<!-- ***************************************************************  -->
                                                                         <--  dtd(5) -->
<GENE2>

<GENE1>
<<FEATURE_TYPE>         gene                    </FEATURE_TYPE>
<START_COORDINATE>      132                     </START_COORDINATE>
<END_COORDINATE>        1313                    </END_COORDINATE>
<Feature_Name>          GALT                    </Feature_Name>
<DNA_SEQUENCE>
```

```
      121>          atggcagcg accttccggg cgagcgaaca ccagcatatt cgctacaacc
                                        ...

<PROMOTER>
<FEATURE_TYPE>          UAS                     </FEATURE_TYPE>
<START_COORDINATE>      13                      </START_COORDINATE>
<END_COORDINATE>        22                      </END_COORDINATE>
<DNA_SEQUENCE>  13> gggcgggggg                  </DNA_SEQUENCE>
</PROMOTER>
</GENE1>
</GENE2>

<!-- ***************************************************************  -->
```

Specifying and Verifying Collaborative Behavior in Component-Based Systems

Levent Yilmaz

Trident Systems Incorporated
Simulation and Software Division
10201 Lee Highway Suite 300
Fairfax, VA 22030 USA

levent@tridsys.com
Phone: +1 703 267 6754
Fax: +1 703 273 6608
URL: http://csgrad.cs.vt.edu/~lyilmaz

Stephen H. Edwards

Dept. of Computer Science
Virginia Tech
660 McBryde Hall
Blacksburg, VA 24061-0106 USA

edwards@cs.vt.edu
Phone: +1 540 231 5723
Fax: +1 540 231 6075
URL: http://people.cs.vt.edu/~edwards/

Abstract

In a parameterized collaboration design, one views software as a collection of components that play specific roles in interacting, giving rise to collaborative behavior. From this perspective, collaboration designs revolve around reusing collaborations that typify certain design patterns. Unfortunately, verifying that active, concurrently executing components obey the synchronization and communication requirements needed for the collaboration to work is a serious problem. At least two major complications arise in concurrent settings: (1) it may not be possible to analytically identify components that violate the synchronization constraints required by a collaboration, and (2) evolving participants in a collaboration independently often gives rise to unanticipated synchronization conflicts. This paper briefly overviews and summarizes a solution technique that addresses both of these problems. Local (that is, role-to-role) synchronization consistency conditions are formalized and associated decidable inference mechanisms are developed to determine mutual compatibility and safe refinement of synchronization behavior. It has been argued that this form of local consistency is necessary, but insufficient to guarantee a consistent collaboration overall. As a result, a new notion of global consistency (that is, among multiple components playing multiple roles) is briefly outlined: causal process constraint analysis. Principally, the method allows one to: (1) represent the intended causal processes in terms of interactions depicted in UML collaboration graphs; (2) formulate constraints on such interactions and their evolution; and (3) check that the causal process constraints are satisfied by the observed behavior of the component(s) at run-time.

Keywords

Verification, testing, reuse, collaborative behavior, synchronization, parameterized collaboration.

Paper Category: technical paper
Emphasis: research

1. Introduction

Although a considerable amount of research has been devoted to component and composition-based concepts and techniques, the impact of this technology on component certification has yet to be fully explored. The clients of reusable components will expect the parts they purchase and use to work correctly with respect to their catalog descriptions in a given new context. Therefore, a concrete component must be trustworthy; that is, it must correctly implement the abstract service specification on which its clients will depend. Since a reusable component is developed with specific contextual assumptions and is unlikely to comply completely and without conflicts with the domain of interest, the challenge of certification with reusable components is far greater than that of conventional software. For tractability purposes of assuring correctness of applications developed from reusable components, local certifiability principles are advocated to facilitate establishment of component properties outside of the context in which they are embedded [Weide & Hollingsworth, 1992].

1.1 Local Certifiability Notion in Component-Based Systems

RESOLVE discipline provides a framework and language that facilitates achievement of modular verification of correctness of functionality. The discipline enables local or component-wise certification to demonstrate that a concrete component (i.e., realization) achieves the functional behavior of its abstract specification (i.e., concept). The certification is modular and local since no assumptions or references are made with respect to the behavioral features of clients. The composition of a component is argued to work correctly as long as the clients interact with the reused component in conformance with its expectations in a sequential context.

However, the verification as well as reuse of concurrent components faces challenges not seen, for example, in most UI toolkits and data structure libraries [Lea 1999]. Factors such as specification of design policies (i.e., synchronous or asynchronous, local or remote, callback, forwarding, or delegation), reflective, multilevel design (composite components), security, safety, liveness, fault-tolerance, recovery policies make specification, certification, and reasoning of reusable components difficult. Furthermore, components cannot be built in isolation; instead, they must observe compatibility constraints that are often phrased as architectural rules [Lea, 1999; Dellarocas, 1997] and synchronization constraints, especially in concurrent settings.

Development with reuse requires determining if the certified properties and constraints of a component would facilitate its playing certain roles to establish a specific purpose in the context into which it is embedded. This paper briefly summarizes an approach, called collaborative behavior verification, that aims to facilitate the examination of interaction dynamics (synchronization constraints) of components in concurrent settings to assure that they can fit and play the roles required by the parameterized collaborations into which they are embedded.

1.2 The Terminology in Collaborative Behavior Verification Approach

Software designs consist of components, roles, and, collaborations. Actors that represent the components play roles collaboratively to implement collaborations. Collaborations are descriptions of how a group of components jointly behave when configured together in specific ways. A parameterized collaboration represents a design construct that can be used repeatedly in different component designs. The participants of the collaboration, including classifier roles and association roles, form the parameters of the generic collaboration. The parameters are bound to particular model elements in each instance of the generic collaboration. Such a parameterized collaboration can capture the structure of a design pattern. Each role is associated with a type specification, called interaction policy. Interaction policies are language independent specifications of external interface and visible synchronization behavior that dictate the patterns of interchange of messages among collaborating peers. A role describes a collaboration participant fulfilled by a specific component when instantiated with a particular connector. Hence, a component can play one or more roles. The context into which the component is embedded needs to be aware of the assumptions, as shown in Figure 1, regarding the correct usage of the component. The component also publishes its obligations indicating that as long as the context uses the component in conformance with its assumptions, it will guarantee the interaction behavior denoted by the obligations.

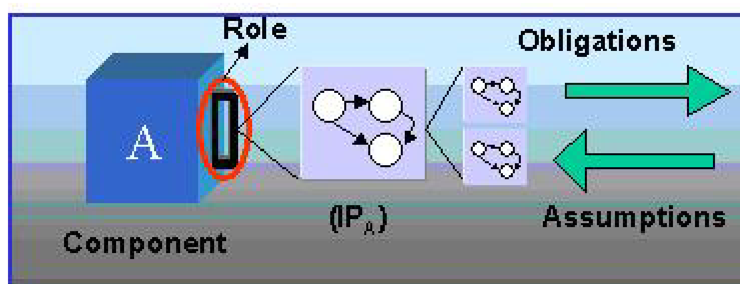


Figure 1: Assumptions and Obligations of a Role

This paper is organized as follows. In section 2 we identify and formulate the problem. Sections 3 and 4 outline and summarize our approach to local consistency, while section 5 is devoted to the global consistency notion. In section 6, we overview the related work in collaboration design and behavior verification. Finally, in section 7 we conclude and discuss contributions of our work.

2. The Problem: The Local and Global Consistency Notions of Collaborative Behavior for Parameterized Collaborations

In a parameterized collaboration design, one views software as a collection of components that play specific roles in interacting, giving rise to collaborative behavior. From this perspective, collaboration designs revolve around reusing collaborations that typify certain design patterns. Each component in a collaboration can have one or more roles. Unfortunately, verifying that active, concurrently executing components obey the synchronization and communication requirements needed for the collaboration to work is a serious problem. At least two major complications arise in concurrent

settings: (1) it may not be possible to analytically identify components that violate the synchronization constraints required by a collaboration, and (2) evolving participants in a collaboration independently often gives rise to unanticipated synchronization conflicts. These complications lead to the formulation of the following questions that will be addressed in this dissertation:

- *Given generic parameterized collaborations and components with specific roles, how do we verify that the provided and required synchronization models are consistent and integrate correctly?*
- *How do we guarantee that the synchronization behavior is maintained and kept consistent with the collaboration constraints as the roles and collaboration are refined during development?*
- *How do we verify if the emergent cross-cutting synchronous causal behavior, which arises due to the causal interference, interaction, and coordination of components, is as intended? Furthermore, how do we determine whether the coordination of roles played by each used component is evolved as expected during the evolution of the collaboration as the system is maintained?*

There are several questions that this research seeks to answer: Which representation scheme and model should be used to represent synchronization contracts (i.e., interaction policies) of a component at the abstract level? What are the necessary and sufficient conditions and corresponding decision procedures that facilitate tracking and maintaining semantic dependencies between components based on the pertinent synchronization model? That is, how do we verify that the roles integrate correctly, and how do we ensure that the incremental enhancements to collaboration role models are consistent with the original commitments made at previous levels of abstractions? Furthermore, given that the used components fit into the roles expected from them, is the actual emergent collaborative behavior, which arises from the cascading of local interactions, achieved or evolved the way intended by the designer?

2.1 Local Role Consistency in Collaboration-Based Software Design

In the establishment of a market for software components, the separation of specification and implementation is a well-known prerequisite.

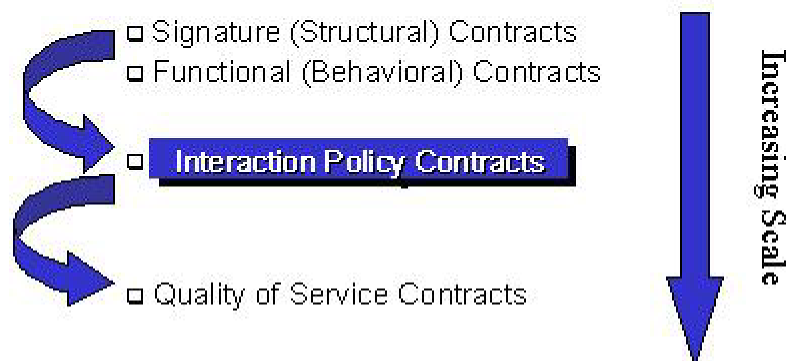


Figure 2: Reuse Contracts

Clients choosing the components based on their catalog descriptions (i.e., contract specifications) expect them to behave correctly with respect to their published contracts. In general, as shown in figure 2, contracts can be divided into four main levels with increasingly negotiable properties [Beugnard *et al*. 1999]. The first level, that of basic contracts, is required to simply define the signature and types. The second level, based on behavioral contracts, improves the level of client confidence, particularly in a sequential context. The third level, emphasizing the synchronization and interaction pattern contracts, improves confidence particularly in concurrent and distributed settings. The fourth level, quality of service contracts, quantifies the quality of service. Although consistency notions exist for functional and structural contracts, still missing with regard to collaboration-based software with collaborative behavior reuse is a general consistency notion that emphasizes synchronization contracts. Hence, determining methodologically whether the synchronization constraints of used components' roles fit into their local context (i.e., horizontal consistency) or are safely extended (i.e., vertical consistency) is still an elusive goal.

2.2 Decision Procedures for the Analysis of Local Consistency Conditions

Once functional and signature compatibility is established, synchronization play a key role in assessing the fitness of components in a collaboration. The preliminary analysis for synchronization fitness (consistency) of a component is determined by its interaction policy compatibility with respect to its context and depends on the interaction semantics under

consideration. Given the semantic domain used to express the synchronization consistency conditions based on a certain interaction and request scheduling mechanism, for correct realization, the next step involves developing decision procedures to analyze the conditions mapped to the semantic domain. This involves procedures determining (1) role compatibility (horizontal consistency), (2) safe refinement (vertical consistency) of role and collaboration evolutions, and (3) correct realization and implementation of the expected local role interaction policies.

2.3 Causal Process Constraint Analysis for Achieving Global Consistency

For achieving global consistency, analysis of local synchronization constraints for horizontal and vertical consistency is necessary but insufficient. This is due to the fact that the synchronization behavior of a role often depends on other roles that, because of shared data state space and control and data dependent causal connections, interfere with its interaction decisions. Furthermore, for many component engineering tasks, including reasoning, verification, and reuse, such causal connections depict why an artifact is configured as it is. This is especially important when non-functional aspects are derived from a particular design rationale based on certain causal processes embedded in the collaboration. Hence, to achieve global consistency, there is a need to provide methods for capturing, constraining, and analyzing expected causal processes. Due to cascaded local synchronous interactions, the emergent collaborative behavior designates how the component's behavior depends upon the composition of its constituent components and how the component behavior is accomplished. Here, the conjecture asserts that in order to verify that a component achieves its goal as intended by the designer, one must show not only that the final expected condition is satisfied but also that the components participated in the expected causal process that resulted in the expected function. Hence, the causal processes underlying the intended design patterns of the component designate a certain aspect of the designer's rationale and purpose to establish certain non-functional as well as functional goals. In order to realize this rationale and purpose it is essential to provide a means of (1) representing the intended causal processes, (2) formulating constraints on causal processes and their evolution, and (3) verifying whether the causal process constraints are satisfied by the observed behavior of the component. In this way, it would be possible to decide if the components (re)used in the intended process work to achieve particular functional or nonfunctional goals.

3. Local Collaboration Consistency Notions using Concepts of UML-RT

In this section the local consistency notion is illustrated in the context of a collaborative behavior design. The utility of the methods is discussed in the context of UML-Real Time (UML-RT), which is an extension of UML. UML-RT places strong emphasis on UML collaboration concept, along with notions of capsules, ports, connectors, protocols, and protocol roles. Capsules are complex and potentially concurrent active architectural components. They interact with their environment through one or more signal-based boundary objects called ports. Collaboration diagrams describe the structural decomposition of a capsule class.

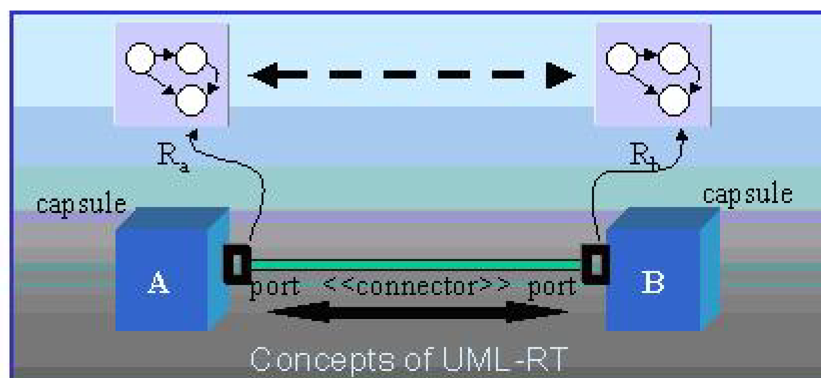


Figure 3: Components of UML-RT Role Models – The Local Perspective

Figure 3 illustrates the components of UML-RT role models. A port is a physical part of the implementation of a capsule that mediates the interaction of the capsule with the outside world—it is an object that implements a specific interface. Ports realize protocols, which define the valid flow of information (signals) between connected ports of capsules. A protocol captures the contractual obligations that exist between capsules. Because a protocol defines an abstract interface that is realized by a port, a protocol is highly reusable. Connectors capture the key communication relationships between capsules. Since they identify which capsules can affect each other through direct communication, these relationships have architectural significance. Collaboration diagrams, which capture architectural design patterns, are used to describe the structural decomposition of a capsule class. They use the primary modeling constructs of capsules, ports, and connectors to specify the structure of software components. Capsules use ports to export their interfaces to other capsules. The functionality of simple capsules is realized directly by finite state machines, whose transitions are triggered by the arrival of messages on the

capsule's ports. Capsules themselves can be decomposed into internal networks of communicating sub-capsules. The state machine and network of hierarchically decomposed sub-capsules allow the structural and behavioral modeling of arbitrarily complex systems.

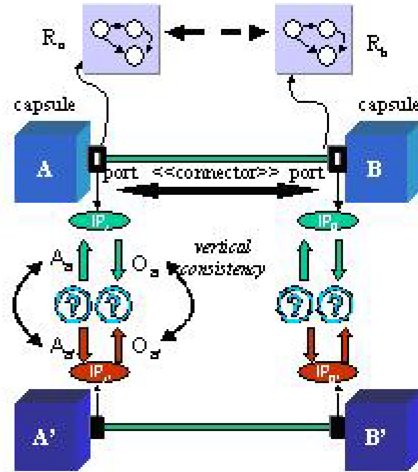


Figure 4: Safe Refinement for Vertical Consistency

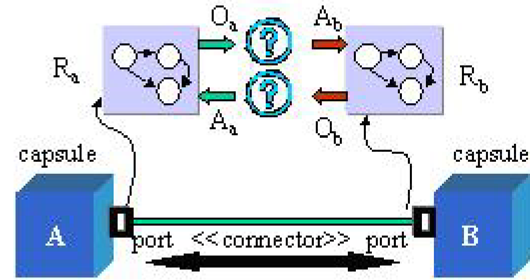


Figure 5: Horizontal Consistency

Given the formal models of assumptions and obligations, the horizontal consistency refers to the compatibility of assumptions and obligations of a port with respect to the obligations and assumptions of its peer capsule port, as shown in Figure 4.

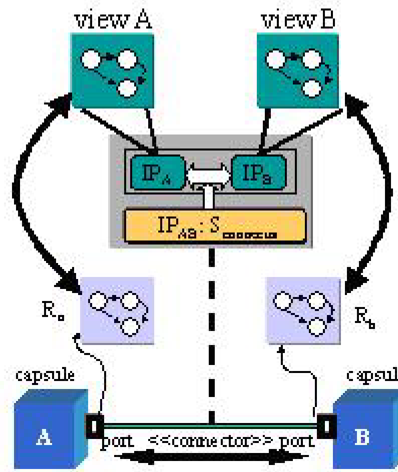


Figure 6: Connector Consistency

It is not always practical, however, to expect reused components to be consistent with the context they are embedded. In such cases designer develops a connector component that facilitates the mediation of interactions between incompatible ports. Figure 6 denotes a connector projected onto the viewpoints of the ports that it associates. Connector consistency entails (1) the projection of the connector protocol onto its assumptions and obligations with respect to each port and (2) horizontal consistency assessment between the projected protocol and each connected port

Safe refinement, as shown in Figure 5, refers the consistency of the original assumptions and obligations to the refined assumptions and obligations, respectively, to assure that the refined component includes all the behavior of the original role. The introduced consistency relation between the assumptions and obligations need to be transitive and closed under the horizontal consistency relation. Hence, as long as a component *A'* (abstract or concrete) safely refines another component *A,* it can safely substitute *A* without violating the originally commited assumptions and obligations.

4. Overview of Safe Refinement and Mutual Role

Compatibility of Interaction Policies

The certification model is based partly on the notion of augmenting an abstraction refinement framework with concepts such as reusability and interface compatibility. Formally representing the semantic dependencies among abstract components and reasoning for their correct realization form the basis of the certification model. Starting from the most abstract component, the certification process for abstract components consists of two actions. The first action determines the interaction policy compatibility of interaction roles and the composition protocol consistency if a connector protocol is declared, while the second ascertains whether the refined abstraction is safe.

4.1 Safe Refinement of Role Interaction Policy Assumptions

The refinement of individual role interfaces is based simply on the implication relation. Safe refinement of all roles of a particular component is essential for safe refinement of a component. Conditions for the refinement are devised such that the collaborating components would still be in agreement with the refined interaction policies of the new role interface. We consider two types of refinement, independent and simultaneous. Independent refinement occurs when a single role interface is modified. Before mutual compatibility of the refined role and its connected role interface can be achieved, one must demonstrate that the refined role interface interaction policies can at least engage in the expected interaction patterns, as dictated by level of abstraction prior to refinement. Formally, the constraints on the input patterns should be weaker, while the output constraints should be stronger. Hence, the refined role interface would be able to accept the expected input streams from its connection, as well as additional patterns required by the refinement, while the outputs of the refined interface should be acceptable by the input pattern assumptions of the connected role interface. Furthermore, due to internal non-determinism of the finite state abstraction of the protocol and interaction policies, those sets of message traces or streams that fail to be accommodated by the refined protocol must also shown to be failures in the original protocol. That is, it is essential to incorporate failure semantics. The above argument requires the refined interface to handle all the traces permissible in the more abstract policy specification. Furthermore, all input/output stream failures of the refined specification are also input/output failures of the abstract specification. It is required that the output stream of the refined policy to be equivalent to the output stream of the original interaction policy under independent refinement. This condition is sufficient to ensure that the output streams of the refined policy will be accepted by the receiving role interface. In simultaneous refinement, the constraint on the set of output streams of the refined policy is relaxed to incorporate additional output streams. This relaxation must ensure, however, that the receiving role interfaces and the composition protocol can consistently handle new sets of output streams

4.3 Mutual Role Interaction Policy Compatibility

The role compatibility method uses semantic analysis to ensure the congruity of connected role interface interaction policies. That is, interaction policies of connected interfaces are checked to ensure that they are consistent with each other. More specifically, collaborating components receive and send messages in a synchronized manner; that is, when a collaboration mate sends a message, the receiving party is shown to be in a state to receive and act upon the message. To this end, each role interface declares the set of input and output channels in terms of the provided (required) services and the input (output) events. The services refer to bi-directional method calls, whereas the events are unidirectional. The mutual role compatibility condition formulation requires successful matching of (1) the provided and required services and (2) the assumption and obligation constraints of each role with respect to its peer. That is, the required services of a particular role need to be a subset of the provided services of its mate. Furthermore, the obligations of a role with respect to its mate should be compatible with the assumptions of the mate component. More specifically, the output stream constraints, called obligations, of a particular role with respect to its mate need to match the trace constraints on or assumptions about the input trace stream constraints of the mate component. Conversely, the input stream trace constraints of the same role must comply with the output stream obligation constraints of its mate. Note that the output messages of a role interface are the input messages of its mate with opposite directions.

4.4 Composition Protocol Consistency

Recent studies in component development and verification, as well as validation processes, indicate that inconsistency is common throughout the development and evolution stages. Inconsistencies occur because the reusable components developed for different contexts have conflicting assumptions with respect to each other. Basically, in our approach, protocol consistency refers to the compatibility of the composition protocol with respect to each of the role interaction policies. A composition protocol incorporates all knowledge and rules regarding the means of coordinating and enacting role interface interactions among the environment, component under certification, and the associated peer components. In addition to identifying internal rules used solely to regulate and adapt mismatched roles, it also stipulates how and in which order the role enactment occurs. For each interaction policy being tested by the mutual compatibility diagnosis, the composition protocol consistency involves two stages. The first stage projects the protocol onto the interaction policy services of the component and the role interface services of the associated component or the environment. The second stage consists of the mutual compatibility check, which covers the projected interaction constraints imposed by the protocol and the interaction

policy constraints of the connected role. The composition consistency rules indicate three requirements. First, the collaboration mate must provide every service required by a role. Secondly, the output channel obligation constraints of the projected role must be compatible with the input assumptions constraints of the role interface under consideration. Finally, the output channel obligation constraints of the role under consideration must be compatible with the input assumption constraints of the projected role interface derived from the original protocol.

5. How Components are Intended to Work and Evolve: Causal Process Constraint Analysis

How components are configured and composed determines the organizational behavior that will emerge from the conglomerates of local interactions. Due to the state space explosion problem, deriving from the set of local interactions a causal behavioral model depicting the computational paths is very expensive. Hence, a new technique, which can be categorized as hybrid simulation verification, is developed. By synthesizing an intermediate analysis method, this technique combines the best of both worlds of run-time testing and verification. It uses test scenario execution to drive the component with a test engine that emulates the context of the component under certification. Test scenario execution also limits the generation of causal computation graphs to the computations consistent with the run-time behavior. This limited computation graph, called the causal behavior model, can be orders of magnitude smaller than the potential graph that could be derived from analyzing the local interaction policies of roles.

5.1 Causal Process Constraint Specification Approach

Local interaction policies consider those roles with pair-wise local connections that are associated with interaction constraints. They can be considered as local integration dependencies between constituent components implementing provided and required roles. In the context of the collaboration under certification, the cascading of these local dependencies leads to the emergent global integration behavior of the constituent components.

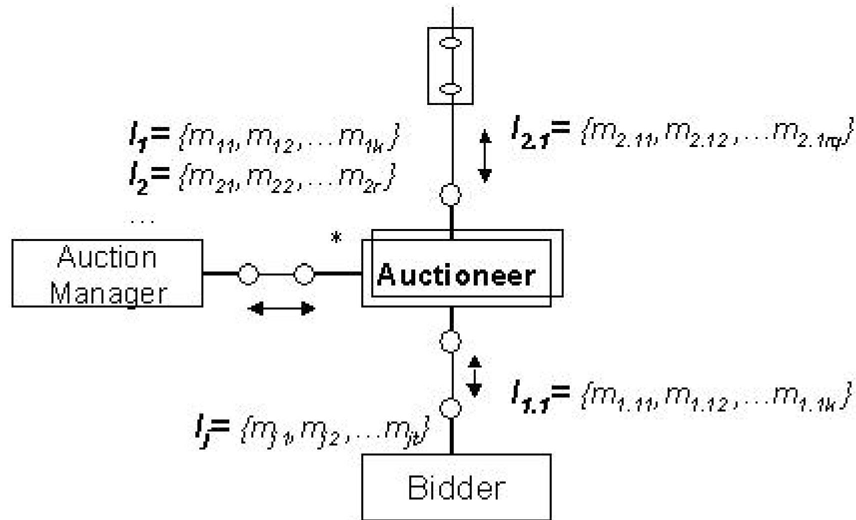


Figure 7: Causal Interaction Sequence Derivation from Collaboration Graphs

Such global integration constraints are considered under the horizontal integration aspect by utilizing mechanisms (i.e., UML collaboration graphs) to specify the processes underlying the computations. This section introduces a graph-oriented representation of causal event sequences based on the UML collaboration graphs. A simple constraint language is provided to enable the designer to constrain the expected causal process patterns and their evolution. Figure 7 depicts the interaction aspect of a collaboration that is utilized to capture causal processes that cross-cut the participant components. Each interaction constitutes a sequence of messages exchanged between a pair of connected roles. In figure 7, the overall interaction, **I=** $\{I_1, I_{1.1}, I_2, I_{2.1} \ldots I_j\}$, is decomposed into subinteractions such as $I_1, I_{1.1}, I_2, I_{2.1} \ldots I_j$, each of which constitutes message exchanges between peers. Note that UML collaboration interaction semantics provides a precedence relation (i.e., partial order) among the interactions. That is, the interaction identifiers that are distinguished in one integer form (i.e., *1, 1.1, 1.1.1, 1.1.2, 1.1.3* and *1, 1.2, 1.2.1, 1.2.2, 1.2.3* are two such sequences) constitute a sequence of interactions. The interaction sequences discussed above can be viewed as directed acyclic graphs, with the nodes representing the events and the arcs representing the causal relationships among them. This representation makes it possible to distinguish between roles that interact and those that do not.

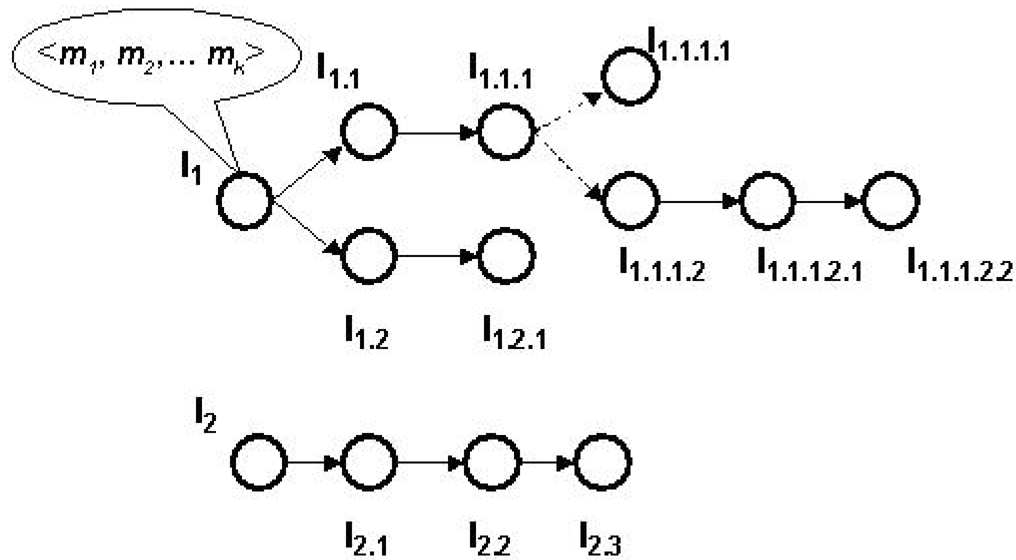


Figure 8: Causal Interaction Sequences

The interaction sequencing rules of UML collaboration graphs are represented in terms of *And-Or* graphs that are directed and acyclic, as shown in Figure 8. These graphs are called causal interaction graphs. Each causal interaction graph defines a single causal process. Each root-to-leaf sequence of interactions is called a causal interaction sequence. Causal processes are the expected computations that underlie services delivered by the collaboration. The processes constrain the overall computation space by imposing a specific causal pattern of interaction sequences. However, by themselves, causal processes cannot relate their occurrence to the existence of other causal processes and cannot represent the evolution of the collaborations. That is, operators and further constraints are needed to determine whether the occurrence of causal processes are temporally or logically related to each other, as well as if the causal processes evolve as intended. Such properties can be extended to check for interesting properties in the overall causal event streams. Below, the syntactic grammar of temporal, logical, and boolean operators on causal processes are presented:

Constraints ::= constraints ':' CPDSet
CPDSet ::= CPD CPDSet | CPD
CPD ::= (**always** |**sometimes**) Cpat |
(**introduce** | **prevent** | **guarantee**) CPat
CPat ::= CedP | (**until** | **unless**| **since**)
(CedP, CedP)
CedP ::= Ced **or** CedP | Ced
Ced ::= cedName **and** Ced |
cedName **implies** cedName | cedName

The constraints involve existential quantifiers (always, sometimes), evolution operators (introduce, prevent, guarantee), temporal operators (until, unless, since), and Boolean operators (or, and, implies). The semantics of the operators are beyond the scope of this short paper and will be presented elsewhere.

5.2 Overview of the Causal Process Constraint Analysis

Once a causal behavioral model is generated from the raw run-time interaction logs of a collaboration, constraint analysis is performed to determine if the run-time behavior satisfies causal process constraints. Due to space limitations we are omitting the causal behavioral model derivation and underlying causal event model here. In this section we briefly summarize constraint analysis mechanism.

As mentioned above, the constraints against which the derived causal behavioral model is checked impose temporal, evolution, and logical relationships on the observed causal processes. The execution of each single test scenario results in a set of observed causal process sequences. Constraint operators, such as always and sometimes, help the designer decide if the certain causal processes that are observed under all interaction scenarios enacted during the run-time monitoring process are invariant. On the other hand, the evolution operators involve comparisons of successive versions of the collaborative behavior under the same test scenarios. The temporal operators, as well as the boolean operators, impose a set of causal process patterns. The results of the application of operator rules to satisfiability vectors for each operand of the constraint tree are then combined to derive a global result for quantifier and evolution constraints.

Once the behavioral models for each exercised interaction scenario are derived and transformed into causal process sequences using the introduced event causality model, one can view the complete observed behavior of the component as a behavior matrix. The rows and columns of this matrix represent the exercised scenarios and the causal process sequences derived from the behavioral model with respect to the corresponding scenario. Formally, the behavior matrix is a two-dimensional *m x n* matrix, called *BM*. Here, *m* stands for the number of scenarios exercised, and *n* stands for the largest in cardinality of the set of causal process sequences derived as a result of each scenario. Each cell of the matrix is denoted as $P_{i,j}$, where *i* stands for the *ith* scenario and *j* stands for the *jth* sequence in the set, P_i, of causal process sequences associated with the scenario *i*. Hence, $P_{i,j}$ stands for the jth sequence of the causal process sequence set of the *ith* interaction scenario observed during run-time monitoring. After defining a matrix-oriented representation of the behavioral model, the next step is to test if the generated model satisfies constraints placed upon it. To determine if such constraints are satisfied, the approach requires a computational model to represent the constraints, as well as methods and reasoning mechanisms regarding their satisfiability. To this end, the constraint tree formalism is introduced.

Definition: A constraint tree, *CT*, is a finite set of one or more nodes such that there is one designated node *R* called the root of *CT*. The remaining nodes in $CT - \{R\}$ are also partitioned into n > 0 disjoint subsets $CT_1, CT_2, \ldots, CT_n$, each of which is a tree, and whose roots $R_1, R_2, \ldots, R_n$, respectively, are children of *R*. The leaf nodes that have out-degrees of zero are causal processes and the internal nodes that have out-degrees of one or more are the operators of the constraints.

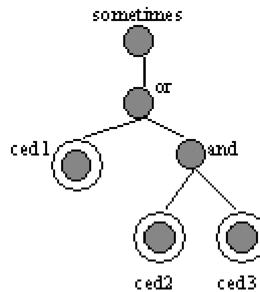


Figure 9: Constraint Tree

Constraint tree derivation: Consider the tree representation, shown in Figure 9, of the constraint *sometimes (ced1 or ced2 and ced3).* The constraint is parsed according to the constraint grammar to produce an abstract syntax tree for the in-order representation. Then, the tree is traversed to derive a pre-order representation of the constraint statement: *sometimes or ced1 and ced2 ced3*. Then the pre-order sequence is used to form the constraint tree in a top-down manner.

Causal Process Analysis Summary: Causal process analysis constitutes the methods that check for the satisfying of constraints represented in constraint tree formalism against the behavior matrix $BM = (P_{i,j})$ for $i = 1,2,\ldots m$ and $j = 1,2,\ldots,k$ as discussed above. The satisfying of a constraint for the recognized causal process sequences of a scenario is represented as a boolean vector. Hence, a two dimensional boolean matrix, $SAT = (b_{i,j})$ for $i = 1,2,\ldots m$ and $j = 1,2,\ldots,k$ is used for all interaction scenarios used in the certification. Each boolean vector for a given interaction scenario constitutes a row of the boolean matrix. We demonstrated the methods that check the satisfying of constraints against the derived causal process sequences of a particular scenario. The same method is applied for each scenario to generate the complete boolean matrix. Let *BMi* denote the *ith* row of the behavior matrix, which represents the set of causal process sequences captured as a result of the *ith* interaction scenario. Similarly, SAT_i denotes the boolean vector that stores the computed boolean value denoting the satisfying of each constraint by the sequences of the *ith* scenario. SAT_i is computed gradually, using the constraint tree and the set of causal process sequences associated with the BM_i. Using the constraint tree, the satisfiability vector is constructed from the bottom up. Initially, each leaf node of the constraint tree is associated with a boolean vector denoting the causal process sequences that satisfy the causal event description depicted at that node. As the rules associated with the operators at the internal nodes are applied to the vectors of its children the derived vectors are propagated upward in the tree to generate a new global boolean vector. The rules for each operator is derived using its semantics. The resulting boolean vector at the root of the constraint tree represents the satisfiability of the constraint with respect to the behavioral model derived after the execution of a given test scenario.

6. Related Work

A variety of research groups have recently investigated the compositional verification of collaboration designs. To prove certain behavioral properties about individual and composition of collaborations, some of these groups—such as Fisler and Krishnamurthi [2001]—employ finite state verification. Others, such as Engels, Kuster, and Groenewegen [2001], use CSP-type process-oriented formalisms to analyze object-oriented behavioral models for certain consistency properties. The interface automata concept introduced by Alfaro and Henzinger [2001] investigates light-weight formalisms that capture the temporal aspects of software with an approach the authors call optimistic view. Additionally, Butkevich, et al. [2000]

describe an extension of the Java programming language that supports static interaction conformance checking and dynamic debugging of object protocols. Their approach considers only the incoming requests and omits the output obligations of components, as well as global causal processes. Also, the UML model integration testing approach discussed by Hartman, Imoberdorf, and Meisinger [2001] illustrates the issues involved in collaboration testing.

7. Conclusions

In this paper we briefly overviewed and summarized an approach to consistency analysis and management for synchronization behavior in parameterized collaborations. Local (that is, role-to-role) synchronization consistency conditions are formalized and associated decidable inference mechanisms are developed to determine mutual compatibility and safe refinement of synchronization behavior. It has been argued that this form of local consistency is necessary, but insufficient to guarantee a consistent collaboration overall. As a result, a new notion of global consistency (that is, among multiple components playing multiple roles) is briefly outlined: causal process constraint analysis. Principally, the method allows one to: (1) represent the intended causal processes in terms of interactions depicted in UML collaboration graphs; (2) formulate constraints on such interactions and their evolution; and (3) check that the causal process constraints are satisfied by the observed behavior of the component(s) at run-time.

References

[Alfaro de L., T. A. Henzinger 2001].
"Interface Automata," Software Engineering Notes, vol. 26, no.5, pp.109-129.

[Beugnard A., J.-M. Jezequel, N. Plouzeau, and D. Watkins 1999].
"Making Components Contract Aware," IEEE Computer, vol. 32 no.7, pp. 38-45.

[Butkevich S., M. Raneco, G. Baumgartner, M. Young 2000],
"Compiler and Tool Support for Debugging Object Protocols," Software Engineering Notes, vol. 25 no. 6, pp.50-59.

[Dellarocas C. 1997].
"Software Component Interconnection Should Be Treated as a Distinct Design Problem," In Proceedings of the 8th Annual Workshop on
Software Reuse (WISR), Columbus, Ohio, March 23-26, 1997.

[Engels G., J. M. Kuster, L. Groenewegen. 2001].
"A Methodology for Specifying and Analyzing Consistency of Object-Oriented Behavioral Models," Software Engineering Notes, vol. 26,
no.5, pp.186-195

[Fisler K. and S. Krishnamurti 2001].
"Modular Verification of Collaboration-Based Software Design," In Proceedings of the 8th European Software Engineering Conference,
pp. 152-163.

[Lea D. 1999].
"Complex Components Create New Challenges," In Proceedings of the 9th Annual Workshop on Software Reuse.

[Weide W. B. and J. E. Hollingsworth 1992].
"On Local Certifiability of Software Components," In Proceedings of the 5th Annual Workshop on Software Reuse.

Fostering Early Development of Expert-Like Knowledge Structures in Computer Science Students

Paolo Bucci and Timothy J. Long
Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210-1277 USA

bucci@cis.ohio-state.edu, long@cis.ohio-state.edu
Phone: +1 614 292 5813
Fax: +1 614 292 2911
URL: http://www.cis.ohio-state.edu/~bucci/, http://www.cis.ohio-state.edu/~long/

Abstract

We propose that development of initial, expert-like knowledge structures should be an explicit learning objective for introductory programming courses. Such knowledge structures can grow with and direct computer science students as they proceed through their undergraduate training. To realize this objective, students will use a novel programming environment and language, tailored to the 18-year-old mind.

Keywords

CS1, CS2, knowledge structures, models of software

Paper Category: position paper
Emphasis: research in education

1. Introduction/The Problem

In virtually every field of learning, novices and experts differ in how they process information and solve problems. In physics, for example, an interesting study reported that competent novices (undergraduate physics students) categorized sample physics problems based on surface-level considerations, such as spring problems, or inclined-plane problems, whereas physics experts categorized the same problems according to the laws of physics [6]. Differences in the knowledge structures of novices and experts are thought to be central to this phenomenon [2]. The knowledge structures of experts allow processing of information by appeal to fundamental governing principles, as opposed to the more surface-level considerations embodied in the knowledge structures of novices.

We know of no studies documenting the novice/expert knowledge-structure phenomenon in the field of software (computer programming) at the undergraduate level. However, over fifty years of combined experience in teaching software development at all levels of the undergraduate curriculum convinces us that computer science graduates, as a whole, leave academia with woefully inadequate knowledge structures of software. We do not believe that this needs to be the case, and it seems to us that the real pity of this situation is one of lost opportunity. Almost all computer science students take a yearlong introduction to programming (often referred to as CS1 and CS2) beginning in the freshman year. If, during CS1/CS2, students formed initial expert-like knowledge structures of software, then the opportunities this would provide are truly exciting. Undergraduate students could, for example,

- deepen these structures and their basis in fundamental principles throughout their studies
- develop an ability to think in terms of these structures and thereby abandon, of their own volition, ad-hoc and "hacker" modes of software development
- hone recommended software skills and practices within the context of these structures
- make deeper connections between concepts appearing across the computer-science curriculum.

When we think in terms of what could be, it appears to us that failure to foster student development of initial, expert-like knowledge structures of software is one of the most serious shortcomings of current CS1/CS2 education. Furthermore, once this failure is identified explicitly, it is quite easy to recognize contributing factors. Regrettably, we hold that most programming languages and programming environments, as well as most textbooks, in popular use in CS1/CS2, are "part of the problem" [5,9,10]. In short, we believe that CS1/CS2 education is in need of a change in culture.

2. Background

Beginning in 1997, and with the support of FIPSE (P116B60717) and NSF (CDA-9634425), we began an ambitious project to revamp CS1/CS2 at Ohio State and West Virginia universities using a component-based software engineering model of software, referred to as the RESOLVE model [12]. Although we believe the new course sequence to be a considerable advancement in CS1/CS2 education, and despite encouraging assessment results [13] and the garnering of local and national awards [15], we have experienced a sense of frustration with the project, almost from the very start. At the surface level, frustration arises in the difficulty of appropriately expressing our model of software using the popular programming languages of the day, namely C++ and Java. In retrospect, this should not be surprising, as both languages are industrial-strength languages designed for industrial use. Bjarne Stroustrup, the inventor of C++, states "From the start, C++ was aimed at programmers engaged in demanding real-world projects. ... The nature of the C++ language is largely determined by my choice to serve the current generation of system programmers solving current problems on current computer systems. ... Trying to seriously constrain programmers to do 'only what is right' is inherently wrongheaded and will fail" [14]. Java white papers describe Java as "A simple, object oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high performance, multithreaded, dynamic language" [7] and state that "The Java programming language is designed to meet the challenges of application development in the context of heterogeneous, network-wide distributed environments" [8]. As suitable as these goals may be when considering experienced programmers, these languages obviously were not designed to convey, simply and elegantly, a clean model of software to the 18-year-old mind. Further, these programming languages appear to impede student learning of such basics as algorithmic thinking. Georgia Tech, for example, has abandoned the use of real programming languages altogether in its CS1 course for engineering students [11].

At a deeper level, our inability to appropriately convey our model of software through popular languages has inhibited student development of the desired knowledge structures. We believe this to be a serious and fundamental failure that must be corrected if our approach, or any approach, to CS1 and CS2 is to fully realize its potential in student learning.

3. Our Proposal

The RESOLVE model of software is based on fundamental and time-invariant principles that both pervade and transcend the field of computer science. Examples include system thinking, mathematical modeling, modular reasoning, information hiding, and abstraction. The RESOLVE model is accompanied by a programming language, called the RESOLVE language, used to express software artifacts according to the model. The model and the language have been painstakingly crafted to "fit together" and to incorporate and reflect fundamental principles as simply and as elegantly as possible. They have not been designed to compete with current trends in programming language fashion. We believe that by combining our current use of the RESOLVE model with use of the RESOLVE language, we can begin to address the fundamental issue of student development of expert-like knowledge structures of software, beginning in CS1/CS2. Thus, we propose to

- implement an innovative programming environment, called *SCW* (Software Composition Workbench), for student use of the RESOLVE language in CS1/CS2
- adapt our current CS1/CS2 materials/develop new materials to work in conjunction with *SCW*
- determine, through assessment, the extent to which beginning undergraduate students can develop initial, expert-like knowledge structures of software and the extent to which this can impact learning throughout their undergraduate computer science studies.

The *SCW* is central to our entire effort. Design of the *SCW* will be based on the idea of a conceptual editor [4] and will feature just the minimal amount of functionality necessary to develop programs according to the RESOLVE model and language. The self-explanatory interface will use visual structure, such as forms, to present the static elements of programs. This is in contrast to the usual method of keywords and indentation standards and will help students to distinguish between syntax and the elements that syntax is designed to express. Further, the interface will enforce a prescriptive pattern of interaction, through the use of wizards, so that the model of software will be unavoidably embodied in the very process of developing software through the *SCW*. (Although many programming environments are available already, almost all of them are complicated in their functionality and are not designed to foster student development of knowledge structures of software. DrJava [1], for example, is one effort to provide a more appropriate educational environment for Java.)

To further nurture student development of appropriate knowledge structures, students will be consciously engaged in metacognitive strategies in the classroom [3]. Principles and fundamentals will be introduced in homework readings, and the manifestation of underlying principles in the RESOLVE model and language, and in the *SCW*, will be reinforced constantly through classroom activities. Our existing course materials will be adapted for this integration of classroom activities with the *SCW*, and we will package and make available complete materials for in-class and at-home student activities.

Formative evaluation will be especially important for proper development of the *SCW*. In formative evaluations we will assess the ease of use of the *SCW*, its suitability for developing software, and its effectiveness in conveying the desired model of software. Summative evaluations will concern student development of expert-like knowledge structures of software. Desired knowledge-structure features will be identified and assessment instruments will focus on the extent to which each such feature is seemingly represented in student knowledge structures. Finally, for longitudinal evaluations, we will select a

small number of subsequent courses involving considerable software development. Following each such course, we propose to assess how students' previously acquired knowledge structures of software impacted their performance in the course, and how their knowledge structures evolved.

4. Conclusion

CS1/CS2 is fragmented with respect to programming languages, programming paradigms, and programming environments. This trend may worsen as competing companies aggressively market their technologies to the educational community. Within this context, our goal of fundamentally impacting CS1/CS2 education may appear quixotic, even if the technological elements of our approach are superior, which we believe them to be. Hence, we see evaluation as the key to our ultimate success. For us, development of knowledge structures that can grow with students as they proceed through the computer science curriculum is now an explicit learning objective for CS1/CS2. Through rigorous assessment demonstrating the extent to which this is possible and revealing measurable gains in subsequent student performance, we envision a CS1/CS2 community persuaded to embrace a fundamentals-based approach. At that time, the base of our curriculum will become the place where models are established, principles explained, and fundamentals practiced.

References

[1] Allen, E., R. Cartwright, and B. Stoler, "DrJava: A Lightweight Pedagogic Environment for Java", Proceedings of the 33rd ACM SIGCSE Technical Symposium on Computer Science Education, ACM Press, 2002, 137-141.

[2] Bransford, J., A. Brown, and R. Cocking, eds., How People Learn: Brain, Mind, Experience, and School, National Academy Press, Washington, D.C., 1999.

[3] Bruer, J., Schools for Thought: A Science of Learning in the Classroom, MIT Press, Cambridge, MA, 1993.

[4] Bucci, P., Conceptual Program Editors: Design and Formal Specification, Ph.D. Dissertation, Dept. of Computer and Information Science, Ohio State University, 1997.

[5] Bucci, P., T. Long, and B. Weide, "Do We Really Teach Abstraction?", Proceedings of the 32nd ACM SIGCSE Technical Symposium on Computer Science Education, ACM Press, 2001, 26-30 (http://www.cis.ohio-state.edu/~weide/sce/papers/index.html).

[6] Chi, M.T.H., P. Feltovich, P., and R. Glaser, "Categorization and Representation of Physics Problems by Experts and Novices," Cognitive Science 5, 1981, 121-152.

[7] Gosling, J., The Java Language: an Overview, 1995 (http://java.sun.com/docs/Overviews/java/java-overview-1.html).

[8] Gosling, J. and H. McGilton, The Java Language Environment: A White Paper, 1996 (http://java.sun.com/docs/white/langenv/).

[9] Long, T., B.Weide, P. Bucci, D. Gibson, M. Sitaraman, and S. Edwards, "Providing Intellectual Focus to CS1/CS2," Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education, ACM Press, 1998, 252-256 (http://www.cis.ohio-state.edu/~weide/sce/papers/index.html).

[10] Long, T., B. Weide, P. Bucci, and M. Sitaraman, "Client View First: An Exodus from Implementation-Biased Teaching", Proceedings of the 30th ACM SIGCSE Technical Symposium on Computer Science Education, ACM Press, 1999, 136-140 (http://www.cis.ohio-state.edu/~weide/sce/papers/index.html).

[11] Shackelford, R., Introduction to Computing and Algorithms, Addison Wesley, 1998.

[12] Sitaraman, M. and B. Weide, eds., Component-based software using RESOLVE, ACM Software Engineering Notes 19, 1994, 21-67.

[13] Sitaraman, M., T. Long, B. Weide, E. Harner, and L. Wang, "A Formal Approach to Component-Based Software Engineering: Education and Evaluation," Computer Science Education, Swets & Zeitlinger Publishers, to appear.

[14] Stroustrup, S., The Design and Evolution of C++, Addison-Wesley, 1994.

[15] http://www.cis.ohio-state.edu/~weide/sce/now/awards.html

Subsetting Language Elements in Novice Programming Environments

Peter J. DePasquale
Department of Computer Science
Virginia Tech
660 McBryde Hall
Blacksburg, VA 24061, USA
pjdepasq@vt.edu
Phone: 540 231-5914
Fax: 540 231-6075
URL: http://csgrad.cs.vt.edu/~pjdepasq

Abstract

This author has concerns of two issues with respect to student instruction in CS1 courses (1st semester programming courses). The first is that novice programming students need not be exposed to professional strength programming tools so early on in their curriculum. Many of these environments have highly complex interfaces and include a plethora of features and tools (including profilers, project management features, etc.) which are not needed initially by novice programmers. In the author's experience, many times these additional "features" only confuse and frustrate the student. Secondly, I argue that a novice programmer's environment should grow in lockstep with their in-class experience. Novice programmers generally are introduced to simple, stand-alone concepts that can be easily digested and understood. Further concepts are then added in an iterative manner as additional language elements are introduced. In essence, what educators are doing in the classroom is subsetting a programming language and introducing successive supersets to the student participants. The programming environment novice programmers utilize should be modeled after the pedagogical style used in the classroom.

Keywords:

Novice Programmers, Programming Environments, Language Subsetting

Paper Category: Position Paper
Emphasis: Education

1. Introduction

Within the computer science community, there has been an examination of programming languages and integrated development environments (IDEs) for use in conjunction with teaching CS1 courses (introductory programming courses generally follow the syllabus for ``CS1" within the ACM/IEEE-CS curriculum of 1991 [TUCKER]) to novice programmers. [FELLEISEN, KOLLING96, GRIES, DEEK, PERIS]. Most striking about these teaching / learning environments is that for the most part they seek to follow one of the following approaches:

- the creation of a simplified language in which the student learns to program (Logo, Euphoria), with the idea of subsequent migration to a more mainstream (commercially successful) language with advanced features;
- the simplification of the development environment in order to simplify the code development process [HOLT]; and/or
- the provision of improved tools to support code development and visualization of the authored code [CROSS, KOLLING].

Ironically these approaches seemingly lack a similarity to the methods by which we teach introductory programming in the classroom. When first introduced to a programming language, novice programmers generally are introduced to simple, stand-alone concepts that can be easily digested and understood. Further concepts are then added in an iterative manner as additional language elements are introduced. In essence, what educators are doing in the classroom is subsetting a programming language and introducing successive supersets to the student participants. By semester's end, the students gain exposure to a majority of the components of the programming language and usually have the ability to author programs given only problem requirements.

As they learn to implement (code) programs, students use either a simplified programming environment (mentioned earlier) or a programming environment with an interface and compiler designed for professional application development. While there are many commercial professional programming environments, there are few learning environments for novice programmers.

Are these CS1 students developing large-scale applications? Do they need full strength debuggers, or project management tools (such as source code control tools and profilers)? In many cases, these extraneous tools are never covered as part of the syllabus in novice programming classes and their presence in the programming environment only serves to confuse novice programmers (if discovered at all). To make matters worse, the courses that follow CS1 may assume that these tools were covered as part of the learning experience.

2. The Problem

This author argues that novice students do not need complex tools and that the introduction of a highly complex structured programming environment to these students is detrimental to a clear understanding of their first programming language and programming in general. In essence to require students to learn both a programming language and the command language of the implementation environment is akin to learning French and Spanish at the same time. A novice programmer's first development environment should be simple to use, provide clear consistent error and warning messages (which do not befuddle the student), and should grow in lockstep with the pedagogical methodology used in the classroom.

3. The Position

In support of the author's dissertation, a prototype programming environment for use by novice students is being constructed. The environment has three

distinct goals in mind:

- **Support for pedagogy** - permits the course instructor to create and propagate language subsets to the student's programming environment. In this fashion the student can be forced to solve a programming problem utilizing only material that has been covered (or conversely, target those language constructs which the student(s) are struggling with).
- **Simplicity in the interface** - Microsoft Visual C++ (version 6.0) [MICROSOFT] initially appears with 14 buttons, 4 drop-down menus and over 75 menu items (by default). The author argues (based on his experience as a CS1 instructor and teaching assistant) that this environment's interface has a observed overwhelming effect on novice programmers who are utilizing the environment in their first several sessions. The prototype environment greatly reduces the number of graphical user interface (GUI) components needed to support the development and execution of the programs typical to this user-base.
- **Help messages that help** - provides useful error messages and help displays which are novice-readable and help to elucidate the true problem/solution. Messages such as the following (taken from an actual programming environment used by CS1 students at Virginia Tech) are not helpful to novice programmers:

```
test.obj : error LNK2001: unresolved external symbol "void __cdecl test(float)" (?test@@YAXM@Z)

Debug/test.exe : fatal error LNK1120: 1 unresolved externals.
```

4. Justification

Through the use of this prototype, the author plans to explore the following hypotheses:

- the application of subsets to the programming language used by novice students can yield quantifiable pedagogical gains; and
- the reduction of complexity of a programming environment's interface (used by novice) students can yeild quantifiable pedagogical gains.

Obviously, this is a work-in-progress and lacks quantitative data to prove or disprove the hypotheses at this time.

The author has previously run a pilot study to monitor the reactions to a professional strength programming environment on novices programming their first project(s). The results of the study clearly indicated the need for a simplified interface, clearer error messages and more straightforward in-application help dialog windows.

The author plans to initiate an experiment in the fall of 2002 utilizing the prototype programming environment in a CS1 course offering at Virginia Tech. While the experiment design is still taking shape, the basic plan is to divide one section of students (roughly 150 students) into three demographically equal groups and study their academic results using one of three programming environments:

- Microsoft Visual C++ (version 6.0),
- the prototype programming environment without the use of subsets; and
- the prototype programming environment using at least 5 subsets.

5. Related Work

The introduction to this position paper points out several related works that are currently available. While the exploration of programming enviroments and support applications for novice programmers and students alike has been ongoing for more than 30 years, there are several worth a closer look. Theses were selected as the best representative samples of related works that are currently in use today.

BlueJ

BlueJ is an integrated Java software development environment which contains an editor, compiler, debugger and other assorted development tools. What makes BlueJ unique is the graphical representation of executing objects with which a user can interact. That is, the user can instantiate objects (in the runtime environment, see them represented graphically and then send messages interactively to them (call methods contained within the object) or have objects interact with each other.

BlueJ attempts (with a good deal of success) to change the level of abstraction that users face while using their environment to produce Java programs. As with many programming environments, the users must deal directly with the file level while editing source code. BlueJ seeks to modify the file view to one of an object view, where the users can codify, compile and manipulate objects directly (interactively). More over, by having the BlueJ programming environment interact directly with the Java runtime environment, some tricky linguistic issues can be avoided. Specifically, a main method in Java must be of the form:

```
public static void main (String[] args) {

}
```

To cover this statement at the beginning of a course (so that one understands where their program begins) may involve a discussion on visibility modifiers, static methods, arrays and void return types. Creation and instantiation of Java objects directly in the programming environment (with the help of the runtime system) avoids the need to implement a main method and increases the level of abstraction to the user.

One of the arguments against the BlueJ environment is that it focuses less on program development and more towards the visualization of the object-oriented paradigm. Additionally, the environment abstracts the object-oriented paradigm to such a high degree that one could argue that the programming environment abstracts away the concept of a single method of execution of the resulting program.

DrScheme

DrScheme is an ongoing project of the Programming Languages Team at Rice University. DrScheme is a Scheme programming environment targeted toward CS1 students who are beginning to learn how to program.

DrScheme provides a graphical user interface to a Scheme programming environment. The DrScheme approach however, is somewhat unique in that it utilizes a tower of Scheme subsets within the programming environment. There are only four such subsets in the DrScheme environment: *Beginning Student*, *Intermediate Student*, *Advanced Student* and *Full Scheme*, each of which are coded into the application by the DrScheme developers and can not be modified. Users can select a language level (Beginning Student is the default level) prior to developing a Scheme program in the environment. The advertised design of Dr.Scheme is that each of the levels matches the natural introduction of Scheme to novice students.

While DrScheme is one of the most recent programming environments to utilize layers (or towers of the language as referred to in the documentation), this author has been unable to location any published empirical evidence on the efficacy of improving student learning through their environment.

ProgramLive

ProgramLive is a multimedia textbook authored by David and Paul Gries for learning Java and object-oriented programming. The ProgramLive software graphically reproduces a book paradigm for the students (complete with chapters, sections, page tabs and a spiral bound edge). ProgramLive contains narrative presentations using recorded audio tracks to present programming concepts to the learner.

Other features of the Gries effort include in-text quizzes, a glossary linked to the chapter prose by hypertext links, an index, and the ability for the learner to set a bookmark and return to it at a later point. One of the most interesting features is the code tool section, which is a simulated program development environment. The code tool features a panel of Java source code which is animated to a voice narration and a textual and graphical presentation in a second window. Users can enter this section to observe code development complete with a narrative track which can be stopped and repeated.

An more complete analysis of the ProgramLive system was not possible on the demonstration disk made available for review. Attempts failed to obtain a complimentary copy for examination purposes. It is hoped that complete review can be undertaken in the near future as the ProgramLive environment seems to have promising aspects to it.

GRASP

The GRASP (Graphical Representations of Algorithms, Structures, and Processes) project at Auburn University is a software package that parses Java, C, C++, Ada or VHDL input, and produces control flow and data structure diagrams of source code input. GRASP output aims to assist in the comprehension of a program flow of control during execution. This is accomplished by overlaying the coding display with a series of graphical notation objects (lines, boxes, ellipses) to denote the relevant portions of the input (source code) which affect the execution path.

GRASP utilizes a pre-installed compiler (though a command line interface) to provide an environment in which a student can author and visualize their programs. The graphical notation that GRASP provides can be generated without the use of an installed compiler. That is, the compiler is required only using the environment in which to compile the code. GRASP is implemented with a language recognizer to support the creation of the notation.

6. Conclusion

Novice programmers must learn a great deal to become proficient (even in the most rudimentary sense) in computer programming. This includes the syntax and the semantics of the language they are striving to learn; and the concepts and actions which lead to a meaningful executable program. Additionally, the command language of the programming environment into which software is to be written (which in itself has its own language that the student must master to properly operate it); and in some cases, the language of the source code editor must also be mastered.

This author believes that through the application of subsets to the programming language it can be shown that novice programmers will yield better academic results (higher grades), and have a reduced time-on-task (due to utilizing a programming environment which does not assume the learner is familiar with the breadth of the language). An additional side-effect may be shown that novice students may feel less intimidated by a less-complex and more intuitive interface. In conclusion, all signs point towards this type of programming environment being able to positively affect students in a CS1 programming course.

The author would be interested in discussing the issues of interface complexity (in programming environments) as well as the notion of applying subsets to a novice's programming language. This author believes that by reducing the complexity and applying controlled access to language constructs we may find improved student understanding of programming as well as lessening the anxiety of a CS1 course.

References

[CROSS] James H. Cross. Introduction to the CSD - Overview. Internet WWW page, at URL: <http://www.eng.auburn.edu/grasp/documentation/intro_to_CSD/c_index.html> (last accessed 5/1/2001).

[DEEK] Fadi P. Deek and James A. McHugh. A review and analysis of tools for learning programming. In Proceedings of the 10th World Conference on Educational Multimedia and Hypermedia and on Educational Telecommunications, pages 251-256, 1998.

[FELLEISEN] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. Functional programming: The DrScheme Project: An overview. ACM SIGPLAN Notices, 33(6):17-23, June 1998.

[HOLT] Holt Software Associates Inc. Ready to Program. Internet WWW page, at URL: <http://www.HOLTsoft.com/ready/home.html> (last accessed 5/1/2001).

[GRIES] David Gries and Paul Gries. ProgramLive: A Multimedia, Java-based Livetext on Programming. Internet WWW page, at URL: <http://www.datadesk.com/ProgramLive > (last accessed 5/1/2001).

[KOLLING] Michael Kölling. Why BlueJ? An introduction to the problems BlueJ addresses. Internet WWW page, at URL: <http://bluej.monash.edu/why/why.html> (last accessed 5/1/2001).

[KOLLING96] Michael Kölling and John Rosenberg. An object-oriented program development environment for the first programming course. In Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education, pages 83-87. ACM, February 1996.

[MICROSOFT] Microsoft Corporation. Microsoft Visual C++, version 6.0.< http://msdn.microsoft.com/visualc/default.asp> (last accessed 5/1/2001).

[PERIS] R. Jiménez-Peris, C. Pareja-Flores, M. Patiño Martínez, and J.A. Velázquez-Iturbide. Towards truly educational programming environments. In

Tony Greening, editor, Computer Science Education in the 21st Century, pages 81-111. Springer-Verlag, 2000.

[TUCKER] A.B. Tucker et al. A summary of the ACM/IEEE-CS joint curriculum task force report computing curricula 1991. Communications of the ACM, 36(6):68-84, June 1991.

Testing in Undergraduate Computer Science: Test Cases Need Formal Specifications

Allen Parrish
Department of Computer Science
The University of Alabama
Tuscaloosa, AL 35487-0290

parrish@cs.ua.edu
Phone: 205-348-3749
Fax: 205-348-0219
URL: http://cs.ua.edu/~parrish

Abstract

Software testing has typically received scant attention in most computer science undergraduate curricula. Even when testing is taught, we have observed little success in getting students to appreciate the motivation for expending effort in validating their programs. There are a number of reasons for this lack of success. One of the problems is that students don't necessarily see the bigger picture in the relatively simple typical introductory programming context. We believe that one approach to solving this problem is to concede the motivation problem and address issues of form first. Formal notations eliminate a lot of ambiguity often associated with introductory testing discussions that are not typically well-motivated to begin with. In this paper, we discuss some simple approaches to writing formal specifications for test cases.

Keywords

Software testing, formal specifications, computer science education

Paper Category: Position Paper
Emphasis: Education

1. Introduction

Most introductory students don't do well with testing because they fail to appreciate the real need for it. This is not helped by the lack of attention paid to testing in most introductory textbooks. While there usually is a short discussion devoted to testing, it is typically vague and ill-motivated. It is hard for students to appreciate the need for testing boundary or extreme conditions, for example, until they have seen a number of situations where defects have arisen that are consequences of such conditions.

We claim that this lack of motivation is truly inherent. Rather than addressing it directly, a better approach might be to teach students the "form" of testing in an unambiguous way, without trying to emphasize the substance. The idea is that if students understand the form first, the substance of actual test design can follow on top of a solid foundation.

As such, we propose the idea of formally specifying test cases using (mostly) simple notations. An important idea here is to capture the concept that test cases are (in their simplest form) discrete events. Students should understand that testing consist of *N* runs of the program, where each run involves executing a test case. Thus, a *test set* is a set of *N* test cases. Each test case is either *revealing* or *unrevealing* of a defect. A revealing test case should be followed by some kind of corrective action that would then be followed by re-running the entire test set. The process concludes when all test cases in the test set are unrevealing.

At the introductory level, there is really two types of testing: "value testing," which involve providing actual input values to programs and "function testing," which involve executing actual sequences of program functions (frequently method sequences) in various ways to test classes and other reusable components. We find that virtually all discussion of testing in introductory textbooks is with regard to value testing. Function testing is largely ignored, which we believe is a product of the lack of clarity in the software engineering literature regarding how to discuss testing classes and reusable components in general.

We propose formalizing the notion of a "value test case" as an input/output vector and a "function test case" as either a function sequence or pair of sequences. These two ideas are discussed individually in Sections 2 and 3 below. Section 4 contains a brief summary and conclusion.

2. Formalizing Value Test Cases

As an example of a typical introductory textbook treatment, we consider the testing discussion from [3]. The principal discussion in [3] consists of a case study of a simple application that computes "weekly pay" from some inputs (employee type (full-time or part-time), hourly rate, hours worked). The discussion in [3] then seeks to motivate the idea of equivalence classes based on the fact that overtime is treated differently for full-time employees; thus, 0 to 40 hours constitutes one equivalence class, while over 40 hours constitutes a second equivalence class. It is then argued that test data for this program should test the program's behavior under boundary conditions and extreme conditions relative to these equivalence classes. It concludes with the following characterization of good test data for this program:

- Employee type: 1; Hourly rate: 10; Hours worked: 30 and 50.
- Employee type: 1; Hourly rate: 10; Hours worked: 39, 40 and 41
- Employee type: 1; Hourly rate: 10; Hours worked: 0 and 168
- Employee type: 2; Hourly rate: 10; Hours worked: 30 and 50.

We feel that this discussion would benefit from a simple formal characterization of the testing process along the lines described above. In particular, this particular program should be viewed as a function from an input vector (employee type, hourly rate, hours worked) to an output vector (weekly pay). Each test case is an (input vector, output vector) pair. The characterization of "good test data" could then be represented as a set of test cases, as follows:

- (<1, 10, 30>, <$300>)
- (<1, 10, 50>, <$600>)
- (<1, 10, 39>, <$390>)
- (<1, 10, 40>, <$400>)
- (<1, 10, 41>, <$420>)
- (<1, 10, 0>, <$0>)
- (<1, 10, 168>, <$2960>)
- (<2, 10, 30>, <$300>)
- (<2, 10, 50>, <$500>)

Representing the test data as a set of nine discrete test cases would reinforce the idea that the program must be run nine separate times (if the idea is reinforced that each test case corresponds to an execution of this program). Test cases using this formalism each explicitly includes the correct output, so the idea of running the program and each time checking the output against the test case is explicitly factored into this model. Because the model provides clarity to the notion of a test case as a discrete entity, it thereby provides clarity to the idea of a test case being "revealing" or "unrevealing" of a defect. The idea of repeating test cases until no test case reveals a defect can also be unambiguously explained in this model. None of these concepts are discussed in [3].

While the idea of presenting discrete test cases can be adopted without using formal notation, the notation is important because of the complexity of input and output vectors even in trivial programs. We once gave an assignment to compute sales commissions early in an introductory programming course. The program was to iteratively read sales amounts and generate commissions based on some simple rules. This was to be done repeatedly until the appearance of a sentinel value, at which point the program was to print some simple totals (total commissions, average commission, number of salespeople).

We instructed the students to develop a test plan consisting of test cases of the form *({<s1, c1>, <s2, c2>,...<sn, cn>}, {t, a, n})*, where *<si, ci>* was the input sales and output commission for salesperson *i*, and *{t, a, n}* were the three totals values. Each such tuple *({<s1, c1>, <s2, c2>,...<sn, cn>}, {t, a, n})* was a test case. This makes explicit the idea that one run of the program was insufficient even if it tested for a lot of different sales amount values, because it did not consider the dimension of how many salespeople to compute in a given run. Formalizing the test cases forced the students to realize that there were two dimensions in the test design for this problem. It seems unlikely that an informal discussion would have ever made this point convincingly. But once the students realized that they had to design more than one tuple of the form above, they realized that the testing process had these multiple dimensions.

In short, we feel that specifying the formal tuple for each test case causes the students to think more concretely about testing than simply having an abstract discussion about equivalence classes and boundary conditions. Of course, the formalism itself does not provide any semantic content about how to choose test cases. However, boundary conditions can be expressed more

concisely and unambiguously in text using the formalism.

One current limitation is that this concept seems to only fit well with traditional imperative programs that run to termination. A somewhat different notational paradigm may be more appropriate for capturing inputs for event-driven programs. This could be a potentially interesting topic for discussion.

3. Testing Classes

Introductory textbooks seem to completely ignore the problem of unit testing. Such an omission might make sense in traditional procedural programming where programs do not get more complex than a few procedures. But with object-oriented programming, classes are the principal program unit, and classes are typically complex enough that unit testing is necessary. This is even more important in cases where a class is produced as a reusable component that is independent of any particular application.

We deal with this problem by specifying class test cases as method sequences. Moreover, we define two types of test cases: *state inspection* test cases and *state comparison* test cases. For simplicity, assume that any class has three types of methods in its interface: *constructors*, *modifiers* and *observers*, with the obvious definition of each. The state inspection test cases are of the form *CM*O+* (i.e., constructor, followed by zero or more modifiers, followed by one or more observers). These sequences simply produce the values returned by observer methods, which are capable of being observed (often built-in types). State comparison test cases are of the form *(CM*, CM*, comp)* or *(CM*O, CM*O, comp)*, where "comp" is a comparison operator. The idea with state comparison test cases is to use the comparison operator to return the single results produced by the two method sequences; such results are either objects of the class under test (produced by the sequence of modifier methods in *CM**), or the results returned by an observer method (i.e., by method *O* in *CM*O*).

By giving students these templates, they have a syntactic notion of a test case, just as with the input/output vectors in the previous section. Students then are asked to write down each test case as a pair (method sequences, result). Sequences are then written using standard "dot" notation (e.g., Stack().push(10).push(20).top). So a state inspection test case might be written (Stack().push(10).push(20).top, 20), while a state comparison test case might be written ((Stack().push(10), Stack().push(10).push(20).pop(), equal), true).

The JUnit [1,2] unit testing framework for Java has also helped to address this problem, in that it provides a way to easily build and execute unit tests on Java classes. Each test case is captured as a method within this framework, which contributes to a reasonable formal framework to capture tests as discrete cases. JUnit has been used in the context of undergraduate computer science education [4], and we feel that it holds great promise in this regard.

4. Conclusion

Little progress has been made over the past 20 years in teaching software testing concepts early in the undergraduate computer science curriculum. Introductory computer science textbooks do not seem to have changed significantly in this regard. We feel that one of the problems is the vagueness with which testing concepts are typically discussed. In particular, the notion of a test case as a discrete entity corresponding to a run of the program or an execution of a sequence of methods is not emphasized to students. We feel that one solution to this problem is the use of a formal framework as a basis for pedagogical discussion, much in the way that research discussions are aided by the use of such frameworks [5]. Such frameworks make the concepts associated with testing more precise, and do not typically require the use of mathematics more complex than basic sequences, sets, functions and tuples.

References

[Beck97] Beck, K. and E. Gamma, JUnit Open-Source Testing Framework, www.junit.org.

[Beck98] Beck, K. and E. Gamma. "Test Infected: Programmers Love Writing Tests." *Java Report*, Volume 3, Number 7, 1998.

[Lambert02] Lambert, K. and M. Osborne, *Java: A Framework for Programming and Problem Solving,* Brooks/Cole, 2002.

[Noonan02] Noonan, R. and R. Prosl, "Unit Testing Frameworks," *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, pp. 232-236.

[Parrish93] Parrish, A. and S. Zweben, "Clarifying Some Fundamental Properties in Software Testing," *IEEE Transactions on Software Engineering*, vol. 19, no. 7, July, 1993, pp. 742-746.

Capturing the Reference Behavior of Linked Data Structures

Gregory Kulczycki
Murali Sitaraman
Clemson University
gregwk,murali@cs.clemson.edu

William F. Ogden
The Ohio State University
ogden@cis.ohio-state.edu

Joseph E. Hollingsworth
Indiana University Southeast
jholly@ius.edu

Copyright ©2002 by the authors. All rights reserved.[1]

May 3, 2002

Abstract:

The use of references in programs can complicate reasoning and break modularity. Language designers need to identify the predominant reasons for employing references in high-level languages and offer safer and more manageable alternatives. One area where references are used is in the implementation of linked data structures such as lists and trees. This paper presents a concept--*Location_Linking_Template*--that captures the behavior and performance benefits of traditional pointers and is especially useful for implementing linked data structures. The concept is used in conjunction with a simple component design methodology to enable programmers to isolate the complex reasoning about reference behavior to a small portion of an overall program while preserving program modularity. Through this concept, the conceptual model of pointers is presented in a new way that students and programmers may find easier to grasp than traditional explanations of pointers.

Introduction

References in high-level languages are useful for implementing software efficiently, but the mechanisms that provide reference behavior in popular programming languages are subject to abuse and routinely break modular reasoning.

A partial solution to this problem involves an efficient mechanism for data movement and parameter passing that does not introduce aliasing, such as swapping [3,17]. Swapping eliminates the need to reason about aliasing for the vast majority of components in a software system, but there are a few situations where the ability to use aliasing is necessary for efficient computing despite the added complexity. In particular, the implementation of linked data structures such as lists, trees, and graphs requires the use of aliasing for efficiency. The *Location_Linking_Template* introduced in this paper provides an alternative to implementing these data structures with traditional pointers.

Section 2 of this paper describes the problem that has continued to face language designers since references were first introduced into high-level languages. It explains the traditional benefits of references and also discusses why they present problems. Section 3 offers a two part solution to the problem. The first part--providing mechanisms for efficient data movement that do not introduce aliasing--is discussed only briefly and the reader is referred to other papers for an depth explanation. The second part is the main focus of this paper. It involves a general strategy for component construction centered on the Location_Linking_Template, a component that effectively captures the reference behavior of linked data structures. Section 4 gives an informal introduction to the component and section 5 covers its formal specification in some

detail. Finally, section 6 gives an example of its use in the implementation of a *List* component. Appendix A contains the specification for Location_Linking_Template, Appendix B contains the List_Template and its realization, and Appendix C contains a realization for List_Template that presents possible syntactic sugar for the operations in Location_Linking_Template.

Problem

Ever since their introduction into high-level languages over 30 years ago, there has been a steady stream of warnings about the dangers of references. Tony Hoare called their introduction ``a step backward from which we may never recover" [4] and presented a detailed list of reasons for that claim in his paper on recursive data structures [5]. Around the same time, Dick Kieburtz continued to explain why we should be programming without reference variables [8], and Steve Cook published his seminal paper on the soundness and relative completeness of Hoare logic [2] in which he identified aliasing of arguments to calls (even in a language without reference variables) as the key technical impediment to modular verification. There have been recent papers [13,14] showing how it is technically possible to overcome such problems, but apparently only at the cost of even further complicating the programming model that a language presents to a software engineer.

Benefits of References

Despite these warnings, references continue to be used routinely in today's popular programming languages. Why? One reason may be the notorious claim that ``Any problem in Computer Science can be solved with another level of indirection." [2] This catchy saying has become a rule of thumb for many programmers, and since references represent a prime source of indirection in computer programs, the adage seems to imply that references are the primary mechanism for solving tricky programming problems. But vague claims such as this offer little help in deciding what specific problems references help to solve. Certainly at higher levels of abstraction any so-called indirection in a system will be a product of the system design and would have nothing to do with whether the implementation language supported references.

Specific areas in which high-level imperative languages benefit from references almost always involve efficiency--efficient data movement, efficient parameter passing, and efficient implementations of linked data structures. [3] Reference variables provide efficient data movement by allowing a programmer to copy an object's reference rather than its value, since value copying is expensive for non-trivial objects. Call-by-reference parameter passing is more efficient than call-by-value for the same reason. When implementing linked data structures, references allow a programmer to insert or remove objects efficiently from arbitrary locations in the structure.

Problems with References

References tend to be confusing for students to learn and difficult for programmers to use. This is partly why Andrew Koenig and Barbara Moo argue for a rethinking of how C++ is taught in [9]. Koenig's introductory C++ textbook introduces vectors, strings, and structures before even mentioning pointers because ``pointers are a slippery subject to master, and beginners have a much easier time with the value-like classes."

The difficulty in reasoning about references informally is no doubt related to the difficulty in reasoning about them formally. The problem is not with references themselves, but with the aliasing they can introduce. A simple description of the problem may be found in [6]. Consider the Hoare triplet

{x = true} y := false {x = true}

At first glance it seems trivial to prove that if *x* is true before the assignment to *y*, *x* will still be true after the assignment to *y*. But this reasoning is sound only if *x* is not aliased to *y*. If aliasing is permitted it forces the programmer to consider that modifying a variable *y* may potentially modify any other variable of the same type. If these other variables are not local to the procedure or even the facility where the modification occurred, the programmer must possess knowledge of the global program state in order to understand the effects of the change. When a programmer must reason globally about a change that occurred locally, modular reasoning becomes impossible.

References are beneficial because they allow us to implement software efficiently, but aliasing can complicate reasoning and break modularity. Is there a way to get the efficiency of references without the reasoning complexity commonly introduced by aliasing? The next section attempts to answer this question.

Position

The section above describes three distinct areas of imperative programming languages that can benefit from the efficiency of references: data movement, parameter passing, and the implementation of linked data structures. Can we get the efficiency of references without the reasoning complexity that aliases introduce in each of these areas? For data movement and parameter

passing the answer is yes, because there are mechanisms that can move data efficiently without introducing aliases. In particular, the swapping operation is described in [3] as an efficient alternative to assignment that does not introduce aliasing, and call-by-swapping is described in [17] as an efficient alternative to call-by-reference that does not introduce aliasing.

Unfortunately, the answer to the aliasing question as it applies to implementations of linked data structures is no--we cannot get efficient implementations without complicating reasoning. This is because the efficiency in implementing linked data structures depends on a programmer's ability to work with aliases. The good news is that the reference behavior needed to implement linked data structures can be encapsulated in a single component. The fact that aliasing may be desirable in programs would probably not surprise most programmers, but the fact that aliasing is desirable so rarely might surprise them. The big picture looks hopeful: The vast majority of software components can be implemented efficiently without introducing aliases, and only a small minority of components need programmers to use aliases for efficient implementations. If one further consider that this small minority of components will be comprised of such basic and well understood data structures as lists, trees, graphs and graph algorithms, the outlook for programs with a diminished alias presence is even more hopeful, for these components need only be implemented once and distributed as part of a language's component library. As an example, the Java component library already includes a list component that programmers can use instead of implementing their own. It is not unreasonable to assume that complex and efficient software systems can be built using a sufficiently rich component library without any of the programmers ever having to worry about effects of aliasing.

The remainder of this paper describes the Location_Linking_Template, the component that captures the reference behavior of linked data structures.

Informal Introduction

The concept Location_Linking_Template provides all the functionality necessary for implementing linked data structures such as lists, trees, graphs, and networks. As a prelude to introducing its specification, we will informally describe the organization, the properties, and the actions permitted in an arbitrary system of linked locations. The informal description given here differs from traditional explanations of pointers, and may be easier to grasp for students new to the notion of references, but it is also straightforward to programmers already comfortable with pointers and linked data structures.

Organization

A system of linked locations is made up of a finite number of locations and one-way connections between them called links. Locations contain information and a fixed number of links. The kind of information and the precise number of links the locations contain is determined by the client when she creates the system. Each location is either *free* or *taken*. Locations start off as free and are taken by the client one at a time as she needs them. The client can only manage locations that she has taken; free locations are only of use in that they are available to be taken. When a client no longer needs a (taken) location she can abandon it, at which time it is returned to the pool of free locations. Since there are only a finite number of locations available, a smart client will be careful to abandon the locations that she no longer intends to use.

The client manipulates locations through her workers. All workers reside at some location in the system and serve as representatives of the location they occupy. If the client wishes to alter some aspect of a location, she must do so through a worker. For example, the client may wish to redirect the third link at Mark's location toward Gary's location. Rather than change the link herself, she must direct Mark to change it.

> *Mark, redirect the third link at your location toward Gary's location.*

The practical result of this command structure is that even if a location has been taken, the client cannot modify its information or links unless the location is occupied by a worker. Therefore, the smart client is careful to ensure that she can get a worker to any location she wants to update. A location that can be reached by a worker is said to be *accessible*.

Every system has a special location named *Void* that serves as a default location. The default location is perpetually free--it cannot be taken by the client. Unlike other free locations, however, the Void location is a useful part of the system. When a client recruits a new worker, he is automatically located at the Void location. Furthermore, the links of every location always point to Void until the client modifies them.

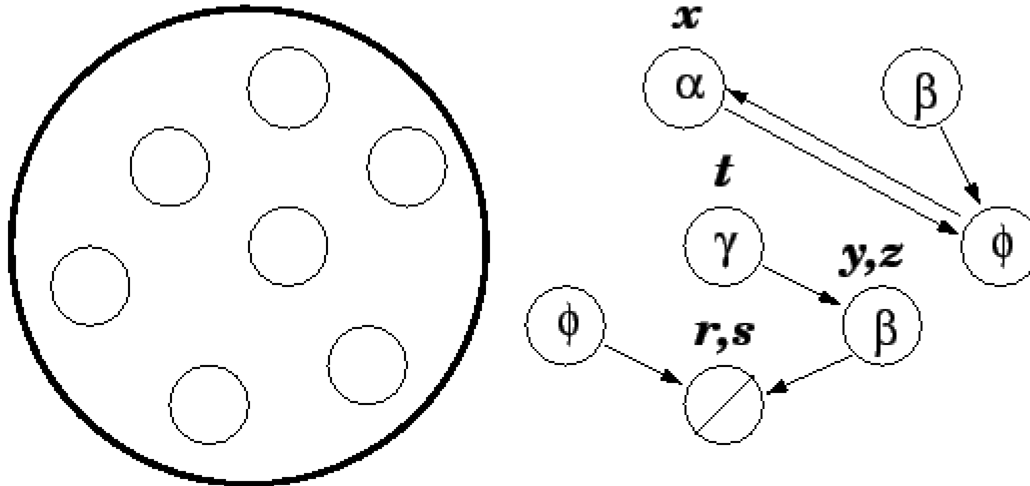


Figure 1: A system of linked locations. Small circles are locations, arrows are links, Greek letters are information, and roman letters are workers. The circle with a slash through it is the Void location, and the large circle is the pool of free locations.

Figure 1 shows a diagram of an example system where Greek letters are the information and where each location has exactly one link. The seven locations in the circle on the left are free, the six locations on the right with Greek letters are taken, and the location with the slash through it is the Void location. This figure is slightly inaccurate because all locations--even free ones--have information and links. The information in free locations is the default information (represented here by the Greek letter ϕ), and the links of free locations always point to Void. We omit these details in the inerest of making the diagram less cluttered.

Actions

The actions presented here are those that a client may perform on a system. They include administrative actions, such as establishing the system and recruiting workers, and management actions, such as changing information and redirecting links. Examples of each action are presented followed by a short description and comments.

Establish a system of linked locations that hold account information and have 1 link. Before the client can manage her system she must create it, and this action establishes the specified system. As previously stated, the information and links in free locations are not observable by the client. There are a finite number of locations but the client has no control over the number.

Recruit Mark and Gary. The client manages locations through workers, so she needs a means to recruit workers for the system. Here she is recruiting two workers who--like all newly recruited workers--begin their careers at the Void location.

The following actions enable the client to manage the system. Since locations are managed through the worker, the actions will typically take the form of commands to a worker or workers.

Mark, take a new location. A client must take a location before she can manipulate it, but the taking can only be performed under certain conditions. First, Mark must not occupy a taken location. If Mark is a new recruit, this is not a problem because he resides at the perpetually free location Void. Second, there must be at least one free location to take. Once Mark takes the location, its status changes to *taken*. A taken location remains taken until the client abandons it.

Mark, abandon your location. For this action to be performed Mark must occupy a taken location. After Mark abandons the location, he relocates to Void and the location is added to the pool of free locations. Recall that all free locations have default information and default lins, so a side effect of this action is that the information in the location is changed back to the default and all links in the location are redirected to Void. A problem may occur if two workers reside at the same location when the client abandons it. For example, suppose Mark and Gary reside at the same location. Both workers are representatives of the location, so the client can direct either of them to abandon it. If she tells Mark to abandon the location, Mark will relocate to Void, but Gary will find himself occupying a free location. If the client does sloppy bookkeeping she may try to manipulate Gary's location, causing unpredictable results.

Mark, relocate to Gary's location. This is one way a client can move Mark from one location to another. In this case the new location must already have a worker at it, but the locations may be free or taken. Thus, this action may be used to move workers to or from Void. A companion action to this enables the client to ask Mark if he and Gary are colocated.

Mark, follow your location's third link. If Mark is at a taken location, the client can move him to a new location by directing him to follow any of the links there. A location is accessible if it is occupied by a worker or if it can be reached by a worker who follows a series of links. For example, if Mark occupies a location that contains a link to a second location, both locations are accessible regardless of whether the second location is occupied by a worker, because Mark can reach the second location by following the link from his original location. When a location is first taken it is accessible because a worker resides there, but actions that relocate a worker can potentially change the accessibility of the system. In an ideal setting a client would like to ensure that all taken locations remain accessible. A taken but inaccessible location presents a problem: The location is not free so the client cannot take it, and the location is not accessible so the client cannot use it. The total number of locations is finite so if enough locations fall into this state the client may find herself trying to take a new location from an empty pool. Note that it is technically possible for the Void location to be inaccessible, but practically the client need only recruit a new worker and Void becomes accessible.

Mark, exchange the information at your location with messenger Bob's information. The client needs a way to modify the information contained in locations and this action provides it. Implicit in this action is an assumption that the client has a way to manage information, including the ability to recruit messengers who carry that information. This action can only be performed on taken locations.

Mark, redirect the third link at your location toward Gary's location. If Mark occupies a taken location, any of its links can be redirected toward Gary's location. Gary's location need not be taken, so a link at Mark's location can be directed toward Void. This action can potentially change the accessibility of the system.

Formal Specification

This section describes the formal specification of the Location_Linking_Template which appears in Appendix A. Most of the relationships between the mathematical objects in the concept and the notions introduced in the system above are straight forward.

Locations

The **Defines** clauses at the beginning of Location_Linking_Template indicate that any implementation of this concept must provide definitions for the mathematical type *Location*, the mathematical object *Void*, and the mathematical object *Taken_Location_Displacement*. Though we expect that objects of type *Location* will somehow be tied to a machine's memory addresses, we don't want to presume how the implementer will model them. In particular, the specification is flexible enough so that Locations that are free or inaccessible (or both) need not correspond to real memory locations.

Objects of type *Location* in the concept correspond to the notion of locations in the system of linked locations described above. In the concept, the type parameter, *Info*, indicates the type of information that a location contains, while the second parameter, *k*, indicates the number of links that a location contains. The three conceptual variables near the beginning of the concept are functions that take locations as arguments: *Contents*(*q*) returns the information at a given location *q*, *Target*(*q*,on i) returns the location targeted by the *i*-th link of *q*, and *Is_Taken*(*q*) returns true if *q* is taken and false if *q* is free.

The **facility constraints** ensure that the distinguished location *Void* is always free, and that all free locations have default information and default links. The **facility initialization** clause ensures that immediately after this concept is instantiated all locations are free.

Displacement

The value of *Taken_Location_Displacement* is the amount of space that a newly taken location occupies in memory. This value depends on the type *Info*, *k*, and the implementation.

The last part of the facility constraints ensures that the total number of locations is at least as large as the total amount of memory in the system divided by the amount of memory that a newly taken location occupies. In other words, the number of mathematical locations is always greater than the number of locations that can be stored in real memory. The operation Take_New_Location has as part of its requirement that the amount of real memory available (**Remaining_Memory_Capacity**) is greater than the memory that must be allocated to a newly taken location. The client has the ability to check this condition through the operation *Location_Size* supplied by this module and a global operation that returns the amount of available memory.

A discussion of comprehensive performance specifications can be found in [10]. For complete performance

specifications--those encompassing both duration and displacement--we envision a separate performance profile for each implementation. The displacement information described here merits special treatment because it represents displacement in the system that is permanent until it is explicitly deallocated.

Positions

The concept exports the programming type *Position*. Position variables are the workers described in the system above. The declaration

```
Var Mark, Gary: Position;
```

corresponds to the action *Recruit Mark and Gary* described above. Note that although position *variables* correspond to workers, their mathematical *values* correspond to locations. For example, the **initialization ensures** clause asserts that $p = Void$. Since the symbol *p* that occurs here is the mathematical value of 'the programming variable p' rather than 'the programming variable p' itself, the assertion is interpreted as *The location of the worker named p is Void*, or simply *Worker p resides at Void*.

The predicate *Is_Accessible* is a variable rather than a constant (like most defined objects) because its value depends not only on the value of its parameter, *q*, but on the state of the program--the same location may be accessible in one program state and inaccessible in the next. The definition states that a location is accessible if there is a series of links to that location starting from a location represented by an in-scope Position variable. The predicate Position.**Has_Active_Variable**(*q*) indicates whether the location *q* is represented by a Position variable that is currently in scope.

Operations

The management actions described above correspond directly to the operations given in the concept and its enhancements. One important primary operation that was not described above it *Swap_Locations*. The operation is similar to *Redirect_Link* except it also relocates the worker from the new target location to the old target location (see Figure 2). In effect, the specified link and the specified worker are swapping locations. This operation has the desirable property that it does not effect the accessibility of the system. Furthermore, using *Swap_Locations* and *Relocate* one can implement both *Redirect_Link* and *Follow_Link*, effectively making them secondary operations.

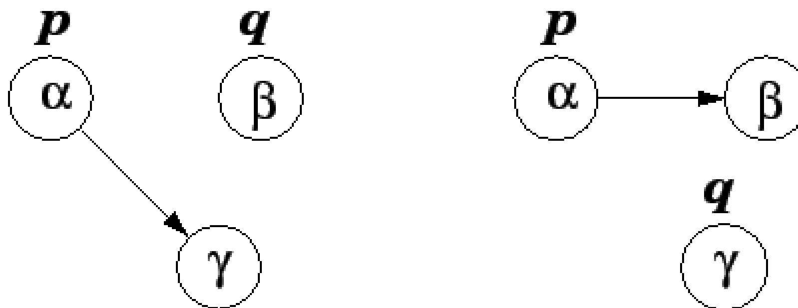


Figure 2: The procedure call Swap_Locations(p, 1, q) redirects the first link of p's location toward q, and relocates q to the link's original target.

Through its operations, the Location_Linking_Template provides the reference behavior needed to efficiently implement linked data structures. In particular, the client can reap the benefits of aliasing by positioning two or more workers at the same location. But the concept also allows the client to fall into the traditional traps involving references: dangling references and memory leaks. A dangling reference occurs when a location is free but remains accessible, and a memory leak occurs when a location is taken but *not* accessible.

Dangling References

The following code segment creates a dangling reference.

```
Var x, y: Position;
Take_New_Location(x);
Relocate(y, x);
```

```
Abandon_Location(x);
```

When *x* abandoned his location, the location's status changed from taken to free. Though *x* was relocated to Void, *y* remained at the location, so the location continues to be accessible. Position variables are effectively bound to the type of *Info* during instantiation, so there is no danger of inadvertently modifying (through the dangling reference) the contents of a memory location that is being used by another variable somewhere else in the program. Real memory locations on a machine are limited, so the specification permits implementations that can reclaim memory even if a dangling reference existed for them.

The Is_Usable operation (provided as an extension to the concept) effectively tells the client whether a worker is a dangling reference. Since a worker resides at the location in question, the location is accessible. If the location is taken, it is usable by the client. But if the location is free, the client cannot affect it.

Memory Leaks

The following code segment creates a memory leak.

```
Var x, y: Position;
Take_New_Location(x);
Relocate(x, y);
```

The location that was taken by *x* continues to have a taken status but has become inaccessible. Real memory locations are limited, so a proliferation of memory leaks is a serious problem. There are traditionally two ways to deal with memory leaks. The first is to avoid them by keeping a careful accounting of the locations in the system and explicitly abandoning a location before it becomes a leak. The second is to do periodic garbage collection. The operation that performs garbage collection, Abandon_Inaccessible, is provided in an extension to the concept.

Extensions

An **extension** differs from an **enhancement** in that its operations cannot *in principle* be implemented as a secondary operation. Extensions are separate from the main concept because they cannot be implemented without costly performance penalties. Default realizations do not implement extensions, but custom realizations may. Both Abandon_Inaccessible and Is_Usable are extensions. A garbage collection implementation of Location_Linking_Template would additionally provide a procedure for the Abandon_Inaccessible operation. A client may then choose to ignore the Abandon_Location operation and periodically invoke the Abandon_Inaccessible operation instead.

Application

Using Location_Linking_Template to implement linked data structures will be familiar to anyone who has implemented a linked list in a language with pointers such as C or Pascal. This section presents a *List* concept and its implementation using Location_Linking_Template. The specification for the list--*List_Template*--is given in the appendix. The mathematical model for the type *List* consists of two strings--a left string and a right string. When an item is inserted into the list it is inserted at the beginning of the right string; when an item is removed from the list it is removed from the beginning of the right string.

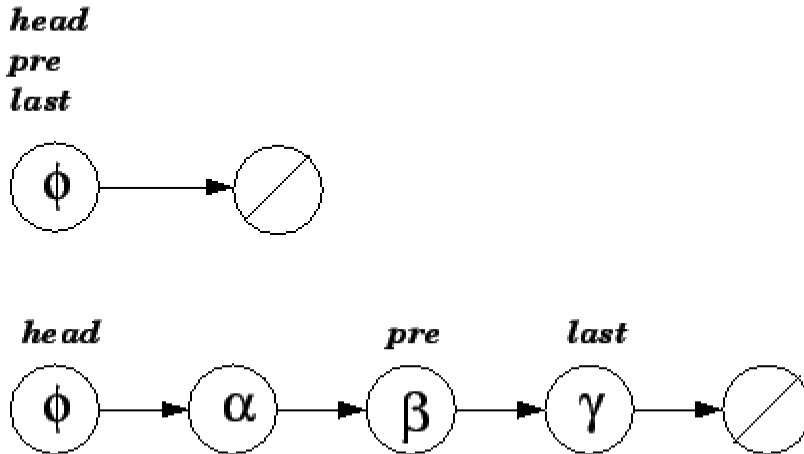


Figure 3: The first system depicts an empty list and the second system depicts a list with three items α, β, and γ where the insertion point is between β and γ.

The implementation of List_Template (*Location_Linking_Based*) appears after List_Template in Appendix B. The list is implemented as a record with three Position fields and two Integer fields. The field *head* perpetually resides at a dummy location at the beginning of the list. This location serves as a sentinel--it will not be used to hold information, but it allows the implementer to ignore certain boundary conditions. The field *pre* is always located at the end of the left string; if the left string is empty, *pre* is colocated with *head*. The field *last* is always located at the end of the right string; if the right string is empty, *last* is colocated with *pre*.

The conventions in the type declaration assert invariants that must hold before and after each procedure. The first three conjunctions in this convention describe how the fields of the record (head, pre, last, left_length, and right_length) are related. From these conjunctions (along with the correspondence) we can show that Void is accessible from all locations that hold items in the list. The last conjunction effectively prohibits dangling references and memory leaks.

Figure illustrates how a system is updated during the *Insert* operation.

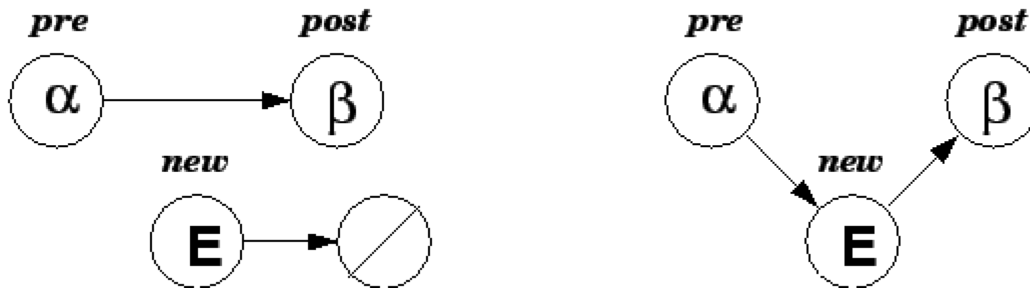


Figure 4: The Insert operation can be thought of as consisting of four main sections. Recruiting a worker named *post* to reside at the location targeted by *pre*, creating a new location with the desired information (the left diagram), redirecting links so that the new location is positioned correctly in the list (the right diagram), and ensuring that the worker *last* resides at his correct location in the updated system.

Related Work and Conclusion

The problems involving references in high-level languages has been around for a long time and therefore a good deal has been written on the subject. Hoare critiqued them early on [4,5] and others followed suit [8,2]. Specifiers were quick to single out aliasing as the main problem caused by references that breaks modular reasoning [2,6]. Modern practical programmers

like Koenig have described the difficulty that students have understanding pointers [9], and Stepanov cites value semantics (along with efficiency) as one of the key principles behind the development of the C++ Standard Template Library [15]. An overview of the problem with reference behavior as it relates to reusable components is given in [11].

The desire for both efficiency and value semantics is what led Harms to consider swapping as an alternative form of data movement in [3], and the same mechanism is applied to parameter passing in [17]. Ownership types are introduced in [1] as a way to control aliasing, and dynamic dispatch is offered as way to eliminate aliasing altogether during parameter passing in [12].

Components that capture the functionality and performance of references were suggested by Hollingsworth in [7]. The Location_Linking_Template described in this paper has gone through many iterations, some of which are presented in [16].

Pointers and reference variables present a problem in high-level languages because of their tendency to introduce aliasing, which makes reasoning more complex and often breaks modularity. Though aliasing can often be avoiding by the use of efficient data movement mechanisms such as swapping, high-level languages also need mechanisms that allow programmers to use aliasing for those rare occasions when it cannot be avoided. Such mechanisms should not break modular reasoning. The Location_Linking_Template introduced in this paper satisfies these conditions.

Along with this mechanism, a method of software development should be employed that ensures the need for aliasing will be rare. Language designers must recognize that many linked data structures and their algorithms are understood well enough that they can and should be provided to programmers as integral parts of a rich and reusable component library.

Appendix A

Concept Location_Linking_Template(**type** Info; **evaluates** k: Integer);
uses String_Theory, Std_Boolean_Facility, Std_Integer_Fac;

Defines Location: $\mathbf{S}$et;
Defines Void: Location;
Defines Taken_Location_Displacement: $\mathbf{N}^+$;

Var Target: $\text{Location} \times [1..k] \rightarrow \text{Location}$;

Var Contents: $\text{Location} \rightarrow \text{Info}$;
Var Is_Taken: $\text{Location} \rightarrow \mathbf{B}$;

Facility Constraints $\neg\text{Is_Taken}(\text{Void})$ **and**

$\forall q : \text{Location}$, $\text{Is_Taken}(q)$ **and** $\text{Info.Is_Initial}(\text{Contents}(q))$ **and**

$||\text{Location}|| \geq \textbf{Total_Memory_Capacity}/\text{Taken_Location_Displacement}$;

Facility Initialization ensures $\forall q : \text{Location}, \ \neg\text{Is_Taken}(q)$;

Type Family Position **is modeled by** Location;
exemplar p;
initialization ensures p = Void;
Definition Variable Is_Accessible(q: Location): $\mathbf{B}$ =

$\exists p : \text{Position}, \ \exists h : [1..k], \ \exists \rho : \text{Str}(\text{Location} \times [1..k]) \ni$

$\text{Position.Has_Active_Variable}(p)$ **and** $\langle(p, h)\rangle$ Is_Prefix ρ **and**

$\forall u, v : \text{Location}, \ \forall i, j : [1..k]$, **if** $\langle(u, i)\rangle \circ \langle(v, j)\rangle$ Is_Substring ρ **then**

$\text{Target}(u, i) = v$ **and** $\langle(q, 1)\rangle$ Is_Suffix ρ;

finalization updates Is_Accessible;

Operation Take_New_Location(**updates** p: Position);
updates Is_Taken, Is_Accessible;
requires $\neg \text{Is_Taken}(p)$ **and**

$\text{Taken_Location_Displacement} \leq \textbf{Remaining_Memory_Capacity}$;

ensures $\neg \#\text{Is_Taken}(p)$ **and** $\neg \#\text{Is_Accessible}(p)$ **and**

$\forall j : [1..k],\ \text{Target}(p, j) = \text{Void}$ **and**

$\forall q : \text{Location},$ **if** $q \neq p$ **then** $\text{Is_Taken}(q) = \#\text{Is_Taken}(q)$;

Operation Abandon_Location(**clear** p: Position);
updates Target, Contents, Is_Taken, Is_Accessible;
requires Is_Taken(p);
ensures $\forall q : \text{Location},$

$$\left(\text{Is_Taken}(q) = \begin{cases} \textbf{false} & \textbf{if } q = p \\ \#\text{Is_Taken}(q) & \textbf{otherwise} \end{cases}\right) \textbf{ and}$$

if $\text{Is_Taken}(q)$ **then** $\text{Contents}(q) = \#\text{Contents}(q)$ **and**

$\forall n : [1..k],\ \text{Target}(q, n) = \#\text{Target}(q, n)$;

Operation Relocate(**updates** p: Position; **restores** new_location: Position);
ensures p = new_location;

Operation Check_Colocation(**preserves** p, q: Position;
replaces are_colocated: Boolean);
ensures if ((Is_Taken(p) **or** p = Void) **and** (Is_Taken(q) **or** q = Void))
then are_colocated = (p = q);

Operation Swap_Locations(**preserves** p: Position; **evaluates** i: Integer;
updates new_target: Position);
updates Target;
requires $1 \leq i \leq k$ **and** $\text{Is_Taken}(p)$;

ensures $\forall q : \text{Location},\ \forall j : [1..k],$

$$\left(\text{Target}(q, j) = \begin{cases} \#\text{new_target} & \textbf{if } r = p \textbf{ and } j = i \\ \#\text{Target}(q, j) & \textbf{otherwise} \end{cases}\right) \textbf{ and}$$

$\text{new_target} = \#\text{Target}(p, i)$;

Operation Swap_Contents(**preserves** p: Position; **updates** I: Info);
updates Contents;
requires Is_Taken(p);
ensures $I = \#\text{Contents}(p)$ **and** $\forall q : \text{Location}$,

$$\text{Contents}(q) = \begin{cases} \#I & \textbf{if } q = p \\ \#\text{Contents}(q) & \textbf{otherwise} \end{cases};$$

Operation Is_At_Void(**preserves** p: Position): Boolean;

ensures Is_At_Void = (p = Void);

Operation Location_Size(): Integer;
ensures Location_Size = (Taken_Location_Displacement);

end Location_Linking_Template;

Enhancement Other_Operations **for** Location_Linking_Template;

Operation Redirect_Link(**preserves** p: Position; **evaluates** i: Integer;
preserves new_target: Position);
updates Target, Is_Accessible;
requires $1 \leq \mathrm{i} \leq \mathrm{k}$ **and** $\mathrm{Is_Taken(p)}$;
ensures $\forall \mathrm{q} : \mathrm{Location},\ \forall \mathrm{j} : [1..\mathrm{k}],$

$$\mathrm{Target(q,j)} = \begin{cases} \#\mathrm{new_target} & \textbf{if } \mathrm{q} = \mathrm{p} \textbf{ and } \mathrm{j} = \mathrm{i} \\ \#\mathrm{Target(q,j)} & \textbf{otherwise} \end{cases};$$

Operation Follow_Link(**update** p: Position; **preserves** i: Integer);
updates Is_Accessible;
requires \(\mathrm{Is} **and** \(1\leq \textrm{ ;
ensures $\mathrm{p} = \mathrm{Target(\#p,i)}$;

end Other_Operations;

Extension Usability_Checking_Capability **for** Location_Linking_Template;

Operation Is_Usable(**preserves** p: Position): Boolean;
ensures Is_Usable = (Is_Taken(p));

end Usability_Checking_Capability;

Extension Cleanup_Capability **for** Location_Linking_Template;

Operation Abandon_Inaccessible();
updates Is_Taken, Contents, Target;
ensures $\forall \mathrm{q} : \mathrm{Location},$

$\mathrm{Is_Taken(q)} = (\#\mathrm{Is_Accessible(q)}$ **and** $\#\mathrm{Is_Taken(q)})$ **and**

if $\mathrm{Is_Taken(q)}$ **then** $\mathrm{Contents(q)} = \#\mathrm{Contents(q)}$ **and**

$\forall \mathrm{i} : [1..\mathrm{k}],\ \mathrm{Target(q,j)} = \#\mathrm{Target(q,j)}$;

end Cleanup_Capability;

Appendix B

Concept List_Template(**type** Entry);

Defines List_Unit_Displacement: $\mathbf{N}^{+}$;

Type Family List **is modeled by Cart_Prod**
Left: **Str**(Entry);
Right: **Str**(Entry);

end;
exemplar S;
initialization ensures $|\mathrm{S.Left}| = 0$ **and** $|\mathrm{S.Right}| = 0$;

Operation Insert(**alters** E: Entry; **updates** S: List);
requires List_Unit_Displacement $\leq$ **Remaining_Memory_Capacity**;
ensures $\mathrm{S.Left} = \#\mathrm{S.Left}$ **and** $\mathrm{S.Right} = \langle \#\mathrm{E} \rangle \circ \#\mathrm{S.Right}$;

Operation Remove(**replace** R: Entry; **updates** S: List);
requires $|\mathrm{S.Right}| > 0$;
ensures $\mathrm{S.Left} = \#\mathrm{S.Left}$ **and** $\#\mathrm{S.Right} = \langle \mathrm{R} \rangle \circ \mathrm{S.Right}$;

Operation Advance(**updates** S: List);
requires $|\mathrm{S.Right}| > 0$;
ensures $\mathrm{S.Left} \circ \mathrm{S.Right} = \#\mathrm{S.Left} \circ \#\mathrm{S.Right}$ **and**
$|\mathrm{S.Left}| = |\#\mathrm{S.Left}| + 1$;

Operation Advance_To_End(**updates** S: List);
ensures \(\vert\mathrm{ **and** $\mathrm{S.Left} = \#\mathrm{S.Left} \circ \#\mathrm{S.Right}$;

Operation Reset(**updates** S: List);
ensures $|\mathrm{S.Left}| = 0$ **and** $\mathrm{S.Right} = \#\mathrm{S.Left} \circ \#\mathrm{S.Right}$;

Operation Swap_Rights(**updates** S1, S2: List);
ensures $\mathrm{S1.Left} = \#\mathrm{S1.Left}$ and $\mathrm{S2.Left} = \#\mathrm{S2.Left}$ and
$\mathrm{S1.Right} = \#\mathrm{S2.Right}$ and $\mathrm{S2.Right} = \#\mathrm{S1.Right}$;

Operation Left_Length(**restores** S: List): Integer;
ensures Left_Length = $(|\mathrm{S.Left}|)$;

Operation Right_Length(**restores** S: List): Integer;
ensures Right_Length = $(|\mathrm{S.Right}|)$;

Operation Unit_Size(): Integer;
ensures Unit_Size = (List_Unit_Displacement);

end List_Template;

Realization Location_Linking_Realiz **for** List_Template;
uses Location_Linking_Template;

Facility Std_Pointer_Fac **is** Location_Linking_Template(Entry, 1)
realized by Std_Realiz;

Definition List_Unit_Displacement = Taken_Location_Displacement;

Definition Variable Next(p: Location): Location = Target(p, 1)

;

```
Type List = Record
        head, pre, last: Position;
        left_length, right_length: Integer;
    end;
```

conventions $S.left_length \geq 0$ **and** $S.right_length \geq 0$ **and**

$$S.pre = \begin{cases} S.head & \textbf{if } S.left_length = 0 \\ Next^{S.left_length}(S.head) & \textbf{otherwise} \end{cases}$$ **and**

$$S.last = \begin{cases} S.pre & \textbf{if } S.right_length = 0 \\ Next^{S.right_length}(S.pre) & \textbf{otherwise} \end{cases}$$ **and**

$Next(S.last) = Void$ **and**

$\forall q : Location, (\ Is_Accessible(q) \textbf{ iff } Is_Taken(q)\ )$;

correspondence

Conc.S.Left = $\prod_{k=1}^{S.left_length} \langle Contents(Next^{k}(S.head)) \rangle$ **and**

Conc.S.Right = $\prod_{k=1}^{S.right_length} \langle Contents(Next^{k}(S.pre)) \rangle$;

```
    initialization
        Take_New_Location(S.head);
        Relocate(S.pre, S.head);
        Relocate(S.last, S.head);
    end;

Procedure Insert(alters E: Entry; updates S: List);
    Var post, new: Position;
    Relocate(post, pre);
    Follow_Link(post, 1);
    Take_New_Location(new);
    Swap_Contents(new, E);
    Redirect_Link(S.pre, 1, new);
    Redirect_Link(new, 1, post);
    If S.right_length = 0 then
        Follow_Link(S.last, 1);
    end;
    S.right_length := S.right_length + 1;
end Insert;

Procedure Remove(replace R: Entry; updates S: List);
    Var p: Position;
    Relocate(p, pre);
    Follow_Link(p, 1);
    Follow_Link(p, 1);
    Swap_Locations(S.pre, 1, p);
    Swap_Contents(p, R);
    Abandon_Location(p);
    If S.right_length = 1 then
        Relocate(S.last, S.pre);
```

```
        end;
        S.right_length := S.right_length - 1;
    end Remove;

    Procedure Advance(updates S: List);
        Follow_Link(S.pre, 1);
        S.left_length := S.left_length + 1;
        S.right_length := S.right_length - 1;
    end Advance;

    Procedure Advance_To_End(updates S: List);
        Relocate(S.pre, S.last);
        S.left_length := S.left_length + S.right_length;
        S.right_length := 0;
    end Advance_To_End;

    Procedure Reset(updates S: List);
        Relocate(S.pre, S.head);
        S.right_length := S.right_length + S.left_length;
        S.left_length := 0;
    end Reset;

    Procedure Swap_Rights(updates S1, S2: List);
        Var post: Position;
        If S.right_length ≠ 0 then

            Relocate(post, S1.pre);
            Follow_Link(post, 1);
            Swap_Locations(S1.pre, 1, post);
            Swap_Locations(S2.pre, 1, post);
            S1.last :=: S2.last;
        end;
    end Swap_Rights;

    Procedure Left_Length(restores S: List): Integer;
        Left_Length = S.left_length;
    end;

    Procedure Right_Length(restores S: List): Integer;
        Right_Length = S.right_length;
    end;

    Procedure Unit_Size(): Integer;
        Unit_Size = Location_Size();
    end;

end Location_Linking_Realiz;
```

Appendix C

The realization below makes use of special syntax provided for the Location_Linking_Template. Workers in the system will be of type *Node*. A Node is a Position for a system in which locations contain information of type *Entry* and a single link, labeled *next*. An up-arrow (↑) before a worker indicates that he is taking a new location, and a down-arrow (↓) in front of a worker indicates that he is abandoning his location. An asterisk (*) in front of a worker denotes the information at that worker's location. A hat (^) between a location and a link name denotes the link at that location, and an arrow (→) between locations denotes movement of a worker or link (at the foot of the arrow) to a new location (at the head of the arrow). So x → y is syntactic sugar for Relocate(x, y), x → x^next is syntactic sugar for Follow_Link(x, 1), and x^next → y^next is syntactic sugar for a secondary operation that redirects the first link at x's location to the location pointed to by the first link of y's location. The double arrow (↔) indicates simultaneous movement in both directions.

```
Realization Location_Linking_Based for List_Template;
```

```
uses Std_Pointer_Fac;

(* Facility already imported through the uses clause. *)
(* Definitions are the same as above. *)

Type Node = ^Entry(next);

Type List = Record
        head, pre, last: Node;
        left_length, right_length: Integer;
    end;
    (* Conventions and correspondence are the same as above. *)
    initialization
        ↑S.head;

        S.pre → S.head;
        S.last → S.head;
    end;

Procedure Insert(alters E: Entry; updates S: List);
    Var new: Node;
    ↑new;

    *new :=: E;
    new^next → S.pre^next;
    S.pre^next → new;
    If S.right_length = 0 then
        S.last → S.last^next;
    end;
    S.right_length := S.right_length + 1;
end Insert;

Procedure Remove(replace R: Entry; updates S: List);
    Var old: Node;
    old → S.pre^next;
    S.pre^next → S.pre^next^next;
    *old :=: R;
    ↓old;

    If S.right_length = 1 then
        S.last → S.pre;
    end;
    S.right_length := S.right_length - 1;
end Remove;

Procedure Advance(updates S: List);
    S.pre → S.pre^next;
    S.left_length := S.left_length + 1;
    S.right_length := S.right_length - 1;
end Advance;

Procedure Advance_To_End(updates S: List);
    S.pre → S.last;
    S.left_length := S.left_length + S.right_length;
    S.right_length := 0;
end Advance_To_End;

Procedure Reset(updates S: List);
    S.pre → S.head;
    S.right_length := S.right_length + S.left_length;
    S.left_length := 0;
```

```
    end Reset;

    Procedure Swap_Rights(updates S1, S2: List);
        If S.right_length ≠ 0 then

            S1.pre^next ↔ S2.pre^next;
            S1.last ↔ S2.last;
        end;
    end Swap_Rights;

    (* Remaining procedures are the same as above. *)

end Location_Linking_Based;
```

Bibliography

1
D. G. Clarke, J. M. Potter, and J. Noble.
Ownership types for flexible alias protection.
In *OOPSLA '98 Conference Proceedings*, pages 48-64. ACM Press, 1998.

2
S. A. Cook.
Soundness and completeness of an axiom system for program verification.
SIAM Journal of Computing, 7(1):70-90, 1978.

3
D. E. Harms and B. W. Weide.
Copying and swapping: Influences on the design of reusable software components.
IEEE Transactions on Software Engineering, 17(5):424-435, May 1991.

4
C. A. R. Hoare.
Hints on programming language design.
In C. A. R. Hoare and C. B. Jones, editors, *Essays in Computing Science*. Prentice Hall, New York, 1989.

5
C. A. R. Hoare.
Recursive data structures.
In C. A. R. Hoare and C. B. Jones, editors, *Essays in Computing Science*. Prentice Hall, New York, 1989.

6
J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt.
The Geneva Convention on the treatment of object aliasing.
OOPS Messenger, 3(2):11-16, 1992.

7
J. E. Hollingsworth and B. W. Weide.
Engineering "unbounded" reusable Ada generics.
In *Proceedings of the 10th Annual National Conference on Ada Technology*, pages 82-97. ANCOST, 1992.

8
R. B. Kieburtz.
Programming without pointer variables.
In *Proceedings of the SIGPLAN '76 Conference on Data: Abstraction, Definition, and Structure*. ACM Press, 1976.

9
A. Koenig and B. Moo.
Teaching standard C++.
JOOP, 11(7):11-17, 1998.

10
J. Krone, W. F. Ogden, and M. Sitaraman.

Modular verification of performance correctness.
In *OOPSLA 2001 SAVCBS Workshop Proceedings*, 2002.
`http://www.cs.iastate.edu/~leavens/SAVCBS/papers-2001/index.html`.

11
G. Kulczycki.
Efficient reusable components with value semantics.
In *ICSR 2002 Young Researchers Workshop*, 2002.
`http://www.info.uni-karlsruhe.de/~heuzer/ICSR-YRW2002/program.html`.

12
G. T. Leavens and O. Antropova.
ACL--eliminating parameter aliasing with dynamic dispatch.
Technical Report 98-08a, Department of Computer Science, Iowa State University, 1998.

13
K. R. M. Leino and G. Nelson.
Data abstraction and information hiding.
Technical Report 160, Compaq SRC, 2000.

14
P. Müller and A. Poetzsch-Heffter.
Modular specification and verification techniques for object-oriented software components.
In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*. Cambridge University Press, Cambridge, United Kingdom, 2000.

15
D. R. Musser, G. J. Derge, and A. Saini.
STL Tutorial and Reference Guide.
Addison-Wesley, Boston, 2nd edition, 2001.

16
W. F. Ogden.
The Proper Conceptualization of Data Structures.
The Ohio State University, Columbus, OH, 2000.

17
M. Sitaraman, G. W. Kulczycki, W. F. Ogden, B. W. Weide, and G. T. Leavens.
Understanding and minimizing the impact of reference-value distinction on software engineering.
Technical report, Department of Computer Science, Clemson University, 2002.

About this document ...

**Capturing the Reference Behavior
of Linked Data Structures**

This document was generated using the **LaTeX**2`HTML` translator Version 99.2beta8 (1.42)

Copyright © 1993, 1994, 1995, 1996, Nikos Drakos, Computer Based Learning Unit, University of Leeds.
Copyright © 1997, 1998, 1999, Ross Moore, Mathematics Department, Macquarie University, Sydney.

The command line arguments were:
latex2html `-split 0 -white main_paper`

The translation was initiated by Gregory W. Kulczycki on 2002-05-10

Footnotes

... reserved.[1]
This research funded in part by NSF grant CCR-0113181.
... indirection."[2]

This saying has been attributed to Butler Lampson.

... structures.[3]

References are also used for memory management, but a discussion of this topic is beyond the scope of this paper.

Gregory W. Kulczycki 2002-05-10

Displacement Violation Checking and Ghost Facilities

Murali Sitaraman and Greg Kulczycki
Dept. of Computer Science
Clemson University
451 Edwards Hall
Clemson, SC 29634-0974 USA

{murali, gregwk}@cs.clemson.edu
Phone: +1 864 656 3444
URL: **http://www.cs.clemson.edu/~resolve**

Abstract

RESOLVE concepts are designed so that a client can write defensive programs if he chooses to do so. This ability to defend should also be available against situations where a client wants to be sure that there is enough memory to declare variables, facilities, or call certain procedures. This is a proposal for accomplishing this objective.

Keywords

Specification of displacement requirements, checking

Paper Category: technical paper
Emphasis: research and education

1. Introduction

The objective of this position paper is to illustrate how to add checks for displacements to simply bounded, communal, and globally-bounded modules.The rest of this document discusses how to achieve the goal of providing defensiveness against displacement violations (including language support), and introduces the need for ghost facilities in the process.

2. Examples to Illustrate Technical Details

Concept Stack_Template(**type** Entry; **evaluates** Max_Depth: Integer);

    uses Std_Integer_Fac, String_Theory;

        requires Max_Depth > 0;

  Type Family StackÍ **Str**(Entry);

    exemplar S;

    constraints ½S½ £ Max_Depth;

    initialization

        requires *Stack.****Disp****(L) £* ***Rem_Mem_Cap****;*

        ensures S = L;

  Operation Push(**alters** E: Entry; **updates** S: Stack);

    requires ½S½ < Max_Depth;

ensures S = <#E> ∘ #S;

Operation Pop(**replaces** R: Entry; **updates** S: Stack);

requires ½S½ > 0 ;

ensures #S = <R> ∘ S;

Operation Depth(**restores** S: Stack): Integer;

ensures Depth = (½S½);

Operation Rem_Capacity(**restores** S: Stack): Integer;

ensures Rem_Capacity = (Max_Depth -½S½);

Operation Clear(**clears** S: Stack);

end Stack_Template;

Discussion

We believe that there is a fundamental difference between truly temporary displacement needed to perform a procedure call versus the displacement needed to create a new bounded stack or push an additional entry on a globally-bounded stack. This is because a runtime system can set aside sufficient memory to handle a nominal number of procedure calls. "Out of memory" errors are more likely to occur on facility and variable initializations and certain operations on globally-bounded concepts (and infrequently-used recursive calls).

Displacement is an operation implicitly declared on every type and a procedure for it is provided by an implementation. Like initialization, the code for this procedure is often automatic, and therefore, can be omitted. The type initialization requires clause is automatic, but we may decide to leave it in till becomes entrenched.

Client Usage

If SF.Stack.**Displacement** ( ) <= Rem_Mem_Cap( ) **then**

Var S:SF.Stack;

Example #2

Concept Stack_Template(**type** Entry);

uses Std_Integer_Fac, String_Theory;

Type Family StackÍ **Str**(Entry);

exemplar S;

constraints ½S½ £ Max_Depth;

initialization

requires *Stack.**Disp**(L) £ **Rem_Mem_Cap**;*

ensures S = L;

Defines Push_Disp: N+;

Operation Push(**alters** E: Entry; **updates** S: Stack);

requires Push_Disp £ **Rem_Mem_Cap**;

ensures S = <#E> ∘ #S;

Operation Push_Displacement: Integer;

ensures Push_Displacement = (Push_Disp);

Operation Pop(**replaces** R: Entry; **updates** S: Stack);

requires ½S½ > 0 ;

ensures #S = <R> ∘ S;

Operation Depth(**restores** S: Stack): Integer;

ensures Depth = (½S½);

Operation Clear(**clears** S: Stack);

end Stack_Template;

Figure 1: A Concept for Locally Bounded Stack_Template

Client Usage

If SF.Push_Displacement ( ) <= **Rem_Mem_Cap**( ) **then**

Push(E, S);

Example #3

Concept Communal_Stack_Template(**type** Entry;

evaluates Total_Max_Depth: Integer);

```
        uses Std_Integer_Fac, String_Theory;
            requires Tot_Max_Depth > 0;

    Defines Disp;

    Facility initialization
            requires Disp £ Rem_Mem_Cap;

    Operation Displacement: Integer;
        ensures Displacement =( Disp );

    …
end Communal_Stack_Template;
```

Displacement is an operation implicitly declared on every facility and a procedure for it is provided by an implementation. As usual, the code for this procedure is often automatic, and therefore, can be omitted. (This operation is also defined for simply bounded and globally bounded facilities, but it is trivial in those cases.) All related assertions and terms in the concept are pre-defined and automatic, and can be omitted.

Client Usage

```
Ghost Facilty SF is Communal_Stack_Template(Integer, 1000)
    realized by Shared_Realiz;

If SF.Displacement ( ) <= Rem_Mem_Cap( ) then
    Ghost SF is permanent;
```

Facility-wide displacement checking raises a problem because the facility needs to be declared before a question about its displacement requirement can be asked.To solve this problem, we propose ghost facilities.The only operation that can be called on a ghost facility is **Displacement**. When a ghost facility is declared, all the facilities declared in the ghost facility also become ghost facility declarations automatically.Once the client has checked memory is available, then she can declare the actual facility

Acknowledgments

This research is funded in part by NSF grant CCR-0113181. This draft paper is motivated in part by what we said we would do under the NSF grant and a discussion with Bill Ogden concerning the specification of Location_Referencing_Template.