1999

Formalization of storage considerations in software design

Lakshminarasimha Reddy Ankireddipally
West Virginia University

FORMALIZATION OF STORAGE CONSIDERATIONS IN SOFTWARE DESIGN

L. R. Ankireddipally

Dissertation Submitted to
The college of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy
in
Computer Science

Dr. Murali Sitaraman, Chair
Dr. James D. Mooney
Dr. Srinivas Kankanahalli
Dr. Hani Ammar
Dr. Marcello R. Napolitano

Department of Computer Science
and Electrical Engineering

Morgantown, West Virginia
1999

Keywords: Formal methods, Storage Specification and verification, Storage Manager

A B S T R A C T

FORMALIZATION OF STORAGE
CONSIDERATIONS IN SOFTWARE DESIGN

by L. R. Ankireddipally

One of the technical impediments for the widespread adoption of the formal methods is an inability to address storage-related concerns such as "out of memory" errors. The focus of this dissertation is on formal specification and modular reasoning of storage-related aspects of practical components and systems. In particular, this thesis tries to address the following fundamental storage-related questions for practical component-based software development:

- Is it possible to reason statically and in a modular fashion that the system would not run "out of memory"?
- Is it possible for the reasoning system to be modular, yet sufficiently precise with respect to storage constraints?
- Is it possible to have a formally specified storage management mechanism that is predictable and efficient, yet allows effective storage utilization?

The main contribution of this research work is a formal and modular framework for storage specification and reasoning. A memory management mechanism that is predictable and efficient is also a part of this dissertation.

TABLE OF CONTENTS

LIST OF FIGURES

Number **Page**

ACKNOWLEDGMENTS

My sincere thanks to my advisor, Dr. Murali Sitaraman, without whose guidance, support, and patience, completing this research could not have been possible. Dr. Murali, I appreciate and am grateful for your amazing patience and willingness to spare time for me, whether it was in Morgantown, Seattle or Victoria, British Columbia. I am also grateful to my committee members, Dr. James D. Mooney, Dr. Srinivas Kanakanahalli, Dr. Hany Ammar, and Dr. Marcello R. Napolitano for their support and understanding.

Also, I am grateful to Dr. Bruce Weide and Dr. Bill Ogden for taking time to provide their insightful comments and suggestions.

Finally, I am awestruck with and indebted to the amazing political and civic institutions of this great country. Especially, I am extremely grateful to the West Virginia University System.

Narasimha (L. R. Ankireddipally)

Chapter 1

INTRODUCTION

Component-based technologies such as Java Beans and COM/ActiveX are finding increasing use in industrial practices because of their potential to improve software quality and productivity [1,2]. However, use of these technologies in life-critical or mission-critical systems is not yet considered safe as seen from the license agreement of the Java Development Kit (JDK) 1.1.x that contains following warning [3]:

> "Software is not designed or intended for use in on-line control of aircraft, air traffic, aircraft navigation or aircraft communications; or in the design, construction, operation or maintenance of any nuclear facility. Licensee warrants that it will not use or redistribute the Software for such purposes."

Not withstanding such warnings, component-based technologies may be already in use in critical applications and this trend is likely to continue because of business and productivity considerations [4,5,6,7,8]. *Reliable* and *effective* construction and use of practical component technologies in developing larger systems are among the most fundamental software engineering problems. It is widely recognized that a key to addressing these problems is abstract behavioral specification of components [9, 10,15].

Specifications make it possible to understand the behavior of the components precisely. They allow reasoning about a system composed from the components, independent of and without understanding internal details of the components [11,10]. If every module or component of a large system has behavioral specifications, then specification-based *modular reasoning* allows implementations of modules to be verified one at a time. This

modularity in the reasoning process is essential to make it scalable. Other benefits of specifications, including their role as formal contracts between implementers and users of components and their use software analysis/testing, are well documented [12,13,14,15].

In spite of their potential, formal specification and reasoning methods have at best found limited usage in the practical component-based technologies. Some of the frequentlycited and valid reasons for this gap between theory and practice are non-technical: managerial reluctance to invest in formal methods and implementer reluctance to adapt formal methods. But, more importantly, there are fundamental technical impediments, ranging from use of ad hoc implementation techniques that preclude abstract behavioral specification to limitations of the formal specification techniques. Examples of these technical problems include-storage related issues such as "out of memory" errors, handling of reference variables and reference parameter passing, and capturing of component behaviors involving inversion control. For practitioners to be able to apply formal specification techniques with confidence, these and other unresolved research issues in describing component behaviors must be addressed.

The focus of this dissertation is on formal specification and modular reasoning of storage-related aspects of practical components and systems. Storage-related errors are the most notorious and difficult to debug in practical software components. In principle, storage-related issues must take on additional importance in formal methods, because the errors cannot be detected easily using informal techniques. However, these issues that arise in the realization of every component and system have received little attention in the formal methods literature. In particular, no formal techniques exist to address the following fundamental storage-related questions for practical component-based software development:

1. Given a component-based system, constraints on input parameters, and maximum available storage capacity, is it possible to reason statically and in a modular fashion so that the system would not run "out of memory"? Alternatively, can the system

predict what constraints should be placed on inputs or what storage capacity is needed to avoid the problem?

2. Is it possible for the reasoning system to be modular, yet sufficiently precise with respect to storage constraints?

3. Is it possible to have a formallyspecified storage management scheme that is predictable and efficient, yet allows effective storage utilization?

This dissertation addresses and answers each of these questions in the affirmative, by discussing for the first time a formal system for specification and reasoning of storage capacity. The rest of this chapter is organized into the following sections. Section 1.1 contains background on alternative storage management techniques used in computing theory and practice and their limitations. Section 1.2 describes the objectives of this dissertation in detail. Section 1.3 contains an outline.

1.1 Alternative Storage Management Schemes

Commonly used techniques for storage management can be broadly classified into those for static storage management and those for dynamic storage management. In static storage management, each programming object is "bounded statically" to a maximum size and is limited to be within this fixed capacity during its lifetime. In dynamic storage management, each object is "unbounded" and is allowed to grow arbitrarily. The objects share global storage. Component-based textbooks such as [24] describe bounded and unbounded versions of components.

Regardless of particular memory management schemes and the kinds of components used, the questions raised in the previous section arise. This is because all practical systems work within bounded capacity and in most cases, there are no natural bounds on inputs to most systems. Even simple systems use module constructs and procedures, introducing runtime storage issues such as procedure call overheads and local variables

that enter and leave the scope. Use of bounded or unbounded objects influence only where the storage limitation questions arise, within an object or within the global pool. The particular schemes used affect complexity of the specification and reasoning techniques to be employed. The schemes also affect the overall performance of the system, including its efficiency, predictability, and effectiveness of storage usage.

In the static storage management scheme, the total available storage needs to be partitioned among the objects of a system a priori, based on application-specific knowledge and experience. Static storage management is conceptually simple, and leads to programs that are easier to understand. Since no storage is allocated or deallocated during execution, this scheme introduces no overhead in terms of response time for a storage request and results in maximum predictability and efficiency. But this approach results in poor utilization of the storage and demands that it be possible to determine storage needed for every programming object a priori. To improve storage utilization and to choose pre-allocation sizes of objects more appropriately, it may be necessary to do complex analysis of runtime patterns based on empirical studies and statistical analysis [16]. Even then, for complex applications that interact with large numbers of inputs from the external environment, it may be impossible to predict storage requirements of individual objects.

In dynamic storage management, each object is allowed to grow arbitrarily. All objects share global storage. Dynamic storage management is programmer-controlled in C/C++ and garbage collection-based in Java. In the first case, programmers explicitly allocate and deallocate storage for objects, using for example **new** and **delete**, respectively, in C++. Given total control on storage management, programmers have greater flexibility in

	Storage Utilization	Predictability	Efficiency	Usability/Productivity
Static Allocation Mechanism	Low	High	High	Low
User Controlled Mechanism	High	Medium	Medium	Medium
Garbage Collection	Medium	Low	Medium	High

Figure. 1: A Comparison of various storage management schemes

terms of storage manipulation, and can develop highly sophisticated programs that are efficient and that utilize memory optimally. The predictability in programmer-controlled mechanisms, however, is poorer than in static storage management. It depends on the allocation algorithms and storage usage pattern of the program, which determines the fragmentation characteristics of the storage [17]. Releasing the same memory twice and accessing an illegal memory location, are among the notorious problems that are hard to

debug in programmer-controlled memory management. Storage-related errors in third party libraries are even harder to locate in this scheme.

Garbage collection (GC), an alternative dynamic memory management scheme, is a process whereby the system identifies unused or out of scope objects and collects their storage without explicit intervention from programmers. GC relieves programmers from the nuance of memory tracking and deallocation, and eliminates related bugs. It enables them to focus on the functionality of the application, and hence improves productivity. Also, by concealing memory management details from the programmers, GC covers security aspect as well. But GC is poor in storage utilization compared to programmer-controlled storage management. Most of the GC implementations are conservative, and hence they can not guarantee the availability of the storage even if it is available at the time of a request. The response time for a request depends on the runtime characteristics, and this introduces predictability problems in using GC [18]. Experimental results [18] show that, compared to programmer controlled memory management, the Boehm-Demers-Weise conservative garbage collector on average is 11 times slower.

1.2 Dissertation Objectives

It is important to establish that software systems work within specified storage constraints because, ultimately, all practical software systems need to function within a fixed storage capacity that is determined by the underlying environment. The ability to reason about storage aspects is a fundamental requirement because improper handling of dynamic storage can often lead to devious and difficult-to-detect software errors that can sabotage critical missions. Storage handling also has tremendous implications on the predictability of real-time systems. It is therefore essential to be able to use and reason about storage with ease, but without compromising efficiency or predictability.

For software systems composed of components, often developed independently and facilitated by most modern languages, storage handling raises additional complexities. In these component-based systems, it must be possible to establish locally that each of the participating components is correct with respect to its specification[9,19]. When the components are combined to form a larger system, it will then be necessary only to show that the composition is correct; participating components would not have to be re-verified. Storage handling complicates modular reasoning because storage is a global resource shared by all (or groups of) objects. For safe and secure use of objects in systems that facilitate easy component sharing (e.g. Java), storage-related issues must be carefully addressed. These observations lead to the following objectives for this dissertation, each of which corresponds directly to a question raised in the introduction.

It is essential to have a formal system that accounts for storage limitations and facilitates specification and reasoning of component-based systems in a modular fashion. The system should allow alternative memory management techniques to be employed. This is the first objective of this dissertation.

In general, specification and reasoning of functional behavior and storage conformance issues must be kept separate, yet related. Whereas storage constraints are normally implementation influenced, behavioral specifications should be abstract without any implementation bias. This dilemma leads to the second objective of this dissertation - to develop a framework that allows accurate, yet alternative levels of abstract specifications of storage without compromising the abstraction desired for specification of functional behavior.

Flexibility of the storage mechanism for allowing users to optimize storage utilization, with guaranteed predictability, is another important consideration in storage handling. The underlying storage mechanism should also give the software developers ability to manage the storage at runtime efficiently. Development of a storage management scheme

that combines predictability, efficiency, and optimal storage utilization is a third objective of this dissertation.

The Thesis

The contribution of this dissertation to the field of computer science is in raising storage-related issues in practical component-based software development, and proposing a framework for the specification and reasoning of such systems. In particular, it defends the following thesis :

1. There is a framework and formal system that makes it possible to reason statically and in a modular fashion that a component-based system would not run "out of memory", given constraints on input parameters and maximum available storage capacity. Alternatively, the system can predict what constraints should be placed on inputs or what storage capacity is needed for the system to avoid the problem.

2. It is possible for the reasoning system to be modular, yet sufficiently precise with respect to storage constraints, by allowing storage constraints to be specified at alternative levels of abstraction.

3. It is possible to have a formally specified storage management scheme that is predictable and efficient, yet allows effective storage utilization. The scheme introduces a log factor of storage and execution efficiency loss in the worst case. However, if suitably parameterized, the losses can be brought down to constant factors for typical usage.

1.3 Organization

The rest of this disseration is organized into the following chapters.

Chapter 2, titled ***Alternative Approaches for Specification of Storage Bounds*** presents different types of storage strategies and their specifications. It also raises issues and limitations of these strategies

Chapter 3, titled ***Introduction to Storage Specification and Reasoning of Unbounded Data Abstractions*** presents key ideas of storage specification. Also, in this chapter new notations for specifying storage in RESOLVE discipline are introduced, along with several storage specification examples. Using the storage specification notation proposed in this chapter, reasoning about the correctness of programs with respect to storage is also presented with examples.

Chapter 4, titled ***Intermediate Levels of Abstraction for Storage Constraint Specification*** demonstrates the need and shows that functional and storage behaviors can be specified at different levels of abstraction for capturing precise, implementation-related, storage constraints.

Chapter 5, titled ***Hierarchical Storage Manager*** presents a storage management mechanism that provides the desirable performance characteristics of efficiency, predictability, flexibility, and superior capacity utilization.

Chapter 6, titled ***Conclusions*** summarizes the results of this research work, and outlines future research directions.

Chapter 2

ALTERNATIVE APPROACHES FOR SPECIFICATION OF STORAGE BOUNDS

This chapter explains alternative approaches for describing storage constraints on objects. It discusses formal specification of concepts that place limits on the bounds of individual or groups of objects. Some of these specifications allow the bounds to be set statically, others allow the bounds to be set at runtime. However, all of them demand that global storage be partitioned in a rigid fashion among the objects of a program.

Section 2.1 explains a behavioral specification of stacks that does not include storage bounds. Section 2.2 explains the specification of a stack type that sets static storage bounds on every object. Section 2.3 presents a stack specification that provides flexibility to set different storage bounds on different objects after the instantiation and before usage. The next section, 2.4, generalizes this specification to change the sizes of the objects of a particular type, in this case it is a stack, within the bounds of a collective bound. The last section, 2.5 summarizes the advantages and disadvantages of these specifications. And, also at the end of this section a discussion is presented to demonstrate the need for the formal storage specification approach in Chapter 3.

2.1 RESOLVE Component Specification Example

As a prelude to the discussion on alternative techniques for specifying bounds, first we introduce a traditional, behavioral specification for a concept stack. This concept, termed Stack_Template, is given in Figure 2.1 in RESOLVE specification notation. Without loss of generality, we use RESOLVE notations

throughout this dissertation, though any of a number of other formal methods such as Larch, VDM, or Z could have been used equally as well. Stack_Template contains behavioral descriptions of an Abstract Data Type (ADT) stack and operations that manipulate objects of type Stack. A RESOLVE **concept** includes a **context** section that contains a list of "imports" and an **interface** section that lists "exports". The **global context** section defines fixed coupling of this module to the others in a shared library. In the Stack_Template specification given in Figure 2.1 for example, Standard_Integer_Facility has been imported as the type Integer is needed in the specification of Depth_Of opertaion furnished by the specification. The **parametric context** defines parameters that must be supplied by clients to create an instance of generic Stack_Template. Here the type of the entries in the stack is a generic parameter. The **local context** section introduces, when needed, convenient local mathematical definitions that make the specification easier to comprehend.

The **interface** section of the concept, formally describes exported types and operations. Each program type (family), the type Stack in the present example, is explained using a standard mathematical model. RESOLVE specifications typically use a combination of standard mathematical models such as integers, sets, functions, relations, tuples and strings. Though it is possible to define and use new mathematical models in RESOLVE modules, to keep the specification understandable, standard ones are typically used where possible. Standard notations on these models are then used in explaining operations. In the example given in Figure 2.1, the object type stack was modeled using a mathematical string of type Entry. Mathematical strings are similar to sequences [21], but are simpler in that they do not include notations for representing positions.

```
concept Stack_Template
      context
            global context
                  facility Standard_Integer_Facility
            parametric context
                  type Entry
      interface
            type Stack is modeled by string of Entry
            exemplar s
            initialization
                  ensures s = empty_string
      operation Push(
                  alters s: Stack
                  consumes x: Entry
                  )
                  ensures s = <#x> * #s
      operation Pop(
                  alters s: Stack
                  produces x: Entry
                  )
                  requires |s| > 0
                  ensures #s = <x> * <s>
      operation Depth_Of(
                  preserves s: Stack
                  ): Integer
                  ensures Depth_Of = |s|
end Stack_Template
```

Figure 2.1 An Unbounded Stack specification

In the Stack_Template specification, the **initialization ensures** clause, together with **exemplar,** specifies that every stack is initially modeled by the empty_string. Following initialization, specification of operations that can be performed on a stack concept is given. The effect of each operation is specified using a **requires** clause (precondition) and an **ensures** clause (postcondition). Each of these is an assertion about the values of the mathematical model of the operation's parameters. Absence of a clause means that the assertion is **true**. Mathematically, an operation is a partial relation on the space of input and output values of the parameters. The requires clause tells where the relation is defined, and the ensures

clause defines it there. Together these two clauses specify the contract between the client and implementation for each operation. If a client calls an operation in a state in which requires clause holds for actual parameters, then the implementer guarantees that the operation will return in a state in which the ensures clause holds a true value. But, if the requires clause does not hold when the call occurs, then the implementer makes no guarantees at all.

There are three operations specified for the concept Stack_Template. The specification of operation Push has two parameters; **alters** s of type Stack, and **consumes** x of type Entry. These two parameters are abstract parameter modes in RESOLVE specification notation. The alters-mode parameter is changed by executing the operation and the ensures clause specifies its new value. A consumes-mode parameter has a legal value after the operation, but the value is left to the implementer's choice. But its old value is relevant to the operation's effect. Push has no requires clause, because the current specification assumes that every stack is unbounded. In the ensures clause, a parametric name stands for its value at the end of the call, while its name with prefixed # (pronounced "old") stands for the value of that parameter at the beginning of the call. The ensures clause of Push operation states that the new value of the stack is the old value of x concatenated with the old value of s. Here, * denotes string concatenation. The operation Pop **alters** its parameter stack s and **produces** a new value in the parameter x. The old value of x is unknown. The Depth_Of operation **preserves** stack s and returns its current depth. In RESOLVE, *swapping* is the basic data movement mechanism. The default use of swapping for data movement results in abstract component designs that are subtly different from traditional ones, and admit more efficient implementations. Every component that exports a type also provides swap and three other operations that are explained below:

1. For each local variable, the initialize operation is invoked automatically at the beginning of the scope where the variable is declared. It gives the variable an initial value, which is specified in the **initialization ensures** clause of the type specification.

2. The finalize operation is invoked only at the end of the scope where its argument is declared. There is no abstract effect of this operation other than the disappearance of the entity.

3. The swap operation (invoked using the infix :=: operator) exchanges the values of its two arguments. This is the basic data movement operation in every object, instead of assignment. It is very efficient and easy to specify formally.

4. The Clear operation gives a variable an initial value, similar to initialization as specified in the initialization ensures clause of the type specification.

2.2 Bounded Plastic Components

A RESOLVE specification of Bounded_Plastic_Stack_Template concept is given in Figure 2.2. This concept is parameterized by type Entry and Max_Depth. Here, the size of each stack is fixed apriori by the value of Max_Depth.

In the interface specification, requires clause of Push operation states an obligation that the length of the stack has to be less than the value of the Max_Depth variable. This is a pre-condition on operation Push. This means that before pushing an item onto the stack the size of the stack should be less than its maximum size, so that it can hold at least one more item. In this example Push is the only operation that involves the storage-specific variable Max_Depth in its specification.

```
concept Bounded_Stack_Template
      context
            global context
                  facility Standard_Integer_Facility
      parametric context
            constant Max_Depth: Integer
                  restriction Max_Depth  > 0
            type Entry
      interface
            type Stack is modeled by string of Entry
            exemplar s
            initialization
                  ensures |s| = 0
                  constraint |s| <= Max_Depth
      operation Push(
                  alters s: Stack,
                  consumes x: Entry
                  )
                  requires |s| < Max_Depth
                  ensures s = #s * #x
      operation Pop(
                  alters s: Stack,
                  produces x: Entry
                  )
                  requires |s|  > 0
                  ensures #s = s * x
      operation Depth_Of (
                  preserves s: Stack
                  ): Integer
                  ensures Depth_Of = |s|
      operation Max_Depth(
                  ): Integer
                  ensures Max_Depth = Max_Depth
end Bounded_Stack_Template
```

Figure 2.2 Bounded Stack Concept

```
facility BSF is  Bounded_Stack_Template(Integer,1000)
realized by Static_Array_With_Top_Index

procedure Example
      context variables
            s1,s2: BSF.stack
      begin
      --Bounds of s1, and s2 are intially set to 1000
      --procedure body
end Example
```

Figure 2.3 Bounded integer stack facility and its usage

Before Bounded_Stack_Template can be used, it must be initialized. An

instantiation of a **facility** declaration of a stack of integers with 1000 elements of maximum size is given in Figure 2.3. At the time of instantiation, a realization for the concept should also be picked. This flexibility to choose any specific realization for a concept allows programmers to have different instances of the same concept supported by different realizations in the same program. In Figure 2.3, Static_Array_With_Top_Index denotes a static array-based realization of Bounded_Stack_Template such as the one given in Figure 2.2. A formal specification of Static_Array_Template and an array-based realization are given in Appendix A.

2.3 Bounded Ceramic Components

In the bounded plastic version of stack all the elements of facility are of size Max_Depth. It may not be the case always that all the instances of Integer stack need to be of size 1000. It would add greater value and optimize the storage utilization if the facility can offer flexibility to change the size of the instance at the time of usage. A bounded ceramic version of the stack specification that supports this need is given in Figure 2.4. This version is called *ceramic* stack template. The upper bounds of all integer stack objects from the facility in Figure 2.3 are fixed to be 1000. The ceramic version of the specification, while still forcing a bound on each object, allows different stack objects to use different upper bounds.

```
concept Bounded_Ceramic_Stack_Template
      context
            global context
                  facility Standard_Integer_Facility
            parametric context
                  type Entry
      interface
            type Stack is modeled by (
                  contents: string of Entry,
                  Max_Depth: integer
                  )
            exemplar s
            initialization
                  ensures |s.contents| = 0 and
                        s.Max_Depth = 0
                  constraint |s.contents| <= s.Max_Depth
      operation Set_Max_Depth(
                  alters s: Stack
                  preserves max_depth: Integer
                  )
                  requires max_depth  > 0  and
                        s.Max_Depth = 0
                  ensures s.Max_Depth = max_depth and
                        s.contents = #s.contents
      opertaion Get_Max_Depth(
                  preserves s: Stack
                  ): Integer
                  ensures Get_Max_Depth = s.Max_Depth
      -- specifications of Push, Pop, and Depth_Of are
      -- similar to those in Figure 2.3 except that now
      -- s.Max_Depth is used instead of Max_Depth
end Bounded_Ceramic_Stack_Template
```

Figure 2.4 Bounded Ceramic Stack specification

The similarities and differences between Bounded Plastic stack and Bounded Ceramic stack conceptualization are important to note. The stack in the Bounded Ceramic case is modeled by a mathematical tuple <String, Max_Depth>. This means that every stack instance has to be thought of as a string of entries with an associated Max_Depth. While the basic operations on stack are still same, there is an addition of two more procedures to the interface in this specification. The procedure Set_Max_Depth allows changing the size of the stack. But the requires clause of this operation specifies that this resizing can be done only once, i.e.,

when Max_Depth is equal to zero. The Ceramic_Array_Template specification and a realization body based on this specification are given in Apendix A.

Both plastic and ceramic versions of specifications in which maximum depth of individual objects are bounded are simple for understanding and reasoning. Typical array-based implementations are efficient and predictable. However, storage utilization is far from optimal, because the sizes of the stacks have to be fixed prior to usage.

It is important to note the differences between plastic bounded components and ceramac bounded components to use them appropriately. The sizes of the objects in a plastic bounded component are set at the time of facility instantiation.

```
facility CSF is  Bounded_Ceramic_Stack_Template(Integer)
realized by Bounded_Ceramic_Array_With_Top_Index

procedure Example
context variables
      s1,s2: CSF.stack
begin
      --Initially, sizes of the stacks are not set
      --Set the size of s1 to 100
      Set_Max_depth(s1,100)
      --Set the size of s2 to 1000
      Set_Max_Deoth(s2,1000)

      --Sizes of these two variables can not be changed
      --procedure body
end Example
```

Figure 2.5 Bounded Ceramic integer stack facility

For example, in Figure 2.3 this size is 1000. In the case of a bounded ceramic component, the facility instantiation does not take the size of the component as a parameter, but users can set the size at object level after instantiating them. Figure

2.5 gives an example of ceramic integer stack facility instantiation. This flexibility at finer granularity levels is desirable for storage utilization and programming convenience.

It is possible to implement plastic component behavior with ceramic versions. For example, in the case of stack components, the size of the stack object can be set at the time of variable instantiation and not be allowed to alter afterwards by the user. In other words, the behavior of plastic components is subsumed by the behavior of ceramic components.

2.4 Communal Components

From the discussion on bounded components in the previous subsection it is evident that the instantiated objects typically need to reserve unnecessary storage that they may never use, because the sizes are fixed. Communal Storage Management (CSM) is one way of dealing with the storage underutilization problems posed by bounded components and provide required flexibility for dynamic changes in storage requirements. The basic idea in communal storage management is to allocate different pools of storage for objects from different facilities. For example, all the objects of an instance of 'Stack of integer' would use the same pool of storage. There is a combined upper limit on the amount of storage that all the objects can use. But each object can grow dynamically within this combined upper limit. Figure 2.6 shows a snapshot of a communal storage management scheme. In this figure, the four instances of Stack share the storage. Each of these stacks can grow dynamically, as long as the combined size of these four, i.e., $|s1|+|s2|+|s3|+|s4|=16000$, is less than the storage allocated for the type, i.e. 20000.

A formal specification in RESOLVE notation for a communal stack concept is given in Figure 2.7. This concept is parameterized by type Entry, as specified in

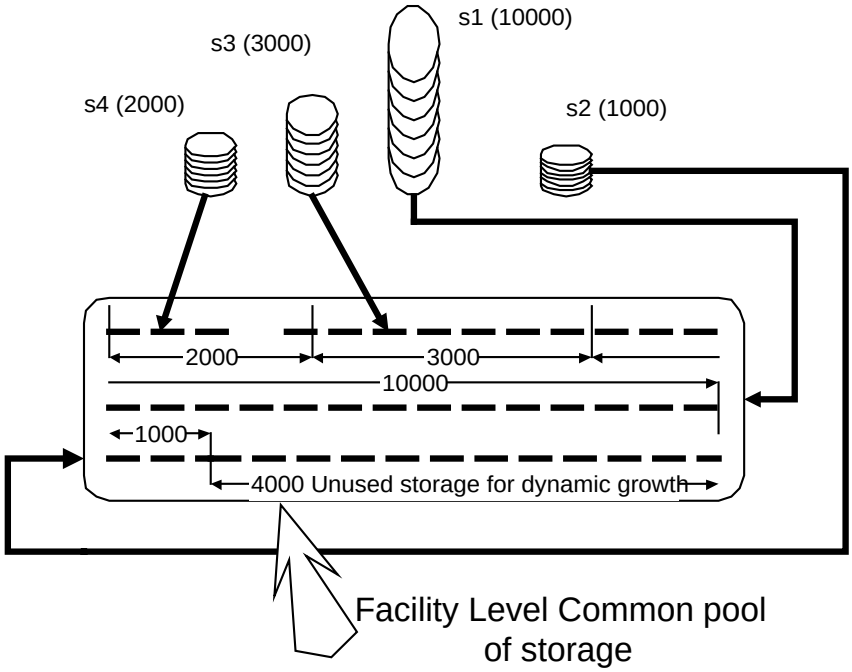


Figure 2.6 A typical example of CSM based component

the parametric context. There is a module level constant Total_Max_Depth, declared in the parametric context, which should be supplied by the programmer at the time of instantiation. This constant specifies the size of the pool of storage that should be pre-allocated for all the instances of the objects of that type. The constraint on this constant states that it has to be greater than zero. In the interface section of this concept, Stack is modeled by a mathematical string of Entry. Also, in this section, a mathematical definition of operation Total_Depth is given**.** This definition computes the sum of the sizes of all instances of Stack at any given instant. To deal with the arbitrary nature of the number of instances,

two notations **Last_Specimen_Num**, and **Denoted_By(i)** are introduced in this definition. The notation **Last_Specimen_Num** denotes a conceptual number that is assigned to the most recent Stack object, where as **Denoted_By(i)** identifies a particular Stack object with the sequence number i. These two constructs help to deal with objects without referring to their names. The value returned by this operation has to be less than or equal to that of the Total_Max_Depth constant as specified by the constraint in the definition of this math operation. The intialization specifies that every object initially would be of size zero. Notable changes in the operations of this specification are the requires clause of Push operation and the addition of Get_Rem_Capacity operation. In the specification of requires clause of the Push operation the constraint states that Total_Depth < Total_Max_Depth, i.e., the combined sizes of all the instances of the stacks must be less than the size of the storage pool. The size of the current instance need not be explicitly set. The size limitations are taken care of by the requires clause in the Push operation specification. The operation Get_Rem_Capcity gets the room left for additional growth. The math operation Total_Depth defined comes in handy in specifying the constraints on these two operations. An example facility declaration is given in Figure 2.8.

```
concept communal_Stack_Template
      context
            global context
                  facility Standard_Integer_Facility
            parametric context
                  type Entry
                  constant Total_Max_Depth: Integer
                        restriction  Total_Max_Depth > 0
      interface
            type Stack is modeled by string of Entry
            math operation Total_Depth : integer
                  definition
                      Total_Depth =∑ i=1^Stack.Last_Specimen_Num
                              (|Stack.Denoted_By(i)|)
                        constraint
                        Total_Depth <= Total_Max_Depth
            exemplar s
                  initialization
                        ensures |s| =  0
      operation Push(
                  alters s: Stack,
                  consumes x: Entry
                  )
                  requires  Total_Depth < Total_Max_Depth
                  ensures s = #x  * #s
      operation Pop(
                  alters s: Stack,
                  produces x: Entry
                  )
                  requires |s| /= 0
                  ensures #s = x *s
      operation Depth_Of (
                  preserves s: Stack
                  ) : Integer
                  ensures Depth_Of = |s|
      operation Get_Rem_Capacity(
                  ): Integer
                  ensures Get_Rem_Capacity =
                        (Total_Max_Depth - Total_Depth)
end Communal_Stack_Template
```

Figure 2.7 Communal Stack Specification

CSM-based storage handling improves storage utilization considerably without significantly compromising efficiency and predictability. For the specification of

CSM-based components, additional language constructs are required, which

```
facility CommSF is  communal_stack_Template(Integer,10000)
realized by Communal_Array

procedure Example
context variables
      s1,s2: CommSF.stack
begin
      --Initially, sizes of the stacks are not set, but
      --items on s1, and s2 can be pushed until there
      --combined size is less than or equal to 10000

      --procedure body
end Example
```

Figure 2.8 Communal integer stack facility

makes the specification slightly more complex.

2.5 Discussion

Bounded components are mainly used in solving the predictability and performance requirements in hard real-time systems. Also, specification and verification of these components are relatively simple, but they result in poor utilization of storage. Systems with bounded (pre-allocated) components can fail even when the necessary storage is available because the unused storage is reserved for other components. Ceramic versions avoid this problem to some extent by allowing the instances to pre-allocate the required storage. Even though it seems that the differences are not that great between bounded and ceramic components, from the Client perspective it has greater impact on the usability. In a program where components with different bounds have to be used, for each different size a different facility has to be declared. Ceramic components avoid

this situation, by allowing the client to set the sizes of the components at the time

	Predictability	**Storage Utilization**	**Specification Complexity**
Bounded Components	Absolute (high)	Low	Low
Communal Storage	High	Medium	Medium
Global Storage	Low	High (optimal)	High

Figure 2.9 Bounded strategies comparison with Global storage management

of object instantiation, as opposed to facility instantiation.

CSM-based components provide better utilization of storage by supporting dynamic growth within an upper limit of the combined storage.

All the bounded mechanisms presented divide the storage into segments and thus result in poor utilization of storage at the global level. Each one of these approaches, when applied, results in a different specification of a concept. For example, the three specifications of Stack using three approaches are different. Even though they support the same concept Stack, it is not possible to use them in an interchangeable way without affecting the behavioral verification. The formal storage specification approach presented in the next chapter addresses these issues and provides a solution framework.

Chapter 3

INTRODUCTION TO STORAGE SPECIFICATION AND REASONING OF UNBOUNDED DATA ABSTRACTIONS

The objective of this chapter is to describe a framework for specification of global storage constraints on unbounded components. It also presents a modular technique for formal verification of storage constraints in an implementation that relies on storage specification of reused components. The technique is a natural extension of the behavioral specification and reasoning mechanism. Section 3.1 of this chapter presents an example of specification and behavioral verification. Section 3.2 extends the example with storage specification and verification. Section 3.4 presents an example of storage specification for layered components. In the following section, i.e., Section 3.5 storage specification examples for some of the base types and other complex user defined types are given. Finally, section 3.6 presents a discussion on the soundness and relative completeness of the proposed framework with respect to the storage constraints specification.

3.1 An Introduction to Behavioral Reasoning

This section describes the technical aspects of behavioral reasoning in the RESOLVE framework. The reasoning is done using the indexed method proposed by Heym [17], which is an alternative to the more traditional formal method of reasoning [22]. A key advantage of the indexed method is that it makes verification more "natural" to programmers. In addition, the complexity of assertions for nested control structures is lower. Heym has also formally established the soundness and completeness of the indexed method [17]. For verification of storage-related aspects, we extend this basic method with reasoning rules that use capacity specification of objects in addition to behavioral specification. As a first step towards presenting storage specification and verification mechanisms, we present behavioral verification of a Reverse Queue procedure using Heym's indexed method. Figure 3.2 contains a specification of the operation Reverse example, which is presented as an **enhancement** to the Queue abstract data type given in Figure 3.1. Enhancements allow new operations to be defined and used on objects in addition to these basic or primary

operations. In the specification, a Queue is modeled as a math String of Entities. In the specification of Reverse, $\#q^R$ denotes the reverse of string #q.

```
concept Queue_Template
        context
                global context
                        facility Standard_Integer_Facility
                parametric context
                        type Entry
        interface
                type Queue is modeled by string of Entry
                exemplar q
                initialization
                        ensures  q = empty_string
        operation Enqueue(
                                alters    q : Queue
                                consumes x : Entry
                                )
                                ensures   q = #q * <#x>
        operation Dequeue(
                                alters    q : Queue
                                produces  x : Entry
                                )
                                requires  |q| > 0
                                ensures   #q = <x> * q
        operation Length_Of(
                                  preserves q : Queue
                                  ) : Integer
                                  ensures Length_Of =  |q|
end Queue_Template
```

Figure. 3.1 Unbounded Queue Specification

```
enhancement Reverse_Capability for Queue_Template
     operation Reverse(alters q: Queue)
        ensures q=#q^R
end Reverse_Capability
```

Figure 3.2 Queue_Reverse Enhancement

An iterative realization body Queue Reverse_Capability that uses the concept Stack is shown in Figure 3.3. In modular verification of the correctness of the body of Reverse with respect to its specification, specifications Queue_Template and Stack_Template are used. The behavioral verification holds regardless of the particular implementations chosen for Queue_Template and Stack_Template. Without the loss of generality, a List-Based implementation has been chosen for the concept Stack. Figure 3.4 illustrates the relationships among the modules involved in this example. A later section contains realization of a recursive implementation of Reverse.

```
realization body Iterative for Queue_Reverse_Capability
        context
            local context
                    Facility SF is Stack_Template(Entry) realized by List-Based
        procedure Reverse(alters q : Queue)
            context
                variables
                    x : Entry
                    s  : SF.Stack
            begin
                maintaining #q=s^R * q
                decreasing |q|
                while (Length_Of(q) > 0) loop
                    Dequeue(q,x)
                    Push(s,x)
                end

                maintaining #q^R=q * s
                decreasing |s|
                while (Depth_Of(s) > 0) loop
                    Pop(s,x)
                    Enqueue(q,x)
            end
        end Reverse
end Iterative
```

Figure 3.3 An Iterative Implementation of Queue Reverse Opearation

In Figure 3.3, there are two loops in the body of Reverse. For each of these loops, programmers need to supply a **maintaining** or loop invariant assertion and a **decreasing** or a progress metric that is used to show loop termination. Figure 3.5 shows the first step in the indexed method, whereby each program sate is numbered starting from step zero. During the execution of the operation Reverse, the program moves from state 0 to state 8. It may not take the same path for every input value. For example, program may go to step 1 from step 0, or to step 4, depending on the length of the queue. If the length of the queue is greater than zero then the program state moves from state 0 to state 1, otherwise it goes to state 4. Also, for the program to go to state 1 from state 0, the operation Length_Of has to be executed as part of

the condition evaluation in the while loop. Before the operation Length_Of can be executed, the requires assertion of the operation has to be true. The requires clause for the operation Length_Of is true, and hence proving it is trivial. A better example is the transition from state 1 to state 2. For this transition to happen, the operation Dequeue has to be executed. To execute this operation the requires clause of the operation Dequeue to be satisfied. This can be done using the facts available in this state, i.e. state 1. To get to this state the length of the queue has to be greater than zero. Since the requires clause of the operation Dequeue needs this condition to be satisfied, it can be proven that the transition from state 1 to state 2 can be proven. This process has to be repeated over all the states of the program to verify it.

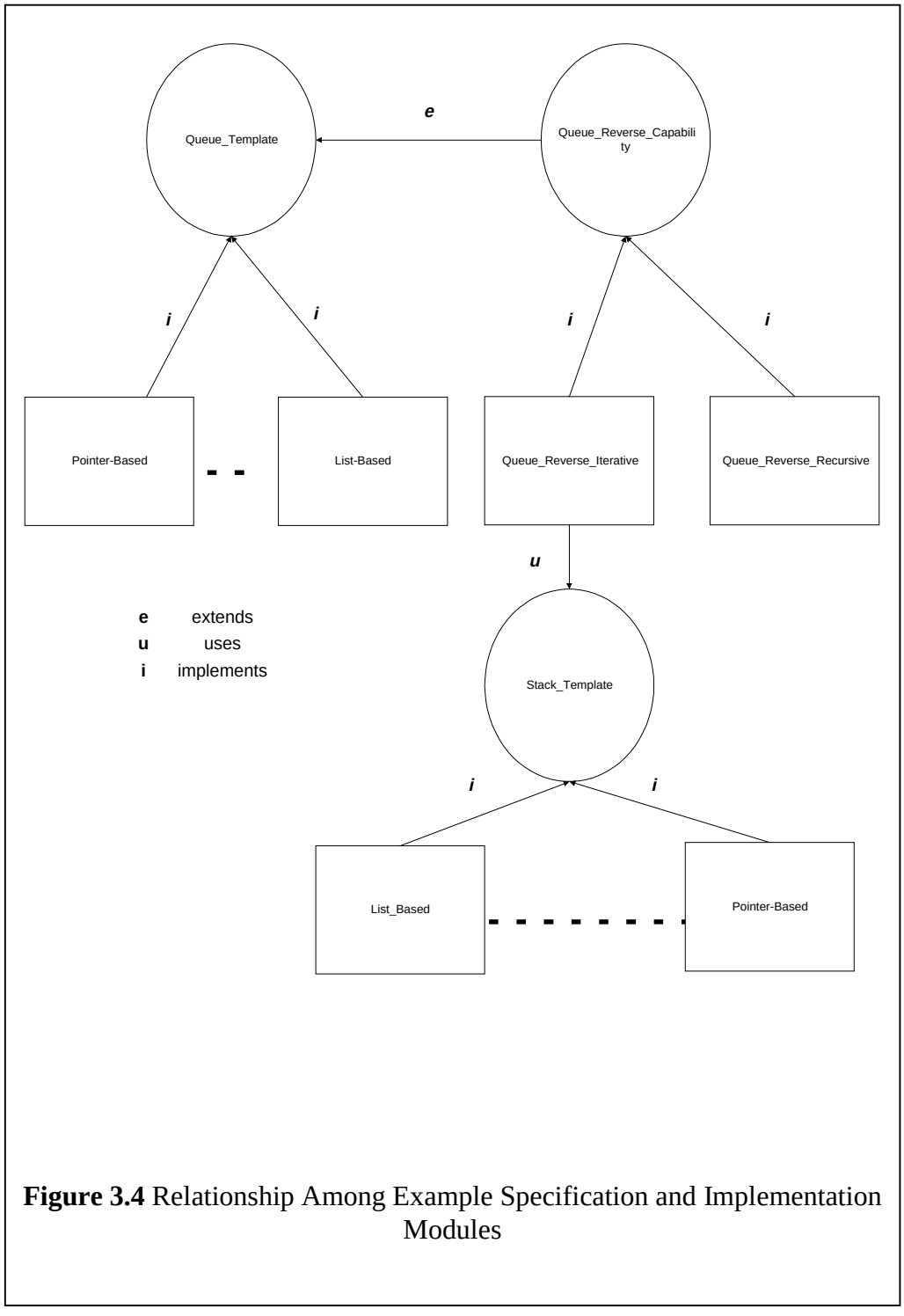


Figure 3.4 Relationship Among Example Specification and Implementation Modules

This process is generalized in natural reasoning methods as a two-step process in which

1. Record all the local information in a *symbolic reasoning table***.**

2. Establish the code's correctness by combining the recorded information into the code's *verification conditions*.

Step 1 above is a symbol-processing activity that is no more complex than compiling and can be done automatically. Assertions recorded for the current state arise from three questions:

1. Under what condition can the program get into this state?

2. If the program gets into this sate, what do we know about the values of the objects?

3. What assertions have to be proved for the program to move to the next state?

Once all these assertions are recorded, they are composed appropriately to form a verification condition that can be proved *true* later. There is one verification condition for each obligation of the form:

assumptions ⇒ obligation

The soundness of natural reasoning depends upon using only the following assumptions in the proof of the obligation for state k:

- (path condition for state i) ⇒ (facts for state i), for every i where 0<=i<=k and

- path condition for state k

In our example, in order to discharge the proof of the obligation in state 1 of the operation *Reverse*, i.e., $|q_1| > 0$ **,** it can be assumed that:

- (**true** ⇒ (s_0 = Empty_String **and** x_0 = Initial_Value) **and**
- ($|q_0| > 0 \Rightarrow q_0 = s_1^R * q_1$)

Following these steps the reasoning table generation and the proofs for the operation Reverse are done. Figure 3.6 lists *facts* and *obligations*, where facts are what after each statement the

developers of Reverse operation can assume. It also includes a path condition for each statement. The subscripted variables in the table represent the value of a variable at that state of the program. For example, the symbol x_1 represents the value of variable x in state 1. Figure 3.7 presents a table with the proof mechanisms for the obligations in all states. It is important to note that nowhere in the verification implementation are aspects of the constituent components (Stack and Queue) referred. Only the abstract specifications are used to verify the program. This is the key for scalability of a verification system. Also, another important aspect to notice is that for the Push and Enqueue operations the verification is done without regard to the storage.

```
procedure Reverse(alters q:Queue)
        context
            variables
                s: Stack
                x: Entry
        begin
                                                          0
            maintianing #q = s ^R * q
            decreasing  |q|
            while Length_Of(q) > 0 loop
                                                          1
                Dequeue(q,x)
                                                          2
                Push(s,x)
                                                          3
            end loop
                                                          4
            maintaing #q^R =  q*s
            decreasing |s|
            while Depth_Of (s) > 0 loop
                                                          5
                Pop(s,x)
                                                          6
                Enqueue(q,x)
                                                          7
            end loop
                                                          8
end procedure
```

Figure 3.5 Reverse realization Body

State ment	Path condition	Facts	Obligations
0		1. s_0 = Empty_String 2. x_0 = Initial_Value	$q_0 = s_0^R * q_0$ **(?)**
Maintaining $\#q = s^R * q$ **Decreasing** $\|q\|$ **while** Length_Of(q) > 0 **loop**			
1	$\|q_0\| > 0$	1. $q_0 = s_1^R * q_1$	$\|q_1\| > 0$ **(?)**
Dequeue(q,x) **operation** Dequeue(**alters** q : Queue,**produces** x : Entry) **requires** |q| > 0 **ensures** #q = <x> * q			
2	$\|q_0\| > 0$	1. $q_1 = x_2 * q_2$ 2. $s_1 = s_2$	

Push(s,x)
`operation Push( alters s: Stack,consumes x: Entry)`
`ensures s = #x * #s`			

3	$\|q_0\| > 0$	1. $q_3 = q_2$ 2. $s_3 = x_2 * s_2$	$q_0 = s_3^R * q_3$ **and** termination: $\|q_3\| < \|q_0$ **(?)**
end loop			
4		1. $q_0 = s_4^R * q_4$ 2. $\|q_4\| <= 0$	$q_0^R = q_4 * s_4$ **(?)**
Maintaining $\#q^R = q * s$ **Decreasing** $\|s\|$ While Depth_Of(s) > 0			
5	$\|s_5\| > 0$	1. $q_0^R = q_5 * s_5$ 2. $\|s_5\| > 0$	$\|s_5\| > 0$ **(?)**

Pop(s,x)
`operation Pop( alters s: Stack,produces x: Entry)`
`requires |s| > 0`
`ensures #s = x*s`			

6	$\|s_5\| > 0$	1. $s_5 = x_6 * s_6$ 2. $q_6 = q_5$	

State ment	Path condition	Facts	Obligations
Enqueue(q,x)		**operation** Enqueue(**alters** q : Queue,**consumes** x : Entry) **ensures** q = #q * <#x>	
7	$\|s_5\| > 0$	1. $q_7 = q_6 * x_6$ 2. $s_7 = s_6$	$q_0^R = q_7 * s_7$ **(?) and** termination: $\|s_7\| < \|s_4\|$
end loop			
8		1. $q_0^R = q_7 * s_7$ 2. $s_8 = s_7$ 3. $\|s_8\| = 0$	$q_8 = q_0^R$ **(?)**

Figure 3.6 Specification-based Reasoning Table for Reverse

State	Obligation	Proof
0	$q_0 = s_0^R * q_0$ **(?)**	$s_0^R * q_0$ $\Rightarrow$q0, Since s0 is empty string $\Rightarrow q_0 = s_0^R * q_0$ is **true**
1	$\|q_1\| > 0$ **(?)**	$q_1 = q_0$, (fact1 from state 1) $\|q_0\| > 0$ $\Rightarrow \|q_1\| > 0$ is **true**
3	$q_0 = s_3^R * q_3$ **and** $\|q3\| < \|q2\|$ **(?)**	$s_3^R * q_3$ $\Rightarrow s_2^R * x_2 * q_3$ (fact 2 from State 3) $\Rightarrow s_2^R * x_2 * q_2$ (fact 1 State 3) $\Rightarrow s_2^R * q_1$ (fact 1 State 2) $\Rightarrow s_1^R * q_1$ (fact 2 State 2) $\Rightarrow q_0$ (fact 1 State 1) $\Rightarrow (q_0 = s_3^R * q_3)$ is **true** termination: $\|q_3\| < \|q_1\|$ $\Rightarrow \|q_2\| < \|q_1\|$ (fact 1 from state 3) and (fact1 from state 2)
4	$q_0^R = q_4 * s_4$ **(?)**	Since,$\|q_4\| = 0$ $q_4 * s_4$ $\Rightarrow s_4$ $\Rightarrow q_0^R$ (fact 1 of state 4) $\Rightarrow q_0^R = q_4 * s_4$ is **true**
5	$\|s_5\| > 0$ **(?)**	True from the fact list
7	$q_0^R = q_7 * s_7$ **(?)** **and** $\|s_7\| < \|s_4\|$ **(?)**	$q_7 * s_7$ $\Rightarrow q_6 * x_6 * s_7$ (fact 1 from state 7) $\Rightarrow q_6 * x_6 * s_6$ (fact 2 from state 7) $\Rightarrow q_6 * s_5$ (fact 1 from state 6) $\Rightarrow q_5 * s_5$ (fact 2 from state 6) $\Rightarrow s_4$ (fact 1 from state 5) $\Rightarrow q_0^R$ (fact1 from state 4) $\Rightarrow q_0^R = q_7 * s_7$ is **true** termination: $\|s_7\| = \|s_6\|$ $\|s_6\| < \|s_5\|$ (fact 1 from state 6) $\Rightarrow \|s_7\| < \|s_5\|$ $\|s_5\| <= \|s_4\|$ (fact 1 from sate 5) $\Rightarrow \|s_7\| < \|s_4\|$
8	$q_8 = q_0^R$ **(?)**	$q_8 = s_4$ (facts, 1,2 and 3 from state 8) $\Rightarrow q_8 = q_0^R$ is **true** (fact 1, and 2 from state 4)

Figure 3.7 Proof Table for the Reverse Procedure

3.2 Storage Specification

This section explains how behaviroal specification and reasoning can be extended to include storage considerations. At the global level it is assumed that there is a finite amount of storage available. The **Max_Capcity** global constant represents this storage limit. During program verification, Max_Capacity would be the size of the RAM or the size of the total virtual memory including RAM. If the underlying operating system uses a demand paged virtual memory mechanism, then this would be the sum of RAM and disk space.

To specify the storage used by an object the notation **uses capacity** is introduced in the interface section of the concept declaration. The value specified by this keyword is equivalent to the storage used by an object at any given instant. A math function **C** (for capacity) from a type to its uses capacity value is defined to compute the storage utilized by an object. The **requires capacity** clause in the interface specification of each operation specifies the storage requirement for that operation. This clause contains both the procedure call overhead as well as the transient storage requirements. The requires capacity clause is also used in the initialization constraints to specify the storage required by that type at the time of initializing an object.

To illustrate these notations and their meaning, an example that shows unbounded stack storage requirements is given in Figure 3.10. In this specification the *uses capacity* assertion is given as $C_{stack} + |s|*C_{Entry\_Overhead} + \sum_{x=Entry} \mathbf{C}(x)$, where IS_ENTRY_OF(s,x). Here C_{stack} is the constant amount of storage required for an empty stack. The constant $C_{Entry\_Overhead}$ represents the storage requirement for adding a new item to the stack. IS_ENTRY_OF(s,x)[1] is a math operation that asserts whether element x is part of the string s or not.

[1] **math operation** IS_ENTRY_OF(str: string of Entry, x: Entry): boolean
definition (∃ alpha, beta: string of Entry (alpha * <x> * beta = str))

concept Stack_Template
context
global context
facility Standard_Integer_Facility
local context
C_{Push} = 2 * $C_{Reference}$ + $C_{Entry\_Overhead}$
C_{Pop} = 2 * $C_{Reference}$
C_{Depth_Of} = $C_{Reference}$ + $C_{Integer}$
parametric context
type Entry
interface
type Stack **is modeled by** string of Entry
exemplar s
uses capacity C_{Stack} + |s|*$C_{Entry\_Overhead}$ + $\sum_{x:Entry}$ **C**(x)
where IS_ENTRY_OF(s,x)
initialization
requires capacity C_{Stack}
ensures s = empty_string
operation Push(**alters** s: Stack
consumes x: Entry
)
ensures s = <#x> * #s
requires capacity C_{Push}
operation Pop(**alters** s: Stack
produces x: Entry
)
requires |s| > 0
requres capacity C_{Pop}
ensures #s = <x> * <s>
operation Depth_Of(**preserves** s: Stack
): Integer
requres capacity $C_{Depthof}$
ensures Depth_Of = |s|
end Stack_Template

Figure 3.8 Unbounded Stack specification including storage constraints

```
concept Queue_Template
        context
        global context
            facility Standard_Integer_Facility
        parametric context
            type Entry
        local context
            C_Enqueue = 2 * C_Reference + C_Entry_Overhead
            C_Dequeue = 2 * C_Reference
            C_Lengthof = C_Reference + C_Integer

        interface
        type Queue is modeled by string of Entry
        exemplar q
                uses capacity C_Queue + |q| * C_Entry_overhead + Σ x:Entry C(x)
                                        where IS_ENTRY_OF(q,x)
    initialization
        requires capacity C_Queue
        ensures |q| = 0
    operation Enqueue(  alters   q : Queue
                        consumes x : Entry
                        )
        requires capacity C_Enqueue
        ensures  q = #q * <#x>
    operation Dequeue(  alters   q : Queue
                        produces x : Entry
                        )
        requires |q| > 0
        requires capacity C_Dequeue
        ensures  #q = <x> * q
    operation Length_Of(    preserves q : Queue
                        ) : Integer
        requires capacity C_LengthOf
        ensures Length_Of = |q|
end Bounded_Queue_Template
```

Figure. 3.9 Unbounded Queue Specification including storage

The implicit operation *swap* is handled in a generic way for all the program types. In RESOLVE the data-movement is done by swapping the pointers, and for representing a pointer only a constant amount of storage, $C_{Reference}$ is required. To swap two values, an

intermediate variable is required, so the requires capacity for a swap operation is always $C_{Reference}$. In RESOLVE, parameter passing is done using *call-by-swap*. This requires, at runtime, storage equivalent to $C_{Reference}$ times the number of formal parameters for that operation. The specifications given in Figures 3.8 and 3.9 show these storage constraints in *local context*s.

In the storage verification process, similar to behavioral verification, the *requires capacity* clause

```
enhancement Reverse_Capability for Queue_Template
    operation Reverse(alters q: Queue)
        requires capacity C_stack + C_Entry + MAX(C_push , C_pop ,
                          C_dequeue , C_Enqueue , C_depthof , C_Lengthof)
        ensures q=#q^R
end Queue_Reverse_Capability
```

Figure 3.10 Queue_Reverse enhancement with storage specification

of a called operation becomes a caller obligation in the state before the call. For ensuring the absence of capacity bounds-related errors, there is an obligation before every operation call that the total storage used so far plus the storage required by the requires capacity clause of the called operation is less than Max_Capacity. To compute the total storage used so far, a mathematical function **ΣC** is defined. **Σ**C is the sum of the storage used by all the program objects in scope at any given point in the program. Specification of Stack and Queue concepts along with their storage requirements are give in Figures 3.8 and 3.9 respectively. Specification of the Reverse operation with storage requirements is given in Figure 3.10. The *requires capacity* clause in the specification is rather oriented towards an iterative implementation of the Reverse procedure. Reasoning about the storage aspects of the Reverse operation is done in the following paragraphs utilizing these specifications.

In the process of reasoning about the Reverse procedure body, we assume that the requires capacity clause is true and proceed to prove that the total requirement for the storage at each point of the program is less than the Max_Capacity. To represent the total requirement of the storage by all the variables in the scope we use the **∑C** notation. A new column listing the storage obligations is added to the specification-based reasoning table, as shown in Figure 3.11. In the state before the local variables come into the scope denoted by I in the figure, it has to

be proved that there is enough storage available for the two variables. Similarly in state F, just before exiting the scope, it has to be proved that the total storage required by all the variables in the scope is less than the Max_Storage.

State ment	Path condition	Facts	Behavioral Obligations	Storage Obligations
I		1. $\mathbf{C}(q_i)+ C_{Stack} + \mathbf{C}_{Entry} + MAX(C_{Push}, C_{Pop}, C_{Dequeue}, C_{Enqueue}, C_{Depthof}, C_{Lengthof}) <=$ Max_Capacity		$\Sigma C_I + C_{Stack} + \mathbf{C}_{Entry} <=$ Max_Capacity **(?)**
	s: Stack x: Entry			
0		**1.** $\|s_0\| = 0$ **2.** $q_0 = q_i$	$q_0 = s_0^R * q_0$**(?)**	$\Sigma C_0 + C_{LengthOf} <=$ Max_Capacity**(?)**
Maintaining $\#q = s^R * q$ **Decreasing** $\|q\|$ **while** Length_Of(q) > 0 **loop**				
1	$\|q_0\| > 0$	1. $q_0 = s_1^R * q_1$ **2.** $x_1 = x_0$	$\|q_1\| > 0$ **(?)**	$\Sigma C_1 + C_{Dequeue}$ <= Max_Capacity**(?)**
Dequeue(q,x) **operation** Dequeue(**alters** q : Queue **produces** x : Entry) **requires capacity** $C_{Dequeue}$ **requires** $\|q\| > 0$ **ensures** $\#q = <x> * q$				
2	$\|q_0\| > 0$	1. $q_1 = x_2 * q_2$ 2. $s_1 = s_2$		$\Sigma C_2 + C_{push}$ <= Max_Capacity**(?)**

Push(s,x)
`operation Push( alters s: Stack,consumes x: Entry)`
requires capacity C_{Push}
`ensures s = #x * #s`				

State ment	Path condition	Facts	Behavioral Obligations	Storage Obligations
3	$\|q_0\| > 0$	1. $q_3 = q_2$ 2. $s_3 = x_2 * s_2$	$q_0 = s_3^R * q_3$ **and** termination: $\|q_3\| < \|q_0\|$ **(?)**	$\Sigma C_3 <=$ Max_Capacity**(?)**
end loop				
4	**not**($\|q_0\| >$ 0)	1. $q_0 = s^R_4 * q_4$ 2. $\|q_4\| <= 0$	$q_0^R = q_4 * s_4$ **(?)**	$\Sigma C_4 + C_{\text{Depth of}} <=$ Max_Capacity**(?)**
Maintaining #s= q * s **Decreasing** |s|				
While Depth_Of(s) > 0				
5	$\|s_5\| > 0$	1. $s_4 = q_5 * s_5$ 2. $\|s_5\| > 0$	$\|s_5\| > 0$ **(?)**	$\Sigma C_5 + C_{Pop} <=$ Max_Capacity**(?)**
Pop(s,x)				

```
operation Pop(alters s: Stack, produces x: Entry)
  requires |s|  > 0
  requires capacity C_Pop
  ensures #s = x*s
```

6	$\|s_5\| > 0$	1. $s_5 = x_6 * s_6$ 2. $q_6 = q_5$		$\Sigma C_6 + C_{Enqueue} <=$ Max_Capacity
Enqueue(q,x)				

```
operation Enqueue(alters q : Queue consumes x : Entry)
  requires capacity C_Enqueue
  ensures  q = #q * <#x>
```

7	$\|s_5\| > 0$	1. $q_7 = q_6 * x_6$ 2. $s_7 = s_6$	$q_0^R = q_7 * s_7$ **(?) and** termination: $\|s_7\| < \|s_4\|$	$\Sigma C_7 <=$ Max_Capacity
end loop				
8	$\|s_5\| <= 0$	1. $s_4 = q_8 * s_8$ 2. $s_8 = s_7$ 3. $\|s_8\| = 0$	$q_8 = q_0^R$ **(?)**	
F				$\Sigma C_F <=$ Max_Capacity

Figure 3.11 Specification-based Reasoning Table with Storage Obligations

Figure 3.12 presents a table that contains proofs for obligations of states I, 0, 1, and 2. Proofs for the rest of the obligations are similar.

State	Proof
I	$\Sigma C_I + C_{Stack} + \mathbf{C}_{Entry} <= Max\_Capacity$ **(?)** $\Rightarrow \mathbf{C}(q_I) + C_{Stack} + \mathbf{C}_{Entry} <= Max\_Capacity$ **(?)** from fact 2 in state I $\Rightarrow \Sigma C_I + C_{Stack} + \mathbf{C}_{Entry} <= Max\_Capacity$ is **true** from fact 1 in state I
0	$\Sigma \mathbf{C}_0 + C_{Lengthof}$ **<=** $Max\_Capacity$**(?)** $\Rightarrow \mathbf{C}(q_0) + \mathbf{C}(s_0) + \mathbf{C}(x_0) + \mathbf{C}_{Lengthof}$ **<=** $Max\_Capacity$**(?)** $\Rightarrow \mathbf{C}(q_I) + C_{Stack} + C_{Entry} + \mathbf{C}_{Lengthof} <= Max\_Capacity$ **(?)** from fact 3 in state 0 **Since ,** $\mathbf{C}_{Lengthof} <= MAX(C_{Push}, C_{Pop}, C_{Dequeue}, C_{Enqueue}, C_{Depthof}, C_{Lengthof})$ **and** $\mathbf{C}(q_I) + \mathbf{C}_{Stack} + \mathbf{C}(Entry) + MAX((C_{Push}, C_{Pop}, C_{Dequeue}, C_{Enqueue}, C_{Depthof}, C_{Lengthof}) <= Max\_Capacity$ $\Rightarrow \mathbf{C}(q_I) + C_{Stack} + C(Entry) + \mathbf{C}_{Lengthof} <= Max\_Capacity$ is **true**
1	$\Sigma C_1 + C_{Dequeue} <= Max\_Capacity$**(?)** $\Rightarrow C_{Stack} + \lvert s_1 \rvert * C_{Entry\_Overhead} + \sum_{i=x} \mathbf{C}(x)$, where $IS\_ENTRY\_OF(s,x)$ **+** $C_{Queue} + \lvert q_1 \rvert * C_{Entry\_Overhead} + \sum_{i=x} \mathbf{C}(x)$, where $IS\_ENTRY\_OF(q,x) + \mathbf{C}(x_1) + C_{Dequeue} <= Max\_Capacity$**(?)**

	$\Rightarrow C_{stack} + C_{queue} + \|s_1\| * C_{Entry\_Overhead} + \|q_1\| * C_{Entry\_overhead} +$ $(\sum_{i=x} \mathbf{C}(x)$, where IS_ENTRY_OF($s_1$,x) + $\sum_{i=y} \mathbf{C}(y)$, where IS_ELMENT_OF(q_1,y)) + $\mathbf{C}(x0) + C_{Dequeue} <=$ Max_Capacity**(?)** from fact 2 in state 1 $(\sum_{i=x} \mathbf{C}(x)$, where IS_ENTRY_OF($s_1$,x) + $\sum_{i=y} \mathbf{C}(y)$, where IS_ELMENT_OF(q_1,y)) $\Rightarrow$ $\sum_{i=z} \mathbf{C}(z)$, where IS_ELMENT_OF(q_0,z) from fact 1 in state 1 $\Rightarrow C_{stack} + C_{queue} + \|s_1\| * C_{Entry\_overhead} + \|q_1\| * C_{Entry\_overhead} +$ $\sum_{i=z} \mathbf{C}(z)$, where IS_ELMENT_OF(q_0,z) + $\mathbf{C}_{Entry} + C_{Dequeue} <=$ Max_Capacity**(?)** $C_{Dequeue <=} MAX(C_{Push}, C_{Pop}, C_{Dequeue}, C_{Enqueue}, C_{Depthof}, C_{Lengthof})$ $\Rightarrow C_{stack} + C_{queue} + (\|s_1\| + \|q_1\|) * C_{Entry\_overhead} +$ $\sum_{i=z} \mathbf{C}(z)$, where IS_ELMENT_OF(q_0,z) + $\mathbf{C}_{Entry}$ + MAX(C_{Push}, C_{Pop}, $C_{Dequeue}$, $C_{Enqueue}$, $C_{Depthof}$, $C_{Lengthof}$) $<=$ Max_Capacity**(?)** $\Rightarrow C_{stack} + C_{queue} + (q_0) * C_{Entry\_overhead} +$ $\sum_{i=z} \mathbf{C}(z)$, where IS_ELMENT_OF(q_0,z) + $\mathbf{C}$(Entry) + MAX(C_{Push}, C_{Pop}, $C_{Dequeue}$, $C_{Enqueue}$, $C_{Depthof}$, $C_{Lengthof}$) $<=$ Max_Capacity**(?)** $\Rightarrow C_{stack} + \mathbf{C}(q_0) + \mathbf{C}$(Entry) + MAX($C_{Push}$, C_{Pop}, $C_{Dequeue}$, $C_{Enqueue}$, $C_{Depthof}$, $C_{Lengthof}$) <= Max_Capacity is **true**

2	$\Sigma C_2 + C_{Push} <= Max\_Capacity$**(?)** $\Rightarrow$ **C**(q_2) **+** **C**(s_2) + **C**(x_2) + $C_{Push} <= Max\_Capacity$**(?)** **Since,** **C**(x_2) + **C**(q_2) <= **C**(q_1) from fact1 in state 2 **and** **C**(s_2) = **C**(s_1) from fact2 in state 2 $\Rightarrow$ **C**(q_2) **+** **C**(s_2) + **C**(x_2) + $C_{Push} <=$ **C**(q_1) + **C**(s_1) + C_{Push} Since, $C_{Push <=} MAX(C_{Push}, C_{Pop}, C_{Dequeue}, C_{Enqueue}, C_{Depthof}, C_{Lengthof})$ $\Rightarrow$ **C**(q_1) + **C**(s_1) + $C_{Push <=}$ **C**(q_1) + **C**(s_1) + $MAX(C_{Push}, C_{Pop}, C_{Dequeue}, C_{Enqueue}, C_{Depthof}, C_{Lengthof})$ From state 1 storage obligation, **C**(q_1) + **C**(s_1) + $C(x_1)$ + $MAX(C_{Push}, C_{Pop}, C_{Dequeue}, C_{Enqueue}, C_{Depthof}, C_{Lengthof})$ <= Max_Capacity $\Rightarrow$ **C**(q_2) **+** **C**(s_2) + **C**(x_2) + $C_{Push} <=$ Max_Capacity is **true**

Figure 3.12 Proof Table for the Storage Obligations

3.3 Verification of Recursive Procedures

In the case of recursive procedures the run time call stack depth depends on the size of the input. In this case and in general, the capacity required by a specification may not be a constant. The capacity constraints for such case, i.e., recursive implementation, are given in Figure 3.13. Figure 3.14 presents an implementation of the Reverse operation. To validate that the implementation is correct with respect to its storage constraints, we can verify this procedure as shown in Figure 3.16, using the fact/obligation table in Figure 3.15.

```
Enhancement Reverse_Capability  for Queue_Template
   interface
     operation Reverse(alters q: Queue)
        requires capacity   =|q| *  (C_Entry + MAX(C_Length_Of , C_Dequeue , C_Enqueue))
        ensures  q = #q^R
end Reverse_Capability
```

Figure 3.13 Capacity Constraints for a Recursive Implementation of Reverse Operation

```
relization body Recursive for Reverse_Capability
        procedure Reverse ( alters q: Queue)
        decreasing |q|
        context
                variables
                        temp: Entry
        begin
                if Length_Of(q) > 0 then
                        Dequeue(q,temp)
                        Reverse(q)
                        Enqueue(q,temp);
                end if
        end Reverse
end Recursive
```

Figure 3.14 A Recursive Implementation of Queue Reverse Operation

State ment	Path condition	Facts	Obligations	Storage Obligation
I		1. $\Sigma_{i=x}$ **C**(x), where IS_ENTRY_OF(q_I,x) + $\|q_I\|$ * (**C**(Entry) + MAX(**C**$_{Length\_Of}$, **C**$_{Dequeue}$, $C_{Enqueue}$)) <= Max_Capaity		$\Sigma_{i=x}$ **C**(x), where IS_ENTRY_OF(q_i,x) + **C**(Entry) <= Max_Capacity **(?)**
	temp: Entry			
0		1. $q_0 = q_I$ 2. ΣC_I <= Max_Capacity		ΣC_0 + **C**$_{Length\_Of}$ <= Max_Capacity **(?)**
If Length_Of(q) > 0 **then**				
1	$\|q_0\| > 0$	1. temp1 = temp0 2. q1 = q0 3. ΣC_0 <= Max_Capacity	$\|q_0\| > 0$**(?)**	ΣC_1 + **C**$_{Dequeue}$ <= Max_Capacity **(?)**
Dequeue(q,temp)				
2	$\|q_0\| > 0$	1. $q_1 = x_2 * q_2$ 2. ΣC_1 <= Max_Capacity 3. $\#q_1 = temp_2 * q_2$	q_0 is decreasing**(?)**	ΣC_2 + $\|q_2\|$ * (C_{Entry} + MAX(C_{Length_Of} , $C_{Dequeue}$, $C_{Enqueue}$) <= Max_Capacity **(?)**
Reverse(q)				
3	$\|q_0\| > 0$	1. temp3 = temp2 2. ΣC_2 <= Max_Capacity 3. $q_2 = q_3^R$		ΣC_3 + **C**$_{Enqueue}$ <= Max_Capacity **(?)**
Enqueue(q,temp)				
4	$\|q_0\| > 0$	1. ΣC_3 <= Max_Capacity 2. q4 = q3*temp3		ΣC_4 <= Max_Capacity **(?)**
End if				
F				ΣC_F <= Max_Capacity **(?)**

Figure 3.15 Facts and Obligations Table for Recursive Reverse Implementation

State	Proof
I	$\Sigma_{I=x}\mathbf{C}(x)$, where IS_ENTRY_OF($q_I$,x) + **C**(Entry) <= Max_Capacity **(?)** $\Rightarrow$ **true** from the fact 1 in state I
0	ΣC_0 + $\mathbf{C}_{Length\_Of}$ <= Max_Capacity **(?)** $\Rightarrow$ **true,** since $q_0 = q_I$ and from the fact 1 in state I
1	ΣC_1 + $\mathbf{C}_{Dequeue}$ <= Max_Capacity **(?)** $\Rightarrow \mathbf{C}(q_1) + \mathbf{C}(temp_1) + C_{Dequeue}$ <= Max_Capacity **(?)** $\Rightarrow$ **true,** since $q_1 = q_0$, temp1 = temp0 and from the fact 1 in state I
2	$\Sigma C_2 + \|q_2\| * (C_{Entry} + MAX(C_{Length\_Of}, C_{Dequeue}, C_{Enqueue}))$ <= Max_Capacity **(?)** $\Rightarrow \mathbf{C}(temp_2) + \mathbf{C}(q_2) + \|q_2\| * (C_{Entry} + MAX(C_{Length\_Of}, C_{Dequeue}, C_{Enqueue}))$ <= Max_Capacity **(?)** **Since** $q_1 = temp_2 * q_2 \Rightarrow \mathbf{C}(temp_2) + \mathbf{C}(q_2) <= \mathbf{C}(q_1)$ **and** $\|q_2\| < \|q_1\| \Rightarrow \mathbf{C}(temp_2) + \mathbf{C}(q_2) + \|q_2\| * (C_{Entry} + MAX(C_{Length\_Of}, C_{Dequeue}, C_{Enqueue})) <= \mathbf{C}(q_1) + \|q_1\| * (C_{Entry} +$

	$MAX(C_{Length\_Of}, C_{Dequeue}, C_{Enqueue})$ **and** $q_1 = q_0 = q_I$ **and** $\Sigma_{i=x}$ $\mathbf{C}(x)$, where IS_ENTRY_OF(q_I,x) + $\lvert q_I \rvert$ * (C_{Entry} + $MAX(\mathbf{C}_{Length\_Of}, \mathbf{C}_{Dequeue}, C_{Enqueue})$) <= Max_Capaity $\Rightarrow \Sigma C_2 + \lvert q_2 \rvert * (C_{Entry} + MAX(C_{Length\_Of}, C_{Dequeue}, C_{Enqueue}))$ <= Max_Capacity is **true**
3	$\Sigma C_3 + \mathbf{C}_{Enqueue}$ <= Max_Capacity **(?)** $\Rightarrow \mathbf{C}(q3) + \mathbf{C}(temp3) + \mathbf{C}_{Enqueue}$ <= Max_Capacity **(?)** $\mathbf{C}(q_3) = \mathbf{C}(q_2)$, since $q_3 = q_2^R$ and temp3 = temp2 $\Rightarrow \mathbf{C}(q2) + \mathbf{C}(temp2) + \mathbf{C}_{Enqueue}$ <= Max_Capacity **(?)** **Since** $\mathbf{C}(q2) + \mathbf{C}(temp2) + \mathbf{C}_{Enqueue} <= \mathbf{C}(temp_2) + \mathbf{C}(q_2) + \lvert q_2 \rvert * (\mathbf{C}_{Entry} + MAX(C_{Length\_Of}, C_{Dequeue}, C_{Enqueue}))$ $\Rightarrow \mathbf{C}(q3) + \mathbf{C}(temp3) + \mathbf{C}_{Enqueue}$ <= Max_Capacity is **true**
4	ΣC_4 <= Max_Capacity **(?)** $\Rightarrow \mathbf{C}(q_4) + \mathbf{C}(temp_4)$ <= Max_Capacity **(?)** $\mathbf{C}(q_4) + \mathbf{C}(temp_4) = \mathbf{C}(q_1) + C_{Entry}$ $\Rightarrow \Sigma C_1$ <= Max_Capacity is **true**

Figure 3.16 Proof Table for Recursive Implementation of Reverse

3.4 Storage Specification of Layered Components

In this section storage specification of built-in types and user-defined types are discussed.

3.4.1 Storage Specification of Built-In types

The storage specification and verification question discussed in this chapter is based on the availability of storage requirements for every object type. For built-in types, such as integer, float, or pointers the capacity requirements are fixed and they are assumed to be known statically. For built-in composite types, such as records, the capacity requirements include some overhead plus capacity for members of the type. The capacity used by a static array s is given below:

type Static_Array **is modeled by function from** integer **to** Entry
exemplar s
uses capacity $C_{sa} + \Sigma_{i=s.l\_bd}^{s.u\_bd} \mathbf{C}(S(i)) + (s.u\_bd - s.l\_bd + 1) * C_{Entry\_Overhead}$

Here s.l_bd and s.u_bd represent the lower and upper bounds of the array and Entry denotes the type of the elements in the array. The term $(s.u\_bd - s.l\_bd + 1)*C_{Entry\_overhead}$ specifies the overhead required to hold the pointers. The constant C_{sa} represents the amount of overhead storage required by the array.

The capacity used by a 2-field record type with two elements of types Entry 1 and Entry 2 is given below:

type Record **is modeled by** (f1: Entry1, f2:Entry2)
exemplar r
uses capacity $C_{r2} + \mathbf{C}(r.f2) + \mathbf{C}(r.f2)$

In this specification the storage requirement of a record type is given as the sum of a constant amount of storage, i.e. C_{r2}, the storage required by field f1 and the storage required by field f2. For example, if the two fields are of type integer, the storage required by the record type would be $C_{r2} + C_{Integer} + C_{Integer}$ The complete functional and storage specification of these and other built-in types are given in Appendix B.

3.4.2 Relationship between Base Types and ADTs

In any programming system, any user-defined type can be viewed as an aggregation of base types in that system. Once the capacity requirements of base types are given, the deriving requirements for ADTs implemented directly using base type is straightforward. Figure 3.19 contains a simple example. It shows the capacity usage for a typical "linked list" implementation (given in Appendix B) of the List ADT. Figure 3.17 presents a stack of floats implemented based on a List concept. The total storage taken up by the List of integers in this figure can be calculated as $C_{pointer} + 2* C_{pointer} + \sum_{x:Entry} (C_{pointer} + \mathbf{C}(x))$ where

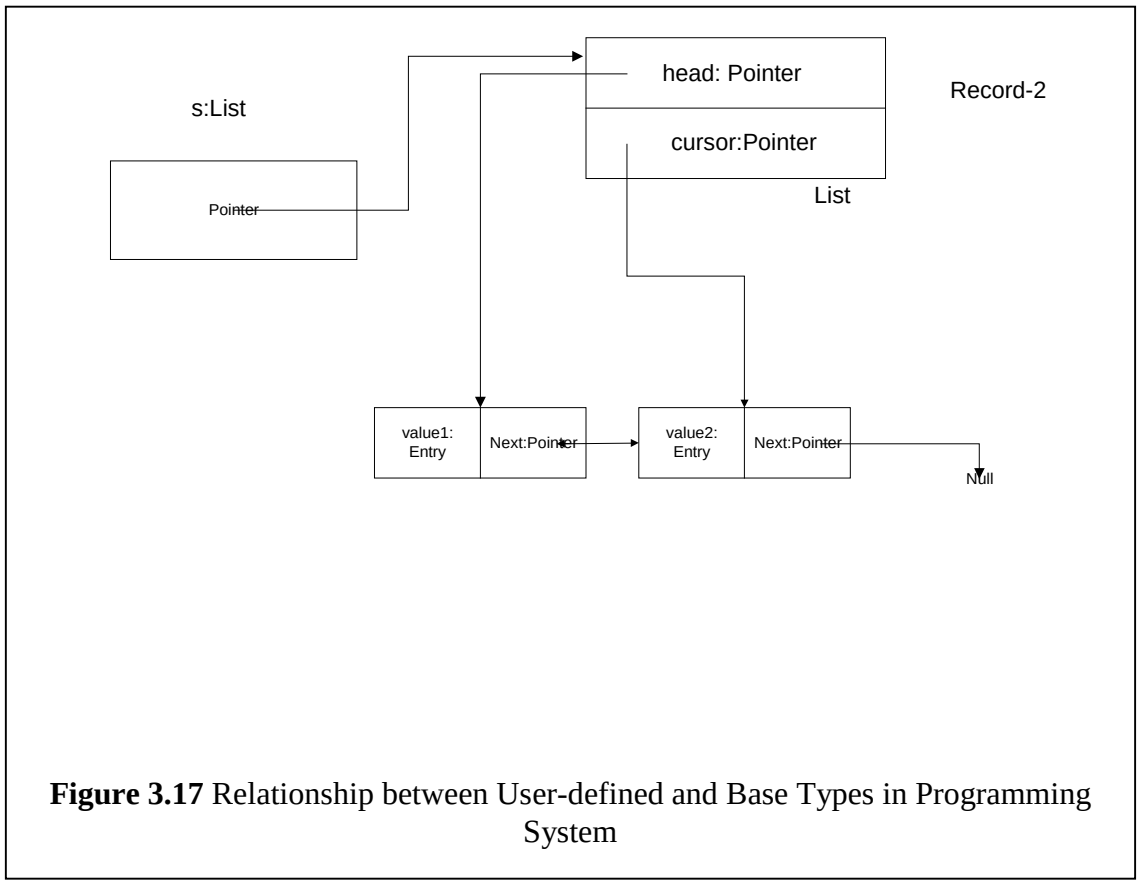


Figure 3.17 Relationship between User-defined and Base Types in Programming System

IS_ENTRY_OF(s,x) holds. If Entry is type Integer, then **C**(x) denotes $C_{integer}$,, a representation of the storage required by x because, all integers take the same storage. However, for arbitrary type Entry, **C**(x) defines the storage required by the x at that point of the program. For the specific example in Figure 3.17, if integer and pointer sizes are 2 bytes each, then the list size can be calculated to be (2 + 2 * 2) + (2 + 2) + (2 + 2) = 14 bytes.

3.4.3 Storage Specification of Layered Objects

We conclude this section by explaining how capacity requirements can be calculated for objects layered using other objects. Development of such layered objects is a common practice in software development and, thus it is essential for the proposed framework to be useful in the case of layered components. To illustrate these issues we present an example of Stack implemented using a List. Figure 3.18 presents a stack realization based on List template. In figure 3.20, the type Stack is represented as a List. To enable this a facility for a List of type Entry is declared in the local context. Along with the representation, a module level **correspondence** assertion is given. To show that the realization is behaviorally correct, it is essential to show that the correspondence is "well formed" (i.e., there is an abstract value for each representation value allowed by the exemplar) and that the code for each procedure meets its specification in Stack_template [23]. Storage verification raises additional two obligations:

- The storage for the representation is within the storage bounds specified for the object in the concept, and

- The storage used by each procedure requires no more capacity than specified in the requires capacity clause for that operation.

The first obligation leads to the following assertion in the present case:

for all s.rep: **math** [List],
s.rep.left = **empty_string**
implies C(s.rep) <= C_{stack} + |s|*$C_{Entry\_overhead}$ + $\sum_{x:Entry}$**C**(x) **where** IS_ENTRY_OF(s,x)

Proving that this assertion true is rather trivial. In the obligation, **C**(s.rep) can be replaced by **C**(List) which is equivalent to C_{List} + (|s.left| + |s.right|) * $C_{Entry\_Overhead}$ + $\sum_{x:Entry}$**C**(x) **where** IS_ENTRY_OF((sleft * s.right),x). From the conventions and correspondence, it is straight forward to prove that s.left * s.right = s of Stack. The constants C_{stack} and C_{list} are equal, as both represent the storage taken by a pointer type.

The second set of obligations essentially amount to showing LF.C_{insert} <= C_{push} , LF.C_{Remove} <= C_{pop} and LF.$C_{Right\_Length\_Off}$ <= C_{Depth_Of} The realization of Push operation internally makes a call to the insert operation of List. This implies that the transient storage required for the push operation is the sum of the storage required for making the call to operation Insert and the runtime storage requirements of the operation Push. Similarly, the other two assertions can be proved true.

```
realization body List_Based for Stack_Template
    context
        global context
                concept List_Template
                realization Pointer_Based for List_Template
        local context
                facility LF is List_Template(Entry)
                        realized by Pointer_Based
    interface
        type stack is represented by List
                conventions s.rep.left = empty_string
                correspondence s=s.rep.right

    procedure Push (
                        alters s Stack
                        consumes x Entry
                        )
        begin
                LF.Insert(s.rep,x)
        end Push
    procedure Pop(
                        alters s Stack
                        produces x Entry
                        )
        begin
                LF.Remove (s.rep,x)
        end Pop
    function procedure  Depth_Of(
                                        preserves s Stack
                                        ): Integer
        begin
                return(LF.Right_Length_Of(s.rep))
        end Depth_Of
end List_Based
```

Figure 3.18 Realization body for List-based Stack Template

3.5 Discussion

We conclude this chapter with a discussion of the soundness and completeness of the proposed storage specification and verification system. A formal system has to be *sound* and *relatively complete* to be useful. Soundness of a verification system can be defined as

~∃ (Assertive Code)/VerificationSystem(Assertive Code) = **true** Λ (Assertive Code) ≠ **correct,**

where (Assertive Code) is a program with specification.

In other words, a verification system is not sound if there exists an assertive code which is false, and the verification system proves it to be true. As a characteristic of formal systems, completeness is the dual of soundness[40]. A complete system does not have too few formal theorems; a sound system does not have too many. Because of the inherent incompleteness of number theory, a notion of 'relative completeness' is adopted for the formal systems of program verification. In other words, the formal system is complete with respect to the truth of the purely mathematical formulas, i.e., it is complete relative to the incompleteness of mathematical theories. The incompleteness is not due to the flaws in the proof rules regarding the computation, but to the incompleteness of mathematics itself. Relative completeness of a verification system can be defines as

∀(Assertive Code)/ (Assertive Code) = **correct** ⇒ Verification System(Assertive Code) = **true**

The *natural reasoning* verification process is a system of purely syntactic manipulation which transforms a formula, consisting of a program with a specification, into a formula of classical mathematics. This transformation may occur in many steps, each step justified by a rule of the formal system. The idea of this system is that if the final purely mathematical formula is true, then the correctness conjecture for the original program and specification is also true. The transformation rules, also called proof rules, are specified and proved to be sound and relatively complete by Heym[24]. With the proposed additions for storage specification we have to prove that the verification system is still sound and relatively complete.

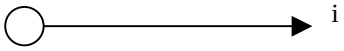
For verification that a software system does not violate its storage specification, it is essential to show that at every state of the system the capacity used by all active objects at that state plus the capacity required for the next step is within the maximum storage capacity of the physical system. To avoid the state explosion and to keep the verification system modular, storage specification of objects and operations are used; this makes it possible to ignore transient objects and state transitions that arise when an operation is called. Before a procedure call P in the state i, the storage verification obligation simplifies to showing $\sum C_i + C_p <=$ Max_Capacity, where C_p denotes the transient capacity required by P. Before, a sequence of calls in state i, P;Q the obligations $\sum C_i + C_p <=$ Max_Capacity and $\sum C_i + C_q <=$ Max_Capacity need to be shown to be true. For control statements, similar obligations arise. Before an **if** statement

if B **then** P **else** Q **endif**

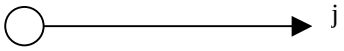
the obligation ((B **implies** $\sum C + C_p <=$ Max_Capacity)**and** (~ B **implies** $\sum C + C_p <=$ Max_Capacity) needs to be proven. For a loop statement, beginning with state i

maintaining Invariant

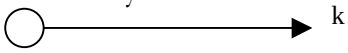
decreasing M



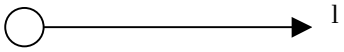
while B **loop**



body



end loop



the obligation (($\sum C_i$ <= Max_Capacity) **and** ($\sum C_j$ <= Max_Capacity) **and** ($\sum C_k$ <= Max_Capacity) **and** ($\sum C_l$ <= Max_Capacity)) needs to be proven. Soundness of the rules described above directly follows from the discussion in [24]. While soundness is essential for all proof systems, relative completeness is important for it implies tightness on the storage bounds. Relative completeness with respect to performance implies the ability of the system to prove the tightest bounds possible. The system proposed in this chapter cannot satisfy this

tightness requirement because the storage specification is done at the abstract level for objects. If the specification has to be generic enough not to preclude any interesting implementations of a concept, the bounds on storage cannot be based on precise values. For example, in the Reverse example, the constants in the *requires capacity* clause $C_1(|q|) + C_2$, have to be big enough to cover both iterative and recursive implementations. Similarly, the storage requirements can be specified at various levels of precision. It is apparent that the abstract specification mechanism proposed may not always provide the precision required for true mission-critical systems. In such systems, precise specification based on intermediate levels of abstraction that reflect aspects of implementations is necessary. This is the central them of Chapter 4.

Chapter 4

INTERMEDIATE LEVELS OF ABSTRACTION FOR STORAGE CONSTRAINT SPECIFICATION

The storage specification examples discussed in this dissertation have considerations on specification of storage constraints at the same level of abstraction as the behavioral specification. However, specification of storage bounds along with the functional specification, without any implementation bias, in general may not provide the necessary preciseness required by many practical systems. The focus of this chapter is to explain this problem and provide a framework for expressing tight bounds on storage constraints.

For a given behavioral specification of a concept, in general there are alternative implementations that differ in their performance characteristics. Given a functional specification, it is not possible to come up with one implementation that satisfies all performance requirements. Different implementations provide different tradeoffs in their performance characteristics. Tradeoffs between the execution times, or average and worst case times, or execution time and storage are among reasons for multiple plug-compatible implementations of a concept. Availability of multiple implementations for the same concept provides the clients performance flexibility. Clients can choose an implementation that has the desirable performance characteristics. Replacing one implementation with another one will not require the behavioral re-verification of the client program, but only need re-verification of client performance.

User need for implementations satisfying different storage constraints can arise either because of the storage limitations of the underlying systems or because of other constraints, such as efficiency or predictability. To pick suitable implementations, users will need to compare the storage requirements of different implementations of a concept. But what level of precision is needed in storage specification for clients to be able to make this choice? This level of precision is the topic of this chapter. In general, it is possible to specify upper bounds on storage requirements for an alternative implementation of a functional specification at the

same level of abstraction as functionality. But, to express tight specification of bounds on storage, implementation aspects have to be considered in specification. In particular, it may not be possible to express the constraints using the same mathematical models of objects that are used in the functional specification. However, expressing complete implementation details in storage specification may not be appropriate or acceptable. This dilemma in expressing storage requirements precisely, together with the need for tight specification of other performance factors such as time bounds [25], raise the need for specification of constraints at intermediate levels of abstraction. The rest of this chapter is organized as follows: Section 4.1 presents the need for intermediate levels of abstraction using examples. Section 4.2 presents a framework for introducing intermediate abstraction models. Section 4.3 contains a discussion on the relative completeness of the framework with respect to storage constraint verification.

4.1 An Example

This section illustrates the issues in tight specification of storage bounds with a simple, albeit artificial, example. Figure 4.1 presents the specification of a concept that provides an ADT to find minimum and maximum values of a given set of type Entry. This concept is parameterized by type Entry. It is also parameterized by a mathematical definition R to compare two entries. At the time of instantiation of this concept, in addition to the actual type for Entry, the client should also supply a math definition that is a total pre-ordering on type Entry. The ADT MinMax is mathematically modeled by a tuple <min,max,valid>. *Min* abd *max* represent the minimum and maximum values of type Entry for a given set of inputs. *Valid* is a flag that indicates whether there are any valid minimum and maximum values. Initially, the value of a *valid* flag is false, because no values have been yet inserted. The first call to the Insert operation ensures that the *valid* flag changes to *true*. When all the values have been inserted, the minimum and maximum values can be found by calling operations Get_Min and Get_Max. To develop any implementation of MinMax_Template, a program specification to compare two entries is needed. This operation will be called to compare Min and Max entry values in the implementation. This representation for a particular Entry type must be supplied at instantiation time. Different implementations of MinMax_Template may have different storage and other performance requirements. One implementation of MinMax_Template may compute Min and Max entry values as the entries are inserted. This implementation would have no need to store all inserted Entry values. An

alternative implementation, one that is layered using Prioritizer_Template would store all input Entry values and compute Min and Max values only when needed. Clearly the storage used by these implementations are widely different. To express this non-functional requirements for a particular class of implementations, implementation-specific information is provided in a **realization header**. Figure 4.2 contains a header for a straight forward "frugal" implementation of MinMax_Template that does not store all inserted Entry values.

```
concept MinMax_Template
        context
            parametric context
                type Entry
                math operation R(x,y : Entry) : boolean
                    implicit definition
                                    for all x,y: Entry
                                     (R(x,y) and R(y,z) ⇒ R(x,z)) and
                                     (R(x,y) or R(y,x)))
                interface
                        type MinMax is modeled by(      min:Entry,
                                                        max:Entry,
                                                        valid:Boolean)
                        exemplar m
                        initialization
                            ensures m.valid = False
        opeartion Insert(   alters m: MinMax,
                            consumes x: Entry)
            ensures     if not (m.valid)
                            m.min = x and m.max = x and valid = true
                        else
                            if R(x,m.min) then m.min = x and
                            if R(m.max,x) then m.max = x
        opeartion Get_Min(   preserves m:MinMax,
                             produces min: Entry)
            reuires     m.valid
            ensures     min = m.min
        opeartion Get_Max(  preserves m:MinMax,
                            produces max: Entry)
            reuires     m.valid
            ensures     min = m.max
        opeartion Is_Valid(    preserves m:MinMax): Boolean
            ensures     isMinValid = m.valid
 end MinMax_Template
```

Figure 4. 2 MinMax_Template Specification

realization header Frugal **for** MinMax_Template
context
parametric context
operation Are_In_Order (**preserves** x: Entry
preserves y: Entry
): Boolean
ensures Min **iff** (MIN(x,y))
local context
$C_{Insert} = 2*C_{Reference}$
$C_{Get\_Min} = 2*C_{Reference}$
$C_{Get\_Max} = 2*C_{Reference}$
$C_{Is\_Valid} = 2*C_{Reference}$
interface
type *MinMax* ***is modeled by*** *(min:Entry,*
max:Entry,
valid:Boolean)
exemplar *m*
uses capacity C_{MinMax} + **C**(m.min) + **C**(m.max) + $C_{boolean}$
initialization
requires capacity C_{MinMax} + 2* $\mathbf{C}_{Entry}$ + $C_{boolean}$
opeartion Insert (**alters** m: MinMax,
consumes x: Entry
)
requires capacity C_{Insert}
opeartion Get_Min(**preserves** m:MinMax,
produces min: Entry
)
requires capacity C_{Get_Min}
opeartion Get_Max(**preserves** m:MinMax,
produces max: Entry
)
requires capacity C_{Get_Max}
opeartion Is_Valid(**preserves** m:MinMax): Boolean
requires capacity C_{Is_Valid}

end Frugal

Figure 4. 2 Realization header for a Frugal implementation

The storage specification for a frugal implementation of MinMax_Template is given in the interface section in figure 4.2. Here the storage required by an object of the type is given as C_{Min_Max} + **C**(Min) + **C**(Max) + $C_{boolean}$. The corresponding implementation uses a three-

element record with two fields of type Entry, to hold the minimum and maximum values, and a Boolean field. On every call to Insert, the implementation updates the Min and Max values. The storage required for the MinMax object by this implementation is exactly equivalent to that of the storage required by the 3-field record. The realization header also shows that to write such an implementation, an operation to 'compare' two objects of type Entry has to be provided as specified in the parametric context. This is because only the client that supplies the type Entry at instantiation time can supply the ordering operation too. The syntax and semantics of the operation supplied by the client have to match the specification given in the parametric context of the header. In the interface section of this realization header, the storage constraints of each operation are specified. Specification of storage constraints in the realization header allows separating implementation-dependent aspects from the behavioral specification.

Figure 4.3 contains an alternate "Profligate" realization header. Implementations based on this header retain all the input values that are inserted. An implementation layered using Prioritizer_Template falls under this class. Such an implementation is much easier to write, because the code for insert merely calls the Prioritizer Insert operation. Get_Min and Get_Max are based on the Prioritizer Remove operation. However, this implementation is inferior to the *Frugal* implementation in all respects, both in storage utilization and in time to compute minimum and maximum values. The mathematical model of the MinMax concept given in Figure 4.1 is not expressive enough to capture the storage requirements of this class of implementations. The realization header given in Figure 4.3 introduces an intermediate model in the interface section to conceptualize the new behavior. But for verification purposes, there must be a correspondence between the behavioral model and the implementation-oriented intermediate model. The correspondence assertion in the interface section relates a value in the intermediate model to a value in the model used in the concept. For the Profligate realization the intermediate model chosen is a set of type Entry. The set represents all the values that are inserted. In the correspondence section, the Min value of the concept is equated to the value of the MIN operation on the set which is defined in the local context. Similarly for the value of Max. The Valid value is true when the size of the set is greater than zero.

The realization header also includes the specification of operations specified in the interface of the concept. The requires and ensures clauses of these operations are based on the intermediate model. For example, the insert operation's ensures class specifies that the insert item would be added to the set. This would increase the size of the set by one, which makes the value of valid *true*. If there is only one element in the set, both the MIN and MAX values are same. The relationship between the elements of the intermediate model and the elements of the conceptual model are defined by the correspondence clause.

We conclude this section by presenting a more complex, but more realistic example of Minimum Spanning Forest (MSF) abstraction[19]. The specification of the MSF concept is reproduced from [19] in Figure 4.4. An MSF data abstraction defines a type Spanning_Forest_Machine_State (an instance of this type is referred as a *machine* henceforth) and operations that support the "two-phase" client usage: put a machine into insertion phase; then insert into it, one at a time, the edges of the graph for which an MSF is to be found; then change the machine to extraction phase; and finally extract from it, one at a time, the edges of one of the (possibly many) MSFs of that graph. The interface of this concept provides the following operations, where m represents the MSF machine:

- Change_To_Insertion_Phase(m): Prepare m for calls to the Insert operations. This operation requires that m be in the extraction phase at the time of the call.

- Insert(m,v1,v2,w): Insert edge (v1,v2,w) into m. This operation requires that m be in the insertion phase at the time of the call.

- Change_To_Extraction_Phase(m): prepare m for calls to the Extract operation. This operation requires that m be in the insertion phase at the time of the call.

- Extract(m,v1,v2,w): extract a (remaining) MSF edge from m, returning its vertices and weight in (v1,v2,w). This operation requires that m be in the extraction phase at the time of the call.

- Size(m): Return the number of edges in m.

- Is_In_Insertion_Phase(m): Test whether m is in insertion phase.

Based on the mathematical model of the specification there are two possible implementations. One of them is based on *Greedy* algorithms in which only the MSF is retained during the insertion phase. All the other edges that are not part of the MSF will be discarded. This implementation conserves capacity. The second implementation can retain the entire graph. This implementation is not capacity-conserving. The behavioral specification of the MSF does not pin down which approach to be taken. The abstract graph value m.edges in the insertion phase now may be any graph that shares an MSF with the input graph, and the exact graph is dictated by the implementations. The SHARE_AN_MSF(g1,g2) predicate makes it possible not specifically say to store just MSF or the entire graph. Including storage specification along with the behavioral specification is not appropriate in this case too, as different implementations need different storage. Also, depending on the implementation scheme, the model represented in the concept may not be suitable to conceptualize the storage required, and hence an intermediate level abstract model is required to precisely define the storage requirement for that implementations.

realization header Profligate **for** MinMax_Template

context

parametric context

operation Are_In_Order (**preserves** x: Entry
preserves y: Entry
): Boolean

ensures Min **iff** (MIN(x,y))

local context

math operation MIN (s : set of Entry): Entry

ristriction |s| > 0

explicit definition

if(|s| =1) MIN(s) = (x: Entry∈ s)

else MIN(s) = (x :Entry ∈ s **and** ∀y:Entry, y≠x **and**
y ∈ s **and** MIN(x,y))

math operation Max (s : set of Entry): Entry

restriction |s| > 0

explicit definition

if(|s| =1) MAX(s)= (x: Entry∈ s)

else Max(s) = (x :Entry ∈ s **and** ∀y:Entry, y≠x **and**
y ∈ s **and** MAX(x,y))

$C_{Insert} = 2*C_{Reference}$

$C_{Get\_Min} = 2*C_{Reference}$

$C_{Get\_Max} = 2*C_{Reference}$

$C_{Is\_Valid} = 2*C_{Reference}$

interface

type MinMax **is modeled by finite set of** Entry

exemplar s

initialization

ensures s = empty_set

uses capacity $C_{MinMax} + \sum_{x:Entry} \mathbf{C}(x)$, **where** x∈ s

correspondence **if**(|s| =0) m.valid = **false**
else m.valid = **true and**
m.valid ⇒m.min = MIN(s) **and**
m.max = MAX(s)

(continued on the next page)

```
    opeartion Insert(  alters m: MinMax,
                       consumes x: Entry
                       )
        requires capacity C_insert
        ensures  m = #m union {#x}
    opeartion Get_Min(   preserves m:MinMax,
                         produces min: Entry
                         )

        requires capacity C_Get_Min
    opeartion Get_Max(  preserves m:MinMax,
                        produces max: Entry
                        )
        reuires |m| >0
        requires capacity C_Get_Max
    opeartion Is_Valid(   preserves m:MinMax ): Boolean
        requires capacity C_Is_Valid
        ensures isValid = if(|m|) true else false

end Profligate
```

Figure 4. 3 Realization Header for Profligate Implementation of MinMax

```
concept Spanning_Forest_Machine_Template
    context
        global context
            facility Standard_Boolean_Facility
            facility Standard_Integer_Facility
        parametric context
            constant max_vertex: Integer
                restriction max_vertex > 0
            constant max_edges: Integer
                restriction max_edges > 0
        local context
            math subtype EDGE is(  v1: integer
                                   v2: integer
                                   w: integer
                                   )
        exemplar e
            constraint      1 <= e.v1 <= max_vertex and
                            1 <= e.v2 <= max_vertex and
                            e.w > 0
        math subtype  GRAPH is finite set of EDGE
        math operation IS_MSF(      msf: GRAPH
                                    g: GRAPH
                                    ): boolean
                definition (* true iff msf is an MSF of g *)
        math operation SHARE_AN_MSF( g1: GRAPH
                                     g2: GRAPH
                                     ): boolean
            definition ∃ msf: GRAPH
                        (IS_MSF(msf,g1) and
                        IS_MSF(msf,g2))
        interface
            type Spanning_Forest_Machine_State is modeled by(   edges: GRAPH
                                                insertion_phase: boolean
                                                )
            exemplar m
            uses capcity   C_SFM + ∑ C(x), where x: GRAPH ∈ m.edges + C(boolean)
            initialization
                requires capacity C_SFM  + C(boolean)
                ensures     m.edges = empty_set and
                            m.insertion_phase
```

(Continued on the next page)

```
operation Change_To_Insertion_Phase(
                                    alters m: Spanning_Forest_Machine_State
                                    )
        requires    not m.insertion_phase
        ensures     m.edges = empty_set and
            m.insertion_phase

operation  Insert( alters m:  Spanning_Forest_Machine_State
                   consumes v1: Integer
                   consumes v2: Integer
                   consumes w: integer
                   )
    requires    |m.edges| < max_edges and
                m.insertion_phase and
                1 <= v1 <= max_vertex and
                1 <= v2 <= max_vertex and
                w > 0
    ensures     SHARE_AN_MSF (m.edges,
            #m.edges union {(#v1,#v2,#w)}) and
            m.insertion_phase
operation Change_To-Extraction_Phase(
                              alters m: Spanning_ Forest_Machine_State
                              )
    requires    m.insertion_phase
    ensures     IS_MSF(m.edges,#m.edges) and
                not m.insertion_phase
operation Extract(      alters m: Spanning_ Forest_Machine_State
                        produces v1: Integer
                        produces v2: Integer
                        produces w: Integer
                        )
    requires    m.edges ≠ empty_set
    ensures     (v1,v2,w) is in #m.edges and
                m.edges = #m.edges – {(v1,v2,w)} and
                m.insertion_phase = #m.insertion_phase
```

(Continued on the next page)

```
        operation Size(    preserves m: Spanning_Forest_Machine_State
                           ): Integer
            ensures Size = |m.edges|
        operation Is_In_Insertion_Phase(
                                   preserves m: Spanning_ Forest_Machine_State
                                   ): Boolean
            ensures Is_In_Insertion_Phase = m.isnertion_phase
end Spanning_Forest_Machine_Template
```

Figure 4. 4 Minimum Spanning Forest Concept Specification

4.2 A Framework

With the addition of the intermediate levels of specification, component development can be

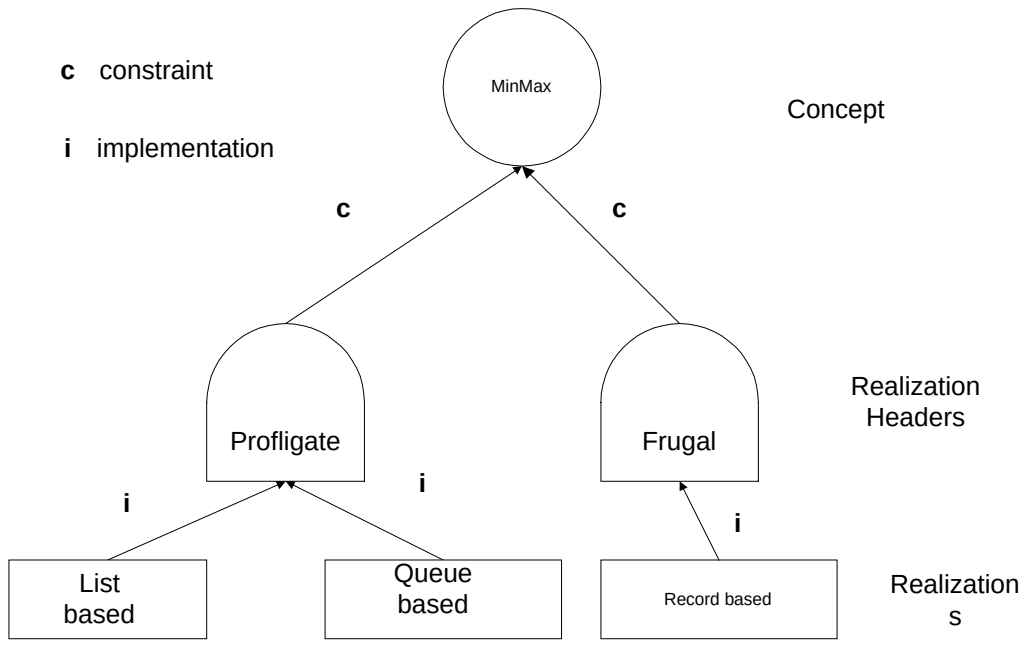


Figure 4. 5 A Frame work for interconnecting concepts, realization headers, and Realizations

represented as shown in Figure 4.5. In this figure circles denote concepts, tombstones denote realization headers, and rectangles denote realizations. An arrow from a realization header to a

concept denotes that the realization header is compatible with the concept. In the figure, both *Frugal* and *Profligate* realization headers-based implementations satisfy the MinMax concept. An arrow from a rectangle to a tombstone represents that the implementation satisfies the constraints set by that realization header. The introduction of realization headers between concepts and implementations provides an additional level of flexibility in software composition. Before a client can use a module, its concept has to be instantiated with the

```
facility Fac is A_Concept(<arguments>)
for Realization_Header_1(<arguments>)
realized by Realization_That_Satisfies_A_Concept_and_Header_1
```

Figure 4. 6 Instantiation with a Realization Header

appropriate realization header with arguments that correspond to the parameters listed in the parametric context of the concept and the realization header. The verification of the client code depends only on the concepts and realization headers used by the client. For actual execution, the client must also pick a realization that has been proven to satisfy the chosen concept and realization header. A facility instantiation can be done as shown in figure 4.6. Once the facility is declared the client can declare variables of types provided by the facility and call operations from the facility to manipulate the variables.

4.3 Discussion

The verification techniques have to reflect the introduction of the intermediate abstractions to accommodate storage constraints. The concepts, headers, and concept-header instances (facility declarations) used by the code have to be added to the *verification context*. Before a header can be included in the verification context, the corresponding concept must already exist in that context. Ensuring the syntactic compatibility of the elements in the header with those in the concept has to be done by a compiler, where as ensuring the semantic compatibility of the mathematical assertions and definitions in the context of the header is the job of the verification system. It is this semantic verification that is crucial for the verification of intermediate models. The realization header, when it introduces an intermediate model, should also include specifications of operations based on the intermediate model. To prove that the realization header does not violate the functionality prescribed by the concept, the

correspondence clause in the realization header has to be utilized. Using this it can be ensured that there is a concept model value corresponding to each intermediate model value. It is also needed to prove that the initialization ensures clause of the provided type, based on the intermediate model, is compatible with that of the conceptual model.

Also, for each operation provided, the requries clause in the header is not any stronger than that in the concept and the ensures clause in the header is not any weaker. When an operation is passed as a parameter to the realization header, the pre-, post- and storage conditions of the actual operation must be consistent with the conditions specified in the header. This can be expressed formally as follows:

Concept_Operation_Req(s,x) **and**
correspondence(s,intermediate_s) ⇒ Intermediate_Operation_Req(s→intermedaite_s,x)
and
Intermediate_Operation_Ens(s→inter_s,#s→#intermedaite_s) **and**
correspondence(s,intermediate_s) ⇒ Concept_Operation_Ens(s,#s,x)

The first two terms of the above predicate state that the requires clause of an operation in the concept, with given correspondence relation in the realization header, must be valid when the concept math model is replaced by the intermediate math model. Similarly, the other two terms in the predicate specifiy that the ensures clause of an operation in the intermediate model must be valid when substituted by the concept math model.

Based on this formalization it can be shown that the MinMax intermediate model is valid for the Get_Min operation. The requires clause of the Get_Min opertaion is m.valid = *true*. From the correspondence clause we know that the cardinality of the set is greater than zero, which in fact is the requires clause for the operation in the intermediate model.

In the case of procedure call rule verification, when the realization header introduces requires and ensures clauses, verification of storage correctness will need to make use of the pre- and post-conditions of the called operation found in the header. Without intermediate levels of abstraction it is not possible to specify tight bounds. This is the problem in trying to specify both functionality and storage-related aspects at the same level of abstraction. Intermediate levels of abstraction address the modularity and relative completeness questions. In other

words, it can be stated that there is always an appropriate model for expressing bounds on storage as tight in as desired.

Chapter 5

HIERARCHICAL STORAGE MANAGER

Dynamic storage allocation is one of the key components of any programming system. In general, dynamic storage allocators deal with the 'heap' storage where a program can request a block of memory to store a program object and later free that block at any time. Both user-controlled and garbage collection storage management mechanisms have to support this basic task, while they may differ only in the policy on who's responsibility it is to take up the task of de-allocation. At the lower levels of storage management, the allocation and de-allocation strategies have to deal with the same problems of efficiency, predictability, flexibility, and storage utilization for both of these schemes. In this chapter we propose a formally-specified storage management mechanism, called Hierarchical Storage Manager, that is predictable, efficient and allows effective storage utilization. Every object, both built-in and user-defined claims and releases storage through calls to this concept. Section 5.1 presents a taxonomy of allocators. Section 5.2 presents the key concepts of the Hierarchical Storage Manager. Section 5.3 contains the formal specification of the storage manager along with its explanation. And section 5.5 contains a discussion about this storage manager.

5.1 Taxonomy of Dynamic Storage Allocators

Allocators can be categorized based on the mechanisms they employ for managing storage. The most important mechanisms are Sequential Fits, Segregated Free Lists, Buddy Systems, Indexed Fits, and Bitmapped Fits[17]. Each one of these mechanisms is briefly discussed below:

Sequential Fits

Many of the alllocator algorithms are based on a single linear list of all free blocks of memory, using Knuth's *boundary tag* technique. There are at least two varieties of this mechanism: 1. Best Fit, 2. First Fit, and 3. Next Fit.

A best fit sequential allocator searches the free list to find the smallest free block large enough to satisfy a request. The basic strategy in this is to minimize the amount of wasted space by ensuring that fragments are as small as possible. This strategy may not always work; if the fits are not perfect most of each block will be used, but a reminder of it will be quite small and perhaps becomes unusable. The best fit search is exhaustive, although it may stop when a perfect fit is found. This exhaustive search means that a sequential best fit search does not scale well to large heaps with many free blocks. In general, the best fit strategy exhibits good memory usage. The worst-case performance of best fit is poor, with its memory usage proportional to the product of the amount of allocated data and the ratio between the largest and smallest object size.

In the first fit strategy, searching begins at the front and continues a free block large enough to satisfy the request is found. If the block is larger than necessary, it is split and the remainder is put on the free list. A problem with sequential first fit is that the larger blocks near the beginning of the list tend to be split first, and the remaining fragments result in having a lot of small blocks near the beginning of the list. These small free blocks accumulate, and the search must go past them each time a larger block is requested. There are several alternate sophisticated implementation policies to improve the performance of the first fit mechanism, one ordering the blocks after the split, such as address ordering etc. A first fit policy may tend over time toward behaving rather like best fit, because blocks near the front of the list are split preferentially. This may result, in turn, in a roughly size-sorted list.

The next fit strategy is a common 'optimization' of the first fit strategy with a *roving pointer*. The pointer records the position where the last search was satisfied, and the next search begins from there. Successive searches cycle through the free list, so that searches do not always begin in the same place and result in an accumulation of splinters. The usual rationale for this is to decrease average search times when using a linear list, but this implementation technique has major effects on the policy for storage reuse. Since the roving pointer cycles through memory regularly, objects from different phases of program execution may become intersperse in memory. This may affect fragmentation if objects from different phases have different expected lifetimes.

The classic linear-list implementations may not scale well to large heaps, in terms of time costs; as the number of free blocks grows, the time to search the list may become unacceptable. This shortcoming has led to a host of other strategies discussed in the following subsections.

Segregated Free Lists

Under this strategy allocators use an array of free lists, where each list holds free blocks of a particular size. When a block of memory is freed, it is simply pushed onto the free list for that size. When a request is serviced, the free list for the appropriate size is used to satisfy the request. There are several variations on this strategy.

Simple segregated storage does not allow splitting of free blocks to satisfy requests for smaller sizes. When a request for a given size is serviced, and the free list for the appropriate size class is empty, more storage is requested from the underlying operating system; typically one or two virtual memory pages are requested at a time and split into same-sized blocks which are then strung together and put on free list. The advantage of this scheme is that it is quite fast,

but it is subject to potentially severe external fragmentation because no attempt is made to split or coalesce blocks to satisfy requests for other sizes.

Segregate fits is an alternative strategy that also uses an array of free lists, but each list holds free blocks of different sizes within a size class. When servicing a request for a particular size, the free list for the corresponding size class is searched for a block at least large enough to hold it. The search is typically a sequential fits search, and many significant variations are possible.

Buddy Systems

Buddy systems are a variant of segregated lists that support a limited but efficient kind of splitting and coalescing. In a simple buddy scheme, the entire heap area is conceptually split into two large areas, and those areas are further split into two smaller areas, and so on. This hierarchical division of memory is used to constrain where objects are allocated, what their allowable sizes should be and how they may be coalesced into larger free areas. For each allowable size, a separate free list is maintained, in an array of free lists. A free block may only be merged with its *buddy*, which is its unique neighbor at the same level in the binary hierarchical division. The resulting free block is there for always one of the free areas at the next higher level in the memory-division hierarchy-at any level, the first block may only be merged with the following block, which follows it in memory; conversely, the second block may only be merged with the first, which precedes it in memory. This constraint on coalescing ensures that the resulting merged free area will always be aligned on one of the boundaries of the hierarchical splitting. Buddy systems generally exhibit significantly more fragmentation than segregated fits and indexed fits. There are several variations of buddy systems. In a binary buddy system, all buddy sizes are a power of two, and each size is divided into two equal parts. This makes address computations simple, because all buddies are

aligned on a power-of-two boundary offset from the beginning of the heap area, and each bit in the offset of a block represents one level in the buddy system's hierarchical splitting memory. This is a result of the fact that if a bit is 0, then it is the first of a pair of buddies, and if it is 1, it is the second. These operations can be implemented efficiently with bitwise logical operations. A major problem with the binary buddy system is that its internal fragmentation is relatively high.

Indexed Fits

In this strategy the characteristics of interest are stored in an indexed data structure to allow efficient searching. The best known example of an indexed fits scheme is the fast fits allocator. This allocator uses a Cartesian tree sorted on both size and address, with address as the primary key. A Cartesian encodes two-dimensional information in a binary tree, using two constraints on the tree shape. It is effectively sorted on a primary key and a secondary key. The tree is a normal totally-ordered tree with respect to the primary key. With respect to the secondary key, it is a 'heap' data structure, i.e., a partially ordered tree whose nodes each have a value greater than their descendants. Cartesian trees give logarithmic search times for random inputs.

Bitmapped Fits

Bitmapped fits is a special case of indexed fits, where a *bitmap* is used to record which parts of the heap area are in use, and which parts are not. A bitmap is a simple vector of one-bit flags, with one bit corresponding to each word of the heap area. Bitmapped allocators are commonly used in garbage collectors, particularly in mark-sweep garbage collectors. Bitmapped allocators have two advantages compared to conventional schemes. One is that they support searching the free memory indexed by address order, or localized searching, where the search may begin at a carefully-chosen address. Another advantage is

that bitmaps are 'off to the side', i.e., not interleaved with the normal data storage area. This may be exploitable to improve the locality of searching itself, as opposed to traversing lists or trees embedded in the storage blocks themselves.

5.2 A Hierarchical Storage Manager

One of the fundamental problems with finding and using suitable storage allocators is that programmer awareness about the tradeoffs is limited[17]. An ideal storage strategy mechanism would keep track of which parts of the storage are in use, and which parts are free, with little wasted space, and almost with a constant response time for each request. But many of them have *ad hoc* storage management strategies built, with hidden limitations. The Hierarchical Storage Manager (HMM) addresses these issues by providing a formal specification.

An allocator can receive requests for new storage blocks and requests for releasing storage blocks in any order from the application program. This creates the potential for holes in the storage space, also known as fragmentation. There is no reliable algorithm for ensuring efficient storage usage, and it has been proven that for any possible allocation algorithm, there will always be the possibility that some application program will allocate and de-allocate blocks in some fashion that defeats the allocator's strategy, and forces severe fragmentation [17,25]. The underlying assumption in these arguments is that the storage is considered as a linear structure. In this view of storage the entire storage is considered as one block at the beginning. This is evident from the taxonomy of the allocators and their description provided in the previous section. We propose to view the storage as a hierarchical structure, where the entire storage can be considered as a hierarchy of a fixed-size blocks as shown in Figure 5.1. This view of the storage provides two advantages: predictability in terms of servicing a request for storage, and less external fragmentation. In this case, the amount of external fragmentation is not totally dependent on the pattern of allocate and de-allocate

requests coming in from the application programs. It is also a function of the size of each node in the hierarchical structure. The apparent disadvantage with this arrangement is the access time of a storage element, which is a logarithmic function of the storage size. On the other hand, this mechanism provides predictability for satisfying a request with tunable storage utilization.

Let us consider that there are **N** number of blocks in the system. The block size is tunable and it is application-dependent. For illustration purposes, let us say each block holds ENTRIES_PER_BLOCK number of basic storage units, or REFS_PER_BLOCK number of references. This basic storage unit, again, is implementation-dependent. It can be either in terms of bytes or words. The Max_Capacity is N times ENTRIES_PER_BLOCK.

To arrange the storage in an hierarchical manner, certain amount of storage is used for book keeping. This storage is called overhead blocks. The blocks needed to satisfy the request **r** and the overhead blocks can be computed by the equations given below:

Number of blocks needed to represent **r**	NUM_BLOCKS(**r**)=	Ceiling_Of(**r**/ENTRIES_PER_BLOCK)
Overhead blocks needed to represent **r**	OH_BLOCKS(r)=	**if** NUM_BLOCKS(r) = 1 **then** OH_BLOCKS(r) = 0 **else** OH_BLOCKS(r) = Ceiling_Of(NUM_BLOCKS(r)/REFS_PER_BLOCK))+ OH_BLOCKS(Ceiling_Of(NUM_BLOCKS(r)/REFS_PER_BLOCK))

Where ENTRIES_PER_BLOCK, is the number of basic storage units per blocks, such as bytes, words, etc., and REFS_PER_BLOCK is the number of pointers that can be stored per block. Let us consider an example where entry

size is 32 bits or words and each pointer takes up the same number of bits for a reference. If we consider the block size to be 8 words, then for a request of 11 words we need 2 blocks of data and 1 block of overhead, as computed below. This scenario is depicted in figure 5.2 (b).

NUM_BLOCKS(11)= Ceiling_of(11/8) = 2

OH_BLOCKS(11) = Ceiling_of(2/8) + OH_BLOCKS(Ceiling_of(NUM_BLOCKS(11)/8))

= Ceiling_of(2/8) + OH_BLOCKS(Ceiling_of(2/8))

= Ceiling_of(2/8) + OH_BLOCKS(1)

= 1 + 0

= 1

Let us consider another example with request size **r** 520. This request needs 65 blocks of storage for data and 10 blocks of storage for overhead. This scenario is depicted in Figure 5.2 (c).

NUM_BLOCKS(520)= Ceiling_of(520/8) = 65

OH_BLOCKS(264) = Ceiling_of(65/8) +

OH_BLOCKS(Ceiling_of(NUM_BLOCKS(520)/8))

= Ceiling_of(65/8) + OH_BLOCKS(Ceiling_of(65/8))

= 9 + OH_BLOCKS(9)

= 9 + Ceiling_of(2/8) + OH_BLOCKS(Ceiling_of(2/8))

$= 9 + 1+0= 10$

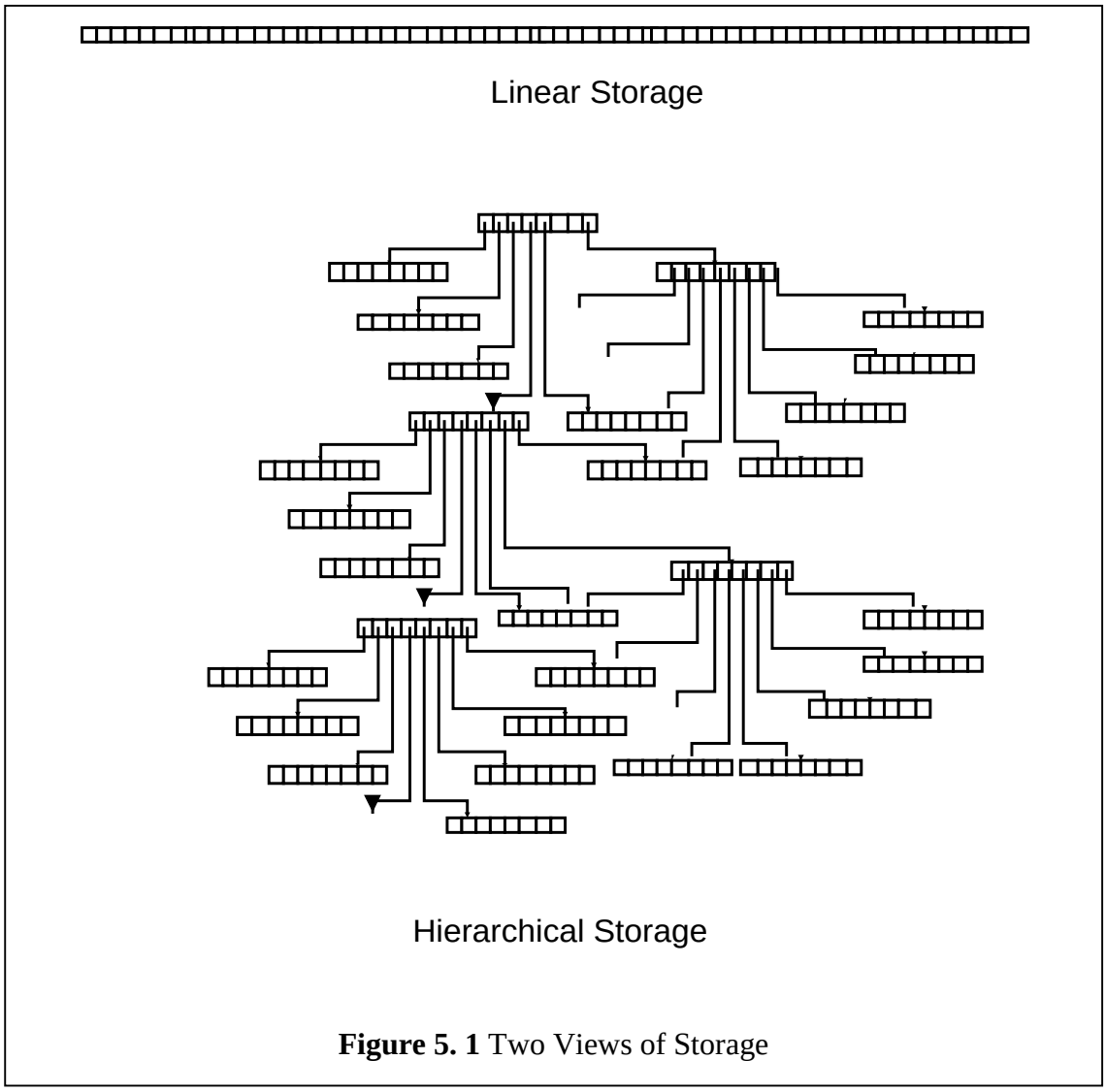


Figure 5. 1 Two Views of Storage

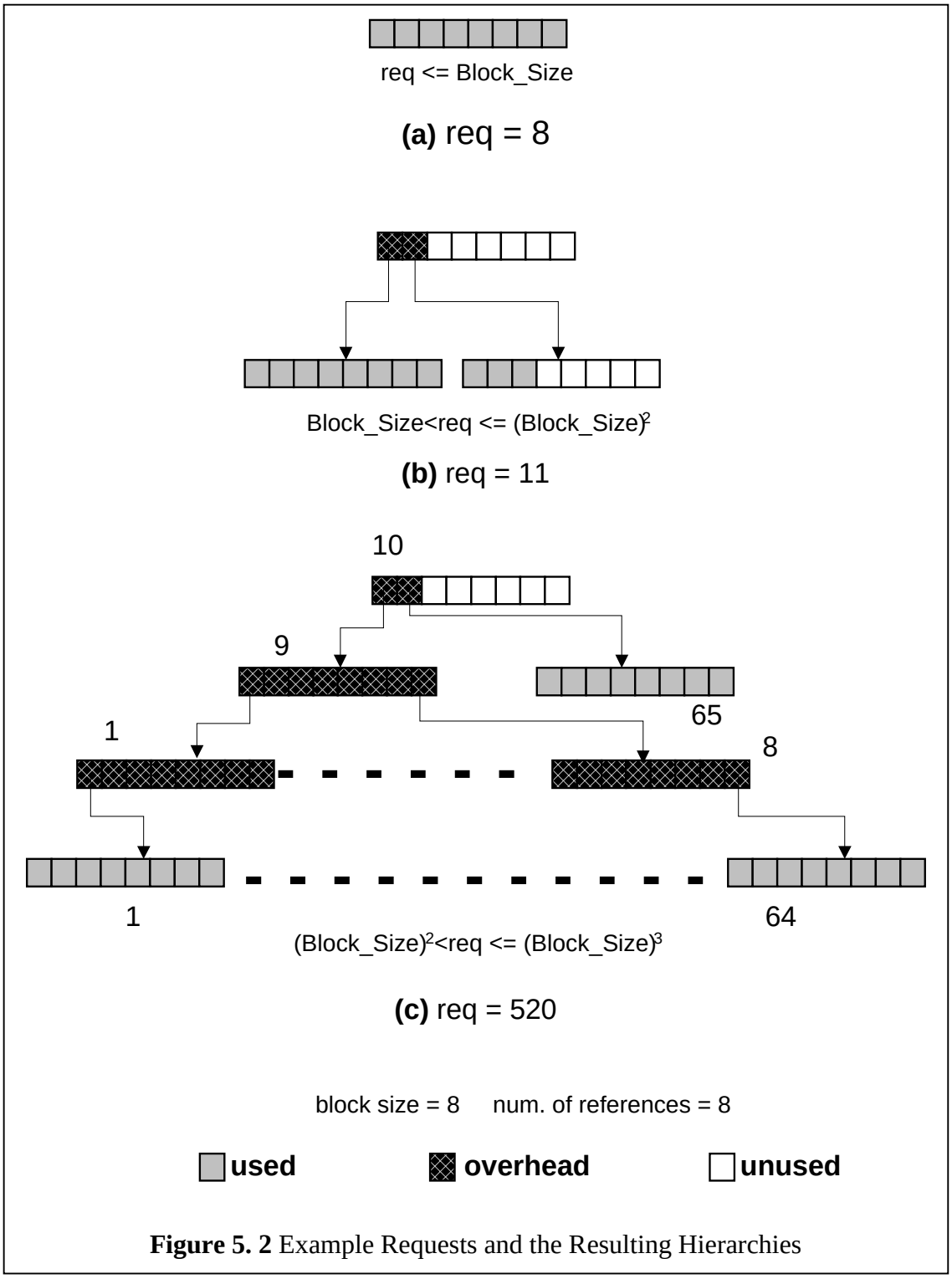


Figure 5. 2 Example Requests and the Resulting Hierarchies

5.3 Hierarchical Storage Manager Specification

Figure 5.3 presents a formal specification of the Hierarchical Storage Manager (HMM) concept. The concept is parameterized by the total number of blocks, and entries per block. Number of references per block is a constant defined in the local context. Number of references per block determines the number of indirect references that can be held in a block. This value influences the depth of the tree, and thus the access time of each block. In all of the diagrams presented in this chapter the number of references and number of entries per block are assumed to be equal. HMM is a communal concept, mathematically modeled by a tuple < contents: function from Integer to Entry, size; Integer>. The communal level bound for this concept is the entire system storage in terms of the total blocks, i.e. Max_Capacity. In effect there should be only one instance of this concept in a programming system at runtime. Under the interface section of the specification of this concept, a number of mathematical definitions are given. The definitions of Needed_Blocks and OH_Blocks specify how the number of needed blocks and overhead blocks have to be computed, respectively. The same formulas are used in the examples given in the previous section. Other definitions are used in the specification of operations of this concept.

There are four basic operations Extend, Shrink, Swap_Entry, and Extension_Limit, provided by this concept. Extend increases the size of the instance of this facility. This is analogous to an *allocate* or *new* operation on C/C++ based programming systems. The requires clause of this operation specifies that there must be the needed number of storage blocks available. The definitions of Additionally_Needed_Blocks and Rem_Blocks are used in the pre-condition of this operation. Additionally_Needed_Blocks is computed as the difference between the Needed_Blocks for current size plus the extension request and Needed_Blocks for the current size. Initially, the size of the object would be zero. Using this operation the required size of the object can be set.

This operation specification also has the requirement for capacity, and it is a constant, i.e., C_{Extend}. The ensures clause of this operation merely states that the new blocks are initialized, the size is increased, and all the other blocks remain the same. The operation Shrink reduces the size which is analogous to *free* or *delete* operations. The Swap_Entry operation is the access operation. This is how each element of this storage manages is accessed. The ensures clause specifies the effects of this swap operation. The interesting part of this specification is the duration, which is a logarithmic function of the size. This is a worst case access time, but this access time can be pre-determined. Theoretically, any single instance in a programming system can grow as big as the entire physical storage, and the worst case access time can not be more than that of this object. The Extension_Limit operation can be used to find out the maximum size to which the current instance can grow.

When all the components in a programming system are realized based on the Hierarchical Storage Manager, and there is only one facility of this type declared for the programming system, then the resulting system can avail the benefits of managing storage as a hierarchical structure. In traditional storage organization, each entry of storage is accessed by a physical address imposed by the hardware. HMM abstracts this view of the storage, so that each entry need not be consecutive, but can be accessed using an index into the abstract array concept that it provides. As the physical layout is organized into a hierarchy, the access time is a logarithmic function of the tree height. The total storage usage of the HMM is equivalent to $\sum C$, i.e. the total storage used by the program at any given point of time. This is specified as Used_Blocks ***** Entries_Per_Block, where the mathematical definition of Used_Blocks is given in the specification. The storage usage expression for each instance of HMM is given under the interface section of the specification as $C_{HMM} + \sum_{i=0}^{i= size-1} \mathbf{C(}contents(i)\mathbf{)} + \mathbf{C}_{Integer}$.

concept Hierarchical_Memory_Manager_Template

context

parametric context

type Entry

constant Entries_Per_Block: Integer

constraints Entries_Per_Block >= 1

constant Max_Total_Blocks: Integer

constraints Max_Total_Blocks >= 1

local context

constant Refs_Per_Block: Integer

constraints Refs_Per_Block >= 2

$C_{Extend} = 2 * C_{Reference}$

$C_{Shrink} = 2 * C_{Reference}$

$C_{Swap\_Entry} = 2 * C_{Reference}$

$C_{Extension\_Limit} = C_{Reference}$

interface

type HMM **is modeled by** (

contents: **function from** integer **to** Entry

size: integer

)

exemplar h

uses capacity $C_{HMM} + \sum_{i=0}^{i= size-1} \mathbf{C(}contents(i)\mathbf{)} + C_{Integer}$

initialization

requires capacity $\mathbf{C_{HMM}} + C_{Integer}$

ensures h.size = 0

duration C_I

finalization

ensures h.size = 0

duration C_F

definition Needed_Blocks(num_entries: integer): integer =
Ceiling_Of((num_entries/Entries_Per_Block))

inductive definition OH_Blocks(num_blocks: integer): integer **is**
if num_blocks = 1 **then** OH_Blocks = 0
else OH_Blocks =
Ceiling_Of(num_blocks/Refs_Per_Block))+
OH_Blocks(Ceiling_Of(num_blocks/Refs_Per_Block))

definition Additionally_Needed_Blocks(prev, new: integer): integer =
Needed_Blocks(prev + new) +
OH_Blocks(Needed_Blocks(prev + new)) –
Needed_Blocks(prev) - OH_Blocks(Needed_Blocks(prev))

definition Used_Blocks: integer =
$\sum_{i=1}^{last\_specimen\_num}$ (Needed_Blocks(HMM.Denote(i).size) +
OH_Blocks(HMM.Denote(i).size))

definition Rem_Blocks: integer = Max_Total_Blocks - Used_Blocks

(continued on the next page)

```
        operation Extend(     alters h: HMM,
                              preserves size: integer
                              )
            requires capacity C_Extend
            requires Additionally_Needed_Blocks(a.size, size) <= Rem_Blocks and
                        size > 0
            ensures a.size = #h.size + size and
                        for all i: integer,   if 0 <= i < #h.size
                                              then h.contents(i) = #h.contents
                                              else if #h.size <= i < h.size
                                              then Entry.Is_Initial(h.contents(i))
            duration C_E
        operation Shrink( alters h: HMM,
                          preserves size: integer
                          )
            requires capacity C_Shrink
            requires 0 < size <= h.size
            ensures h.size = #h.size - size and
                        for all i: integer,   if 0 <= i < h.size
                                              then h.contents(i) = #h.contents
            duration C_S
        operation Swap_Entry(   alters h: HMM,
                                peserves index: Integer,
                                alters x: Entry
                                )
            requires capacity C_Swap_Entry
            requires 0 <= index < h.size
            ensures h.size = #h.size and
                        x = #h.contents(index) and h.contents(index) = #x and
                        for all i: integer,   if 0 <= i < h.size and i/= index
                                              then h.contents(i) = #h.contents
            duration 1 + Needed_Blocks(size) log to the base of Refs_Per_Block
        operation Extension_Limit(preserves h: HMM): Integer
            requires capacity C_Extension_Limit
            ensures  Additionally_Needed_Blocks(h.size, Extension_Limit) +
                                Total_Blocks = Max_Total_Blocks
            duration C_EL
end Hierarchical_Memory_Manager_Template
```

Figure 5. 3 Hierarchical Storage Manager concept specification

5.4 Discussion

The most important advantage of HMM is that it has a formal specification, and hence storage requirements of objects based on it can be verified. Another major advantage of this scheme is that it is predictable. Satisfying a request would take an almost constant amount of time, which is the time to find the required number of blocks from the internal free list. But this flexibility results in a logarithmic increase in the access time. This is given in the *duration clause* in the specification of Swap_Entry given in Figure 5.3. The worst case access time can be computed based on the size of total available storage, and it will always be less than this worst case behavior. This scheme also optimizes the usage of storage compared to conventional storage management.

HMM is predictable in terms of space. For every allocation and deallocation request the space requirements can be determined based on its specifications. This deterministic behavior is not possible with any other popular storage managers. Also, since the storage is organized as a hierarchical structure, storage utilization is efficient. The allocation and deallocation times are almost constant, as there is no need for search or for compaction. The access time is also efficient with HMM, as the worst time behavior is a logarithmic function of the height of the hierarchy.

Figure 5.4 presents a schematic view of various components built on top of HMM. For example, each integer would be a single block with one entry being used. The internal fragmentation, with 8 entries for each block, is 87.5%. Decreasing the size of each block can reduce this internal fragmentation. For example, if we make the size of Integer 4 entries and the size of the block is 4 entries, this fragmentation can be dramatically reduced to zero. The optimal utilization depends on the programming system and its definition of primitive types in terms of their sizes, etc. Also, the underlying hardware can impose

certain restrictions on the sizes of the blocks. In Figure 4.5, a record is shown as a HMM with 3 entries under use. Two of them are references to other record, and the final one is a reference to an Integer which is a HMM instance. Similarly, in the case of stack implementation shown in Figure 4.5, it can grow dynamically as big as the entire available storage. The array data member for Stack is an instance of HMM, which would grow or shrink on demand. In all of these ADTs the worst case access time is a logarithmic function of the size of total storage in number of blocks. Unlike conventional storage managers, this is highly predictable, so that duration specification can be done on software components more precisely. The real contribution of this storage manager is that it has a specification of behavior, storage and duration. The storage utilization aspect of HMM has to be further investigated by developing a practical system and collecting empirical data.

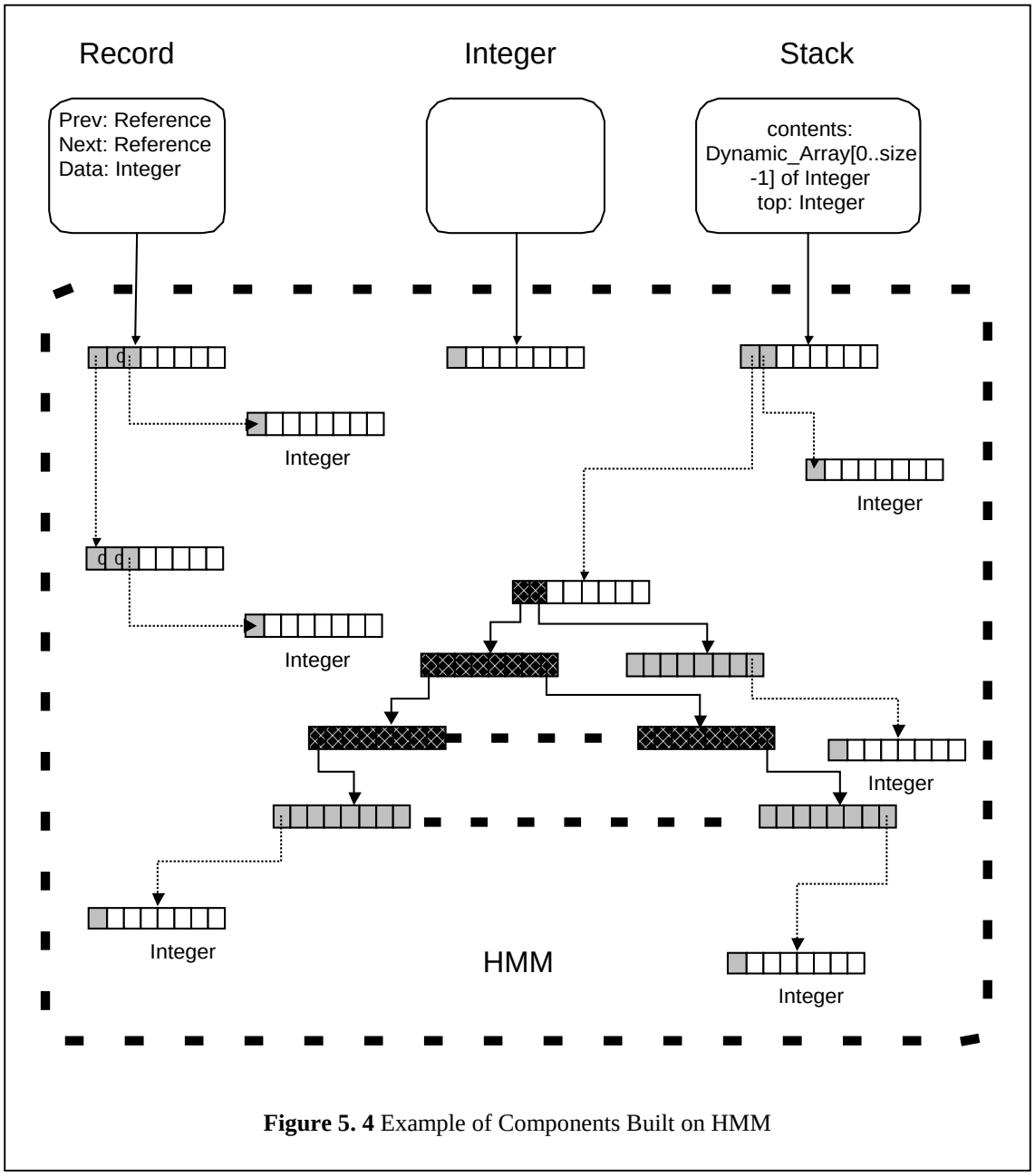


Figure 5. 4 Example of Components Built on HMM

Chapter 6

CONCLUSIONS

We started out stating the importance of specification and verification of storage in component based software development. In Chapter 2 we presented the alternate storage specification mechanisms that are currently in practice. We discussed the advantages and disadvantages of these schemes in this chapter. In Chapter 3 we presented a storage specification mechanism. Based on this mechanism we also showed how verification can be done using the *natural reasoning* mechanism in this chapter. We provided examples of storage specification for a variety of components in this chapter. Further, we showed how recursive procedures are handled in the storage verification using a fact/obligation table approach. In chapter 4 we raised the issue of relative completeness when the storage specification is done at the same level of abstraction as the functional specification. The need for intermediate levels of abstraction was presented in this chapter along with examples. In this chapter we conclude that in order to specify tight bounds on storage, its specification can not be done at the functional level, instead it has to be done at an intermediate level of abstraction. A framework for specifying the storage constraints in realization headers is proposed in this chapter. In the next chapter, i.e. Chapter 5 we have proposed a storage manager that is different from traditional storage managers, and provides flexibility, efficiency and predictability.

6.1 Possibilities for Future Work

Based on the proposed storage specification mechanism, facts and obligations similar to natural reasoning of functional specification can be automatically generated. These obligations then can be provided to a theorem prover such as

PVS and can be verified. This is ideal for highly reliable and scalable component-based software development environments.

There is an interaction between duration and storage requirements of a component. For example, when tight constraints on duration was specified, the resulting intermediate models often tend to use more storage. The impact of this observation is not clear at this time, but it may open up some research possibilities.

The storage management scheme presented in Chapter 5 is not corroborated by any empirical data. There is a possibility for future research work in formal proofs of the mechanism on storage utilization and demonstrating the effectiveness with empirical data.

6.2 Contributions

Our research has defended the following thesis:

1. The proposed framework would allow reasoning about a program statically, and in a modular fashion, so that a component based system would not run "out of memory", given constrains on input parameters, and maximum storage available.

2. The reasoning system presented is modular, and allows specifying storage as precise as necessary at alternate levels of abstraction.

3. The proposed storage management mechanism, HMM, is formally specified. It is efficient, and predictable. It also provides optimal storage utilization.

Specification of storage at functional level abstraction can be used in cases where only upper bounds have to be verified. The runtime storage requirement, transient storage, can be computed based on the programming environment.

When the storage requirements for base types are built into the program environment, computing the cumulative storage requirements for user defined types is straightforward and can be done syntactically.

REFERENCES

[1]. [Kozaczynski 96] Kozaczynck, V. and Ning, J. Q., Panel discussion on Component Based Software Engineering, Procs. Fourth Int. Conf. On Software Reuse, IEEE Computer Society Press, 1996, 236-242

[2]. [Szyperski 98] Szyperski, C. Component Software, Beyond Object-Oriented Programming, Addison-Wesley, 1998

[3]. ____, 10/20/98, http://www.javasoft.com.

[4]. Garlan, D. and Perry, D., "Introduction to the Special Issue on Software Architecture," IEEE Transactions on Software Engineering, April 1995.

[5]. Leavens, G. T., and Nierstrascx, O., Sitaraman, M., "1997 Workshop on Foundations of Component-Based Systems," ACM SIGSOFT Software Engineering Notes 23, NO. 1, January 1998, pp.38-41

[6]. Proceedings of NASA Focus on Software Reuse Workshop, Fairfax, Virginia, Spetember 1996.

[7]. Szyperski, C. "Component Software, Beyond Object-Oriented Programming," Addison-Wesely, 1998.

[8]. Williams, T., Reusable Components for Evolving Systems, Procs. Fifth Int. Conf. On Software Reuse, IEEE Computer Society Press, 1998, 12-16.

[9]. Ernst, G. W., Hookaway, R. J., and Ogden, W. F., "Modular Verification of Data Abstractions with Shared Realizations," IEEE Transactions on Software Engineering, Vol.20, No. 4, April 1994.

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

[11]. Ernst, G. W., Hookway, R. J., Menegay, J. A., and Ogden, W. F., "Modular Verification of Ada Generics," Computer Languages 16. ¾, 1991, pp. 259-280.

[12]. Liskov, B., and Guttang, J., "Abstraction and Specification in Program Development," McGraw-Hill, New York, 1986.

[13]. Luckham, D., "Programming with Specifications," Springer-Verlag, 1990.

[14]. Meyer, B. "On Formalism in Specifications," IEEE Software 1(2), January 1985, 6-26.

[15]. Wing, J. M., "A Specifier's Introduction to Formal Methods," IEEE Computer, 29(9), September 1990, 8-24.

[16]. Daniela Ivan Rosu, Karsten Schwan, Sudhakar Yalamanchili, and Rakesh Jha, "On Adaptive Resource Allocation for Complex Real-Time Applications," Proceedings of the 18th IEEE Real-Time Systems Symposium, San Francisco, December 1997.

[17]. Paul R. Wilson, Mark S. Johnston, Michael Neely, and David Boles, "Dynamic Storage Allocation: A Survey and Critical Review," Department of Computer Sciences, University of Texas at Austin, Austin, Texas, 78751, USA.

[18]. David, Detlefs, Al Dosser, and Benjamin Zorn, "Memory Allocation Costs in Large C and C++ Programs," Technical Report CU-CS-665-93, Department of Computer Science, University of Colorado at Boulder.

[19].Krone, J., and Sitaraman, M. "Expressive Specification and Modular Verification of Performance of Generic Data Abstractions," 1995.

[20]. Sitaraman, M., "On Tight Performance Specification of Object Oriented Software Components," Proceedings of the 1994 International Conference on Software Reuse, Ed. W. Frakes, IEEE Computer Society Press, November 1994, 149-157.

[21]. Spivey, J.M, "The Z notation: A Reference Manual," Prentice Hall, 1989.

[22]. C.A.R. Hoare, "An Axiomatic Basis For Computer Programming," Comm. ACM 29, January 1985, 80-86

[23]. J. Krone, "Proof Rule for Modules," Technical Report, Dept. of Computer and Info. Science, The Ohio State University, 1988.

[24]. Heym, W.D. "Computer Program Verification: Improvements for Human Reasoning." Ph. D. diss., Dept. of Comp. And Inf. Sci., The Ohio State Univ., Columbus, OH, 1995.

[25].Murali Sitaraman, "Impact of Performance Considerations on Formal Specification Design," Formal Aspects of Computing, Springer Verlog, 1997.

[26]. Brenad S. Baker, Edward G. Coffman, Jr. and Dan E. Willard, "Algorithms for Resolving Conflicts in Dynamic Storage Allocation," Journal of the Association for Computing Machinery, Vol.32, No.2, April 1985

[27]. Booch, G., "Software Components with Ada," Benjamin-Cummings, 1987

```
concept Static_Array_Template
  context
        global context
          facility Standard_Integer_Facility
        parametric context
                type Entry
                math operation
                        LOWER_BOUND : Integer
                math operation
                        UPPER_BOUND : Integer
  interface
    type Static_Array is modeled by function from Integer to math[Entry]
      exemplar a
      constraint  for all i : Integer (
                (i < LOWER_BOUND  or i > UPPER_BOUND)  implies
                        Item.init(a(i)))
      initialization
        ensures    for all i : Integer (
                  Item.init(a(i)))
    operation Get_Bounds (
        preserves a : Static_Array
        produces  lower : Integer
        produces  upper : Integer
      )
      ensures    lower = LOWER_BOUND  and
                 upper = UPPER_BOUND
    operation Swap_Entry (
        alters    a : Static_Array
        preserves i : Integer
        alters    x : Item
      )
      requires   LOWER_BOUND <= i <= UPPER_BOUND
      ensures    (for all j : Integer (
                j /= i  implies  a(j) = #a(j)))  and
                a(i) = #x and
                x = #a(i)
end Static_Array_Template
```

Figure A.1 Static_Array_Template

```
realization body Static_Array_With_Top_Index concept Bounded_Stack_Template
context
      local context
           Facility SA is Static_Array_Template(Entry,0,Max_Depth-1) realized by Pointer_Based
interface
    type Stack is represented by
           record
             contents: SA
             top  : Integer
           end
    convention
       0 <= s.top < Max_Depth
    correspondence
      conc.s = <s.contents (i-1) > i=s.top to 1

 procedure Push(
              alters   s : Stack
              consumes x : Entry
            )
  begin
   s.top:=s.top+1
   s.contents[s.top] :=: x
  end

  procedure Pop(
             alters    s : Stack
             produces  x : Entry
           )
  context variables
   temp : Entry
  begin
   temp :=: s.contents[s.top]
   x :=: temp
   s.top := s.top-1
  end

  function procedure Depth_Of(
                 preserves s : Stack
               ) : Integer
  begin
   return (s.top)
  end

  function procedure Max_Depth () : Integer
  begin
   return (Max_Depth)
  end

end Static_Array_With_Top_Index
```

Figure A.2 Bounded Stack Realization

```
concept Ceramic_Array_Template
context
    global context
        facility Standard_Integer_Facility
    parametric context
        type Item
    local context
      math subtype Ceramic_Array is (
            contents    : function from integer to math[Item],
            lower_bound : integer,
            upper_bound : integer
        )
        exemplar a
        constraint
          for all i : integer (
            (i < a.lower_bound  or  i > a.upper_bound)
             implies  is_initial (a.contents(i)))
  interface
    type Ceramic_Array is modeled by ARRAY_MODEL
      exemplar a
      initialization
        ensures     a.lower_bound = 0   and
                    a.upper_bound = -1
    operation Get_Bounds (
        preserves a     : Ceramic_Array
        produces  lower : Integer
        produces  upper : Integer
      )
      ensures     lower = a.lower_bound  and
                  upper = a.upper_bound
    operation Set_Bounds (
        alters    a     : Ceramic_Array
        preserves lower : Integer
        preserves upper : Integer
      )
      requires    a.lower_bound > a.upper_bound  and
                  lower <= upper
      ensures     a.lower_bound = lower  and
                  a.upper_bound = upper  and
                  for all i : integer (
                    is_initial (a.contents(i)))
```

(continued in the next page)

```
    operation Swap_Entry (
                                alters    a : Ceramic_Array
                                preserves i : Integer
                                alters    x : Item
                                )
        requires    a.lower_bound <= i <= a.upper_bound
        ensures     differ(a.contents, #a.contents, { i })  and
                    a.contents(i) = #x                      and
                    x = #a.contents(i)

end Ceramic_Array_Template
```

Figure A.3 Ceramic Array_Template

```
realization body Ceramic_Array_With_Top_Index concept Ceramic_Stack_Template
      context
           local context
                Facility CA is Ceramic_Array_Template(Entry) realized by Pointer_Based
interface
    type Stack is represented by
           record
             contents: CA
             top  : Integer
           end
    convention
       0 <= s.top < (contents.upper_bound – contents.lower_bound )
    correspondence
      conc.s = <s.contents (i-1+contents.lower_bound) > i=s.top to 1

procedure Set_Max_Depth(
                                    alters s: Stack
                                    preserves max_depth: Integer
                                    )
begin
      s.contents.Set_Bounds( s.contents,0,max_depth-1);
end

procedure Get_Max-Depth(
                                    preserves s: Stack
                                    ): Integer
begin
          return((s.contents.upper_bound – s.contents.lower_bound) + 1)
end

 procedure Push(
                     alters   s : Stack
                     consumes x : Entry
                     )
  begin
    s.top:=s.top+1
    s.contents[s.top] :=: x
  end

  procedure Pop(
                     alters    s : Stack
                     produces  x : Entry
                     )
  context variables
    temp : Entry
  begin
    temp :=: s.contents[s.top]
    x :=: temp
    s.top := s.top-1
  end
```

(continued in the next page)

```
 function procedure Depth_Of(
                  preserves s : Stack
                ) : Integer
 begin
  return (s.top)
 end

 function procedure Max_Depth () : Integer
 begin
   return (s.contents.upper_bound – s.contents.lower_bound + 1)
 end

end Static_Array_With_Top_Index
```

Figure A.4 Ceramic Stack Realization

concept List_Template
context
global context
facility Standard_Boolean_Facility
facility Standard_Integer_Facility
local context
$C_{Insert} = 2*C_{Reference}$
$C_{Remove} = 2*C_{Reference}$
$C_{Advance} = C_{Reference}$
parametric context
type Entry
interface
type List **is modeled by** (left : string of Entry right : string of Entry)
exemplar s
uses capacity C_{List} + (|s.left| + |s.right|)*$C_{Entry\_Overhead}$ + $\Sigma_{I=x}$ **C**(x),
where IS_ENTRY_OF(s.left * s.right,x)
initialization
ensures s.left = empty_string **and** s.right = empty_string
operation Insert(
alters s : List
consumes x : Entry)
)
requires capacity C_{Insert} + $C_{Entry\_Overhead}$
ensures s.left=#s.left **and** s.right=<#x> * #s.right
operation Remove(
alters s : List
produces x : Entry
)
requires |s.right| != 0
requires capacity C_{Remove}
ensures s.left = #s.left and #s.right = <x> * s.right
operation Advance (
alters s : List
)
requires |s.right| != 0
requires capacity $C_{Advance}$
ensures s.left * s.right = #s.left * #s.right and
|s.left|=|#s.left|+1

```
        operation Reset(
                        alters s : List
                        )
                        requires capacity C_Reset
                        ensures |s.left|=0 and s.right=#s.left * #s.right
        operation Advance_To_End(
                                        alters s : List
                                        )
                        requires capacity C_Advace_To_End
                        ensures |s.right|=0 and s.left=#s.left * #s.right
        operation Reset(
                        alters s : List
                        )
                        requires capacity C_Reset
                        ensures |s.left|=0 and s.right=#s.left * #s.right
        operation Swap_Rights(
                                        alters s1 : List
                                        alters s2 : List
                                )
                                requires capacity C_Swap_Right
                                ensures s1.left = #s1.left and s2.left = #s2.left and
                                   s1.right = #s2.right and s2.right = #s1.right
        operation Left_Length_Of(
                                   preserves s : List
                                           ) : Integer
                                   requires capacity C_Left_Length_Of
                                   ensures Left_Length_Of = |s.left|
        operation Right_Length_Of(
                                        preserves s : List
                                        ) : Integer
                                   requires capacity C_Right_Length_Of
                                   ensures Right_Length_Of = |s.right|

end List_Template
```

Figure B.1 List Concept specification

```
concept Bounded_Integer_Template
 context
   global context
    facility Standard_Boolean_Facility
    mathematics  Integer_Theory
   local context
    math operation Min_Int : integer
        definition Min_Int <= 0
    math operation Max_Int : integer
        definition Max_Int > Min_Int
 interface
  type Integer is modeled by integer
  exemplar i
        constraint   Min_Int <= i <= Max_Int
uses capacity Ci
   initialization
        requires capacity Ci
 ensures  i = 0
 operation Add(preserves i : Integer,
                preserves j : Integer,
                produces  result : Integer)
                requires   Min_Int <= i+j <= Max_Int
                requires capacity C_Add
                ensures    result = i+j
 operation Multiply(preserves i : Integer,
                    preserves j : Integer,
                    produces  result : Integer)
                    requires   Min_Int <= i*j <= Max_Int
                    requires capacity C_Multiply
                    ensures    result = i*j
 operation Subtract(preserves i : Integer,
                    preserves j : Integer,
                    produces  result : Integer)
                    requires   Min_Int <= i-j <= Max_Int
                    requires capacity C_Substract
                    ensures    result = i-j

        --
        --Other operation are omitted
        --
end Bounded_Integer_Template
```

Figure B.2 Integer storage specification

```
concept Record_Template
 context
  global context
    mathematics Set_Theory
  parametric context
    type T1
    type T2
interface
  type Record is modeled by tuple
              f1 : T1
              f2 : T2
             end
   exemplar r
        uses capacity Cr2 + C(r.f1) + C(r.f2)
   initialization
        requires capacity Cr2 + C(T1.initial_value ) +
                                C(T2.initial_value)
        ensures  r.f1 = initial_value and r.f2 = initial_value
  operation Swap_First_Coord(
                                  alters   r : Record
                                  alters   x : T1
                                  )
                                  requires capacity C_ref
                                  ensures   r.f1 = #x and r.f2 = #r.f2 and
                                            x= #r.f1

  operation Swap_Second_Coord(
                                     alters   r : Record
                                     alters   x : T2
                                     )
                                   requires capacity C_ref
                                   ensures   r.f2 = #x and r.f1 = #r.f1 and
                                             x= #r.f2
end Record_Template
```

Figure B.3 Storage specification of a Record with two fields

```
concept Static_Array_Template
  context
        global context
           facility Standard_Integer_Facility
        parametric context
                type Entry
                math operation
                        LOWER_BOUND : Integer
                math operation
                        UPPER_BOUND : Integer
  interface
    type Static_Array is modeled by function from Integer to math[Entry]
      exemplar a
      constraint  for all i : Integer (
                (i < LOWER_BOUND  or i > UPPER_BOUND)  implies
                        Item.init(a(i)))
      uses capacity  C_sa +   Σ_{i= LOWER_BOUND}^{UPPER_BOUND} C(a(i))
      initialization
        ensures    for all i : Integer (
                                Item.init(a(i)))
        requires capacity C_sa + (UPPER_BOUND - LOWER_BOUND) *
C(Entry.initial_value)
    operation Get_Bounds (
        preserves a : Static_Array
        produces  lower : Integer
        produces  upper : Integer
      )
      ensures    lower = LOWER_BOUND  and
                 upper = UPPER_BOUND
    operation Swap_Entry (
        alters   a : Static_Array
        preserves i : Integer
        alters   x : Item
      )
      requires   LOWER_BOUND <= i <= UPPER_BOUND
      ensures    (for all j : Integer (
                j /= i  implies  a(j) = #a(j)))  and
                a(i) = #x and
                x = #a(i)
end Static_Array_Template
```

Figure B.4 Static_Array_Template

```
concept Ceramic_Array_Template
context
    global context
        facility Standard_Integer_Facility
    parametric context
        type Item
    local context
      math subtype Ceramic_Array is (
            contents    : function from integer to math[Item],
            lower_bound : integer,
            upper_bound : integer
        )
        exemplar a
        constraint
          for all i : integer (
            (i < a.lower_bound  or  i > a.upper_bound)
             implies  is_initial (a.contents(i)))
  interface
    type Ceramic_Array is modeled by ARRAY_MODEL
      exemplar a
            uses capacity Cca +  Σ i=a.lower_bound^a.upper_bound (C(a(i))
       initialization
          ensures      a.lower_bound = 0   and
                       a.upper_bound = -1
           requires capacity Cca
    operation Get_Bounds (
        preserves a     : Ceramic_Array
        produces  lower : Integer
        produces  upper : Integer
      )
      ensures     lower = a.lower_bound  and
                  upper = a.upper_bound
    operation Set_Bounds (
        alters    a     : Ceramic_Array
        preserves lower : Integer
        preserves upper : Integer
      )
      requires    a.lower_bound > a.upper_bound  and
                  lower <= upper
      ensures     a.lower_bound = lower  and
                  a.upper_bound = upper  and
                  for all i : integer (
                    is_initial (a.contents(i)))
```

(continued in the next page)

```
    operation Swap_Entry (
                                alters    a : Ceramic_Array
                                preserves i : Integer
                                alters    x : Item
                                )
        requires    a.lower_bound <= i <= a.upper_bound
        ensures     differ(a.contents, #a.contents, { i })  and
                    a.contents(i) = #x                      and
                    x = #a.contents(i)

end Ceramic_Array_Template
```

Figure B.5 Ceramic Array_Template