An Introduction To RESOLVE/Ada95

Technical Report OSU-CISRC-4/97-TR23

11 April 1997

David S. Gibson

The Ohio State University

Department of Computer and Information Science

Reusable Software Research Group

http://www.cis.ohio-state.edu/hypertext/rsrg

Contents

Preface

This report introduces RESOLVE/Ada95, a discipline for building high-quality software components using the 1995 revision to Ada. RESOLVE/Ada95 is an approach to implementing software components in Ada95 using the principles of RESOLVE[1]. RESOLVE is three things. First, RESOLVE is a framework for software component engineering. RESOLVE is also a language which includes two integrated sub-languages: a model-based formal specification language and an imperative, sequential, programming language. Finally, RESOLVE is a discipline with detailed design principles which guide software engineers in the development of high quality software components and systems. The RESOLVE discipline has been applied to the Ada and C++ programming languages. This report demonstrates how the RESOLVE discipline may be applied to Ada95.

RESOLVE/Ada95 is a major revision to the RESOLVE/Ada discipline[2] which was based on the 1983 Ada language definition. RESOLVE/Ada95 uses many of the new language mechanisms added to Ada in 1995 and in some aspects more closely resembles RESOLVE/C++ than the original RESOLVE/Ada. Neither Ada nor C++ provides an ideal set of language mechanisms for software component engineering. The RESOLVE language, which was designed to support software component engineering, can express notions which are either impossible or extremely awkward to encode in Ada95 and C++. However, Ada and C++ have one advantage over the RESOLVE language. These languages have commercially supported compilers available on a wide variety of platforms. Thus RESOLVE/Ada95 and RESOLVE/C++ make it easier for software engineers to apply the RESOLVE framework to the implementation of components.

The RESOLVE/Ada95 discipline presented in the following sections is one of many possible strategies for applying the RESOLVE discipline to Ada. This version of RESOLVE/Ada95 attempts to explicitly express as many aspects of the RESOLVE language as possible. Due to the current immaturity of Ada compilers supporting

---

1 For a comprehensive description of RESOLVE, see "Special Feature: Component-Based Software Using RESOLVE" is *ACM SIGSOFT Software Engineering Notes*, 19(4) 21-67, 1994.

2 The original RESOLVE/Ada discipline is described in Joseph E. Hollingsworth's 1992 Ohio State University Ph.D. thesis: *Software Component Design-for-Reuse: A Language-Independent Discipline Applied to Ada*.

the 1995 language definition, it has not been possible to test the practicality of this approach on large systems. As Ada compilers become more robust, the RESOLVE/Ada95 discipline may be revised to ensure it provides a viable approach for building large systems with available Ada compilers.

The following sections explain how software components and their relationships are encoded in RESOLVE/Ada95. RESOLVE/Ada95-specific component coupling diagrams and component instantiation diagrams graphically depict component relationships expressed in the sample code. While the reader should be familiar with Ada, the component relationships are not described in Ada-specific terms. Also, the use of new Ada language mechanisms (those added in 1995) is described in some detail. Each section includes sample RESOLVE/Ada95 code demonstrating the concepts discussed. Component instantiation diagrams summarize the component relationships and parameterization requirements of each unit of sample code. Note that these sections do not address all aspects of the RESOLVE/Ada95 discipline.

1. Kernel Abstract Components

An abstract software component provides an abstract description of a subsystem. It describes a structural interface including names of types and names and signatures of operations. It also formally describes functional behavior in a manner completely neutral as to how that behavior might be achieved. In RESOLVE/Ada95, abstract components may either be ***kernel abstract components*** or extensions to abstract components. This section describes abstract kernel components and Section 3 describes extensions to abstract components.

1.1. The Role of Kernel Abstract Components

A kernel abstract component is an abstract description which typically has no dependencies on other specific components. For example, a kernel abstract component might be used to describe a subsystem providing a queue that holds integers. In RESOLVE/Ada95, this description would be based on a *mathematical model* of a queue of integers such as a *string* of integers. Here the term *string* refers to the mathematical concept of a string defined by *string theory*. The structural interface of queue a of integers would be described by providing a type name, such as `Integer_Queue`, and the signatures of operations on that type such as `Enqueue` and `Dequeue`. The functional behavior of the integer queue operations would be described in terms of the queue's mathematical model. For example, the behavior of an initialization operation could be described as returning an empty string of integers. The behavior of an `Enqueue` operation could be described as ensuring that the integer to be enqueued is placed at the right end of the string of integers modeling the queue. While a description such as this does depend on mathematical string theory, it does not depend on any other software components.

A kernel abstract component describing a queue of integers would be an *abstract instance*. A more useful kernel abstract component would be an *abstract template* describing the behavior of a generic queue. Such an abstract template would be parameterized by the type of item to be contained in instances of the template. An abstract instance of a queue abstract template might depend on another specific component providing the item type. However, the queue abstract template itself need not depend on any other specific component. In order to support greater reusability, most RESOLVE/Ada95 kernel abstract components are abstract templates as opposed to abstract instances.

Component coupling diagrams were introduced in Section 3.3.3. Figure 1.1 shows a component coupling diagram for a queue abstract template. The component name

"AT_Queue" follows the RESOLVE/Ada95 naming conventions described in the next section. Since this abstract component does not depend on any other software components, no other related components are shown in the diagram. The component coupling diagrams in subsequent sections will build upon this simple diagram by showing how other components depend on kernel abstract components.

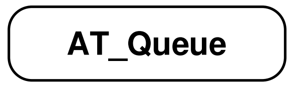

Figure 1.1 — A Queue Kernel Abstract Component

The reader familiar with object-oriented programming might notice that kernel abstract components are similar to *abstract base classes*. An abstract base class, however, only provides a structural interface "signature" with no explicit behavioral description. In RESOLVE components, formal behavioral descriptions are expressed directly using specification elements of the language. In the RESOLVE/Ada95 discipline, formal behavioral descriptions are provided as structured comments. In the examples in this report, the RESOLVE specification sub-language is used for writing these specifications. Other model-based specification languages could also be used for this purpose.

One of the aspects of the RESOLVE framework that most distinguishes it from other disciplines is its use of ***swapping***. RESOLVE uses swapping instead of assignment as the primary method for data movement. The use of swapping provides profound benefits and is one of the cornerstones of the RESOLVE framework[1]. The `Swap` operation simply exchanges the values of its two operands. `Swap` provides the efficiency of assignment implemented by copying a pointer (shallow copying) while maintaining value semantics like assignment implemented by copying a value (deep copying). Using swapping avoids the aliasing problems which result from shallow copying and which complicate reasoning about the behavior of components. An important insight on which the RESOLVE framework capitalizes is that deep copying is rarely needed when components are designed to

---

[1] A detailed description of, and rationale for, swapping appears in the article "Copying and Swapping: Influences on The Design of Reusable Software Components" in the May 1991 issue of *IEEE Transactions on Software Engineering*.

use swapping. RESOLVE/Ada95's support for swapping as the primary data movement operation has significant implications that affect the way in which Ada is used to encode RESOLVE/Ada95 kernel abstract components.

1.2. Encoding Kernel Abstract Components

A RESOLVE/Ada95 kernel abstract component is encoded as an Ada package, usually generic, which exports an ***abstract type*** and ***abstract primitive operations*** of the type. The exported abstract type is declared as an ***abstract tagged limited private type***. In Ada abstract types are ***tagged types*** from which no objects (variables) may be declared. Ada tagged types are types from which new user-defined types may be derived using ***type extension***, a form of inheritance. Primitive operations of a tagged type are those operations declared in the same package as the tagged type and having at least one parameter of the tagged type. In Ada, only primitive operations are inherited by new types derived from another tagged type. (In C++ terms, the exported tagged type is the *class* and the primitive operations are *member functions*.)

The type exported by a kernel abstract component is *limited private* which means that assignment and equality operations are *not* automatically defined for objects of the type. Since RESOLVE/Ada95 uses swap instead of assignment as the primary data movement operation, a primitive assignment operation is unnecessary. In situations where the deep copying functionality of assignment is needed, it may be added with an extension component as described in Section 3. While it would be technically possible to have deep copy assignment as a primitive operation, the automatically defined equality is an Ada *function* and is not compatible with RESOLVE/Ada95 as explained below. Thus, the use of limited private types is the only choice for RESOLVE/Ada95 components.

An unfortunate requirement of Ada functions is that their parameters must be "***in***" ***mode***. The intent of this requirement is to ensure that a function implementation does not change the values of the actual parameters used in a function call. The problem with this requirement is that it is both overly restrictive and largely ineffective in achieving its intended purpose. For example, consider an operation to determine if two queues are equivalent. An implementation of this operation should be permitted to remove and compare items from each queue as long as it returns the queues back to their original values before completing execution. Ada functions *can* be used to achieve such an implementation if a queue is represented by a pointer (an ***access type***) to the queue data structure. However, Ada does not allow an "in" mode formal parameter to be passed as an actual parameter to another operation where an "***in out***" ***mode*** parameter is required (such as an operation to remove items from a queue). Thus Ada functions cannot be used if their implementations may need to directly or indirectly alter formal parameter values.

For many styles of Ada programming, the “in” mode restriction on functions does not create serious problems. For RESOLVE/Ada95, however, this restriction precludes the use of functions as primitive operations. The RESOLVE counterpart to an “in” mode parameter is a ***preserves mode*** parameter. An operation is permitted to change the *concrete value* of a preserves mode parameter as long as there is no net change in the *abstract value* of the parameter. In addition to preserves mode, RESOLVE parameter modes include ***alters mode***, ***produces mode***, and ***consumes mode***. An operation may change the abstract value of an *alters mode* parameter. The initial abstract value of a *produces mode* parameter is irrelevant to an operation’s effect. The initial value of a *consumes mode* parameter is generally relevant to an operation’s effect and its final returned value must be an *initial value* of its type. In RESOLVE/Ada95 all operations are encoded as procedures. All parameters, regardless of the RESOLVE mode, use Ada’s “in out” mode except for preserves mode parameters of built-in scalar types. In this situation “in” mode is used in order to allow scalar literal values as actual parameters. Formal comments identify the RESOLVE mode of each procedure parameter in RESOLVE/Ada95.

All RESOLVE components provide three standard operations: `Initialize`, `Finalize`, and `Swap`. `Initialize` sets the abstract value of an object to an initial value of its type. RESOLVE guarantees that all objects are automatically initialized when they come into existence. `Finalize` is used to reclaim system resources allocated to an object. RESOLVE guarantees that all objects are automatically finalized immediately before they cease to exist. The functional behavior of `Finalize` generally is not specified. `Swap` exchanges the abstract values of two objects as described above.

RESOLVE/Ada95 uses Ada’s ***controlled types*** to achieve automatic initialization and finalization of all objects except built-in scalars. A controlled type is a tagged type derived from either `Controlled` or `Limited_Controlled`, two types defined in the built-in package `Ada.Finalization`. `Limited_Controlled` has two primitive operations: `Initialize` and `Finalize`. These operations have null-body implementations which may be overridden as necessary for descendent types derived from `Limited_Controlled`. While the exact details are quite involved, Ada basically ensures that an `Initialize` operation is automatically called when an object of a controlled type comes into existence.

Similarly, Ada ensures that a `Finalize` operation is automatically called immediately before a controlled type object ceases to exist.

RESOLVE/Ada95 kernel abstract components derive their exported type from `Limited_Controlled` with a ***null record extension*** in the ***private*** part of the package. The private derivation ensures that `Initialize` and `Finalize` cannot be called explicitly. The null record extension adds no data representation to the type `Limited_Controlled` and thus effectively leaves the data representation of the exported type unspecified. (Due to accessibility rules of Ada, the derivation from `Limited_Controlled` must take place in the kernel abstract component and cannot be placed in a concrete component, which would be a more appropriate location.)

In order to derive the exported type from `Limited_Controlled`, the package encoding a kernel abstract component has a fixed dependency on the built-in package `Ada.Finalization`. All fixed dependencies between components are expressed using an Ada `with` ***context clause*** in the ***global context*** section of RESOLVE/Ada95 components.

In addition to the implicit operations `Initialize` and `Finalize` inherited from `Limited_Controlled`, all kernel abstract components explicitly export `Swap` and other ***abstract operations***. An abstract operation is a primitive operation which must be overridden with an implementation for a non-abstract type derived from the abstract type. The exported abstract type and abstract operations along with formal specifications embedded in structured comments provide the kernel abstract component's structural and behavioral interface description.

Kernel abstract components are typically abstract templates parameterized by other components. These abstract templates are encoded in RESOLVE/Ada95 as generic packages. The abstract template parameters are included in a ***specification parameters section*** and are encoded as Ada generic formal parameters. Each component imported as a specification parameter is encoded as a ***limited private generic formal type parameter***. Each generic formal type parameter must have an associated ***generic formal subprogram parameter*** importing the `Swap` operation for that type. Other generic formal subprograms may be used to place constraints on imported components. Ada's subprogram default box specification `(<>)` is used with all generic formal subprograms.

The name of a kernel abstract component is the name of the concept being described prefixed with `AT_` or `AI_` depending on whether it is an abstract template or an abstract instance. For example, `AT_Queue` is the package name of a kernel abstract component describing a queue. The name of the exported type is usually the name of the concept being described. For example, the type exported from `AT_Queue` would be named `Queue`. In some cases, the type name may be shortened to improve the readability of the code. Note that an abstract template only requires a package specification and no associated package body. Figure 1.2 shows the graphical depiction of a RESOLVE/Ada95 kernel abstract component. This notation is used in component instantiation diagrams.

A component coupling diagram shows the dependencies between components which are independent of any particular use of the components. Component coupling diagrams help express how a group of related components, perhaps all contained in a component library, *may* be combined in many *possible* systems. In contrast, a ***component instantiation diagram*** shows how the components in a *particular* system are coupled with each other. Figure 1.2 shows the notation used for a kernel abstract component in a RESOLVE/Ada95-specific component instantiation diagram. As with the notation shown in Figure 1.1, an abstract component is depicted with a clear rounded box. Specification parameters are shown as arrowheads along the top of the box. In this example, `Item` is the only specification parameter (the generic subprogram parameter for `Swap` is implied). The type exported from a component is shown as an arrowhead on the right side of the box. A more detailed version of this notation (not shown in this report) includes the names of each exported operation in the lower right corner of the box. In this example, the abstract type `Queue` is the exported type. Component instantiation diagrams in subsequent sections will build upon this simple diagram.

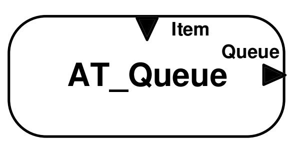

Figure 1.2 — A Queue Abstract Template

1.3. An Example: AT_Queue

The listing below shows the RESOLVE/Ada95 code for the `AT_Queue` kernel abstract component. The only aspect of this code not described in Section 1.2 is the design choice of which operations the queue *kernel* component should provide.

This choice of operations for `AT_Queue` reflects the RESOLVE design principle that the interface of a kernel component should be as simple as possible. While other queue operations will inevitably be useful in various contexts, these operations may be added as extensions and implemented with the use of the operations provided by the kernel component. Section 3 discusses component extensions. The component instantiation diagram shown in Figure 1.2 summarizes the following code.

Package Specification for AT_Queue

```
-----------------------------------------------------------------------------
--
-- Component: AT_Queue
-- Relations: -
--  Comments: -
--
-----------------------------------------------------------------------------
-- Global Context -----------------------------------------------------------
-----------------------------------------------------------------------------

with Ada.Finalization;

-----------------------------------------------------------------------------

generic

    -------------------------------------------------------------------------
    -- Specification Parameters ---------------------------------------------
    -------------------------------------------------------------------------

    type Item is limited private;
    with procedure Swap (left, right : in out Item) is <>;

    -------------------------------------------------------------------------

package AT_Queue is

    -------------------------------------------------------------------------
    -- Interface ------------------------------------------------------------
    -------------------------------------------------------------------------

    type Queue is abstract tagged limited private;

    --! type Queue is modeled by string of Item
    --!     exemplar q
    --!     initialization ensures
```

```
    --!          q = empty_string

    ----------------------------------------------------------------------

    procedure Enqueue (
            q : in out Queue;                   --! alters q
            x : in out Item                     --! consumes x
        ) is abstract;

    --! ensures
    --!     q = #q * <#x>

    ----------------------------------------------------------------------

    procedure Dequeue (
            q : in out Queue;                   --! alters q
            x : in out Item                     --! produces x
        ) is abstract;

    --! requires
    --!     q /= empty_string
    --! ensures
    --!     #q = <x> * q

    ----------------------------------------------------------------------

    procedure Get_Length (
            q      : in out Queue;              --! preserves q
            length : in out Integer             --! produces length
        ) is abstract;

    --! ensures
    --!     length = |q|

    ----------------------------------------------------------------------

    procedure Swap (
            left  : in out Queue;               --! alters left
            right : in out Queue                --! alters right
        ) is abstract;

    --! ensures
    --!     left = #right and right = #left

    ----------------------------------------------------------------------

private

    type Queue is abstract new Ada.Finalization.Limited_Controlled
        with null record;

    ----------------------------------------------------------------------

end AT_Queue;
```

2. Kernel Concrete Components

A concrete software component provides a description of the implementation of a subsystem. Like an abstract component, a concrete component describes the structural interface (type names and operation signatures) and the functional behavior of a subsystem. However, unlike an abstract component, a concrete component describes functional behavior in terms of a specific implementation. The meaning of the behavioral description is based on the semantics of the implementation language and the behavior of other components upon which the description depends. In RESOLVE/Ada95, concrete components are subsystem implementations. This section describes the most basic form of implementation, a kernel concrete component. Sections 3, 4, and 5 describe other types of concrete components.

In order to understand the RESOLVE/Ada95 encoding of concrete components, it is important to understand the relationships a concrete component has with other components. This section begins with two sections that discuss the software component relationships. These component relationships are directly encoded in kernel concrete components.

2.1. The *Implements* Relationship

Section 3.3 briefly introduced the idea of ***component relationships***. Component relationships are concise expressions of the *dependencies* which exist between components. The principle component relationship upon which all other component relationships rely for meaning and utility is ***implements***. *Implements* describes the relationship between an implementation, a concrete component, and a specification, an abstract component. The *implements* relationship may be informally defined as follows:

> Concrete component X *implements* abstract component Y if and only if X exhibits the behavior specified by Y.

This is a fairly intuitive relationship between a specification and an implementation. However, a fully formal and general definition of the *implements* relationship is very intricate and has been the subject of much research. A formal description of the *implements* relationship is beyond the scope of this report.

Implements expresses a dependency between two components in the following sense. If component X *implements* component Y, then X depends on Y to provide an abstract, client-level description of its behavior.

As an example, consider an abstract component which describes the behavior of a queue and a concrete component which describes an implementation of a queue. If the behavior provided by the concrete component can legitimately be described by the abstract component, then the concrete component *implements* that abstract component. This relationship is graphically depicted in the component coupling diagram shown in Figure 2.1. The abstract component `AT_Queue` was introduced in Section 1. Details of the concrete component AT_Queue.CT_2 and an explanation of its odd name are provided in Section 2.3. Note that in the RESOLVE/Ada95 version of component coupling diagrams, clear rounded boxes depict abstract components and shaded rectangular boxes depict concrete components.

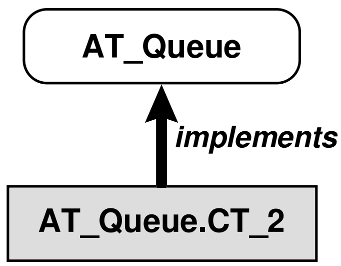

Figure 2.1 — The *Implements* Relationship

The leverage gained from establishing the *implements* relationship between an implementation and a specification has to do with the ***substitutability*** of implementations. If two different concrete components both implement the same abstract component, then either of them may be used in a context requiring the functional behavior described by the abstract component. In this case, the two implementations are ***behaviorally substitutable*** with respect to the specification they both implement.

To illustrate the concept of behavioral substitutability, consider the four components depicted in Figure 2.2. Concrete components `AT_Queue.CT_2` and `AT_Queue.CT_3` are two different implementations which provide the behavior described by `AT_Queue`. They provide only that behavior. `AT_Queue.With_Reverse.CT_1` also provides the behavior described by `AT_Queue`, but it provides an additional operation that reverses the order of items in a queue. (This component will be described in Section 3.) Since each of the three concrete components *implements* `AT_Queue`, all three are behaviorally substitutable with respect to `AT_Queue`. An application requiring the behavior specified by `AT_Queue` (an instance of `AT_Queue`) could use any of these three implementations to satisfy the requirement.

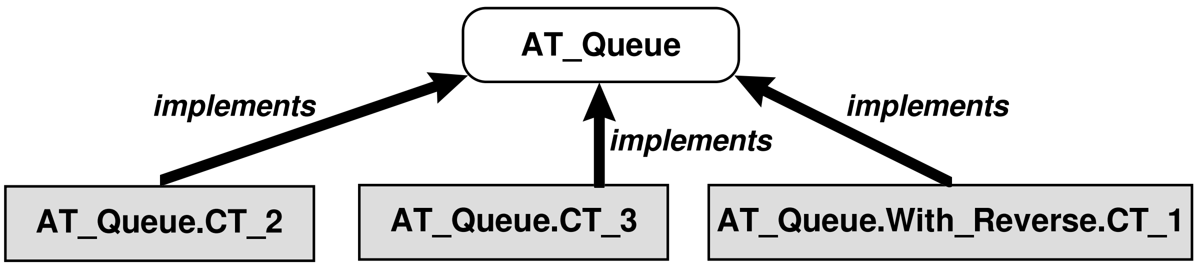

Figure 2.2 — Behaviorally Substitutable Implementations

Although `AT_Queue.CT_2` and `AT_Queue.CT_3` must provide the same functional behavior, they may differ in non-functional ways. For example, `AT_Queue.CT_3` may be a "new and improved" queue implementation with operations that run faster and require less memory than those of `AT_Queue.CT_2`. Significant improvements on a component as simple as a queue seem unlikely, but improving on the performance of more complex components is a common maintenance task. Another possibility is that these two implementations reflect a performance trade-off. Perhaps `AT_Queue.CT_2` provides `Enqueue` and `Dequeue` operations requiring comparable execution time while `AT_Queue.CT_3` provides a relatively fast `Enqueue` but slow `Dequeue`. With this situation, some applications might attain better performance using `AT_Queue.CT_2` while other applications might perform better using `AT_Queue.CT3`. Useful performance tradeoffs between implementations are also more likely to exist with components more complex than a queue. Eventually, non-functional differences such as availability, cost, warranty, legal use restrictions, and the level of trust in correctness might be factors in choosing among behaviorally substitutable components.

`AT_Queue.With_Reverse.CT_1` provides additional behavior that is not required in contexts that only need the behavior described by `AT_Queue`.

Nevertheless its *implements* relationship with `AT_Queue` might be very useful. Consider an application which requires the behavior specified by `AT_Queue` and uses `AT_Queue.CT_2` to supply that behavior. Now assume that the requirements for this application change and that it needs the capability to reverse a queue. Since `AT_Queue.With_Reverse.CT_1` *implements* `AT_Queue`, a maintainer may safely substitute the `AT_Queue.With_Reverse.CT_1` component for the `AT_Queue.CT_2` component. Any components of the application that used `AT_Queue.CT_2` and still do not require the new reverse capability will behave just as before the substitution. These components will require no modification. Only components using the new reverse operation will need to be modified or replaced. Note that `AT_Queue.CT_2` is *not* behaviorally substitutable for `AT_Queue.With_Reverse.CT_1` with respect to an abstract template that includes the queue reverse operation.

2.2. The *Uses* Relationship

The *implements* relationship describes a dependency between an implementation and a specification of a single abstraction. The ***uses*** relationship may describe a dependency that exists between two different abstractions. The relation name *uses* actually describes three different yet very closely related component relationships. *Uses* may describe a dependency between two implementations, between an implementation and a specification, or between two specifications. These three relationships are defined as follows:

> Concrete component X *uses* concrete component Y if and only if X depends directly on the implementation Y.
>
> Concrete component X *uses* abstract component Y if and only if X depends on the behavior specified by Y.
>
> Abstract component X *uses* abstract component Y if and only if X depends on the behavior specified by Y.

While worded differently, the second definition above is equivalent to the definition of *uses* given in Section 3.3.3. Note that none of these *uses* relationships is equivalent to the *is-a-part-of*, *has-a*, or *aggregation* relationship of other disciplines. If implementation X *uses* implementation Y, Y may or may not be part of the data representation of X. The client wishing to use X does not need to know Y's specific role in X, just that component Y is required to use component X. For instance, component X may require component Y to describe the implementation of its interface but not its data representation.

The *uses* relationship between two implementations should be familiar to anyone who has developed software. For example, if one program module calls a routine implemented in another program module, then the former module *uses* the latter. In Ada this relationship is expressed by the `with` context clause. In C and C++, this relationship is expressed with the `#include` directive. Figure 2.3 depicts this relationship in a component coupling diagram. This figure expresses the dependency of the implementation `AT_Queue.CT_2` on the implementation `AT_One_Way_List.CT_1`. `AT_Queue.CT_2`'s description of a queue implementation makes direct use of `AT_One_Way_List.CT_1`'s description of a list implementation. A ***fixed dependency*** exists between these two components.

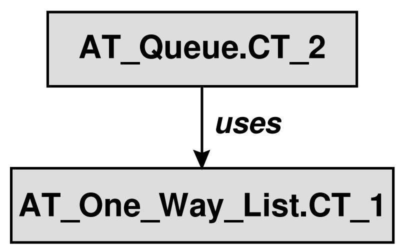

Figure 2.3 — The *Uses* Relationship Between Concrete Components

In general, software engineers want to avoid any unnecessary dependencies (coupling) between components. The *uses* relationship between an implementation and a specification may be applied to achieve this goal. Figure 2.4 depicts the *uses* relationship between an implementation and a specification. This figure expresses the dependency of the implementation `AT_Queue.CT_2` on the specification `AT_One_Way_List`. In this case the description of a queue implementation only depends on the abstract description of a list. During program execution, an object declared from an instance of `AT_Queue.CT_2` will indeed depend on a specific implementation of an instance of `AT_One_Way_List`. Fully understanding the non-functional characteristics of this object would include understanding non-functional characteristics of the specific list implementation used. However, to understand and reason about the functional behavior described by `AT_Queue.CT_2`, it is sufficient to understand the behavior described by `AT_One_Way_List` and `AT_Queue`.

The RESOLVE discipline requires that components be designed and built applying this second form of *uses* instead of the first form most of the time. (The third abstract to abstract form of *uses* is similar to the second and should be applied when one specification depends on another.) The tremendous benefit achieved from building component-based software this way is that it supports the property of ***modular reasoning***. Assume `AT_Queue.CT_2` *implements* `AT_Queue` and *uses* (only) `AT_One_Way_List` as depicted in Figures 2.1 and 2.4. Then it is possible to reason about the ***functional correctness*** of `AT_Queue.CT_2` with respect to its specification `AT_Queue` by only looking at the three components in Figures 2.1 and 2.4.

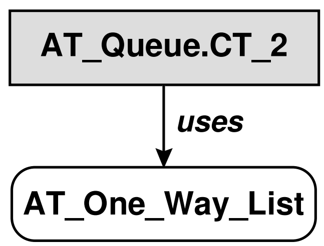

Figure 2.4 — The Preferred *Uses* Relationship

The *uses* relationship between an implementation and a specification expresses a ***polymorphic*** relationship. Many different "forms" or implementations of the abstract component operations may be used by the objects declared from instances of the concrete component. There are two primary strategies for encoding this polymorphism. Object-oriented disciplines typically use ***inheritance*** and ***dynamic binding*** to achieve it. RESOLVE uses ***parameterization*** and ***static binding***. With static binding, the actual implementations that a concrete component *uses* must be fixed prior to program execution time. With dynamic binding, implementations may be selected during program execution. While the RESOLVE/C++ discipline includes special components which support dynamic binding, the RESOLVE/Ada95 discipline as described in this report only supports static binding of implementations. At this point it is unclear if the increased flexibility provided by dynamic binding justifies the increased complexity that support for dynamic binding entails.

2.3. Encoding Kernel Concrete Components

RESOLVE/Ada95 kernel concrete components are encoded in Ada as ***generic child unit*** packages. A kernel concrete component is a generic child unit of the abstract kernel component that it *implements*. In general, it might be possible for a single concrete component to implement more than one abstract component. In RESOLVE/Ada95, however, each implementation is permanently linked to a single specification that it *implements*. Thus an *implements* relationship is encoded directly into each concrete component in RESOLVE/Ada95. Limiting each concrete template to a single *implements* relationship appears to sacrifice little and results in simpler RESOLVE/Ada95 code.

Child units are a part of the ***hierarchical library*** structure added to Ada in the 1995 language definition. A child unit is an Ada package that has full visibility to the package specification of another unit, its parent unit. The association of a child unit to its parent is expressed by the name of the child unit. A child unit package name has as a prefix its parent unit's package name followed by a period. Thus the package `AT_Queue.CT_2` is a child unit of the package `AT_Queue`. As a child unit, `AT_Queue.CT_2` has implicit visibility to all of the `AT_Queue` package specification, including generic formal parameters and the private part. RESOLVE/Ada95 concrete component names encode the *implements* relationship. For example, the package name `AT_Queue.CT_2` identifies the "second" concrete template that *implements* the queue abstract template. (The name is best interpreted from right to left.) By convention, numbers are used to distinguish concrete templates in order to avoid long and possibly misleading component names.

Figure 2.5 shows the RESOLVE/Ada95-specific component instantiation diagram depicting the parent-child relationship between a kernel abstract component and a kernel concrete component. Note that this parent-child relationship is *not* a component relationship; it is merely one possible technique useful for expressing the *implements* relationship in Ada. In this example, the abstract parent unit `AT_Queue` (first shown in Figure 1.2) is enclosed in the concrete child unit `AT_Queue.CT_2`. This notation conveys the idea that the definition of the child unit "includes" that of the parent unit. As with component coupling diagrams, concrete components are depicted with shaded rectangular boxes.

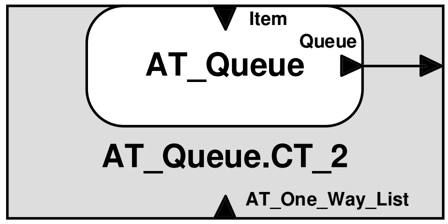

Figure 2.5 — Abstract Parent and Concrete Child Implementation

A RESOLVE/Ada95 kernel concrete component exports a concrete type and concrete primitive operations of that type. The concrete type exported is a ***private type extension*** of the abstract type exported by its parent unit. The type extension forms an inheritance link between the concrete and abstract types. This link is depicted in Figure 2.5 as the line connecting the abstract type exported from `AT_Queue` to the concrete type exported from `AT_Queue.CT_2` (both types are named `Queue`). This inheritance link requires the child unit implementation to provide concrete primitive operations with conforming signatures for each of the abstract type's abstract primitive operations. Therefore, an Ada compiler will ensure that an implementation at least provides the structural interface described by the specification that it *implements*. But like most programming languages, Ada alone provides no support for ensuring that an implementation achieves the behavior described by its specification. The formal semantics, proof rules, and specification sub-language of the RESOLVE language do provide support for formally verifying the *implements* relationship.

Unlike a kernel abstract component, a kernel concrete component requires both a package specification and a package body. The package specification includes a public ***interface section*** in which the exported type is derived from the abstract type exported by the parent unit. Placing this type derivation in the public section provides a ***partial view*** of the exported type. The partial view provides a client of the implementation visibility to the exported concrete operations, but not to the type's private data representation. The interface section also includes a concrete subprogram signature corresponding to each abstract operation in the parent unit. The package body includes all of the corresponding subprogram bodies. The package body may also include a ***local operations section*** for local subprograms used internally to implement the exported operations.

The private part of the package specification includes a ***representation section*** and an ***implicit operations section***. The representation section defines the data structure used to maintain state information for each object declared from the concrete component. The ***full view*** of the exported type is a single field ***record extension*** of the abstract type in the parent unit. To ensure composability of components, the single-record extension field is always named `rep`. The type of the `rep` field is the type of the component's data representation. If the representation has more than one constituent component, the components are encapsulated in a single RESOLVE/Ada95 `Record`*n* component which serves as the type of `rep`. `Record`*n* components are special RESOLVE/Ada95 components that export `Swap` and field selection operations. `Record`*n* components compose with other RESOLVE/Ada95 components and are responsible for encapsulating various memory management strategies. Formal comments in the representation section describe the implementation's ***convention*** and ***correspondence***. The convention specifies any ***representation invariants*** and the correspondence describes the ***abstraction relation*** between the representation and its mathematical model.

The implicit operations section is where the inherited `Initialize` and `Finalize` operations may be overridden, if necessary. The exported concrete type inherits concrete null-bodied implementations of these operations from its abstract parent (which inherits them from `Limited_Controlled`). For typical components, the inherited `Initialize` and `Finalize` will provide the correct behavior. However, some components will need to explicitly override these operations, especially `Initialize`.

Since a kernel concrete component is a child unit of a kernel abstract concept, it has direct visibility to any specification parameters. The specification parameters are essentially implicit parameters of the kernel concrete component. This idea is conveyed in Figure 2.5 since the specification parameter `Item` on the top edge of `AT_Queue` is also on the top edge of `AT_Queue.CT_2`. In addition to implicit specification parameters, a kernel concrete component may also have additional parameters which, if present, are included in the ***implementation parameters section***. Implementation parameters are encoded in RESOLVE/Ada95 as generic formal packages, types, and subprograms for which actual parameters must be supplied through instantiation. Component instantiation diagrams depict implementation parameters as arrowheads along the bottom edge of concrete components. For example, `AT_One_Way_List` is shown as an implementation parameter in Figure 2.5.

The *uses* relationship between a concrete component and an abstract component is encoded in RESOLVE/Ada95 using a pair of related generic formal parameters and a `with` clause in the global context section. The `with` clause names the abstract component upon which the implementation depends. The first generic parameter is a ***generic formal package parameter*** and the second is a ***generic formal derived type***. Both of these generic parameters are new language mechanisms added to Ada in the 1995 language definition. The actual parameter corresponding to the formal package parameter must be an instance of the abstract package component being used. The actual parameter corresponding to the formal derived type must be a tagged type derived from the abstract type exported by the actual package parameter. Together these two parameters ensure that the imported type has all of the primitive operations specified in the abstract package. In other words, the imported concrete type is constrained to have been exported from a concrete component which *implements* the abstract component being used. Of course, only the structural aspects of the *implements* relationship are enforced by the compiler.

The example code in Section 2.4 demonstrates how the *uses* relationship depicted in Figure 2.4 is encoded in RESOLVE/Ada95. A `with` clause in the global context section expresses the fixed dependency of `AT_Queue.CT_2` on the abstract component `AT_One_Way_List`. The first generic parameter in the implementation parameters section is the formal package `AI_One_Way_List`. The actual parameter corresponding to this formal must be a package which is an instance of `AT_One_Way_List`. Furthermore, this instance must have been instantiated with the same `Item` type as the instance of `AT_Queue` which serves as the parent of an instance of `AT_Queue.CT_2`. (References to `AT_Queue` within `AT_Queue.CT_2` implicitly refer to an *instance* of `AT_Queue`.) The second generic parameter is the formal derived type `List`. The actual parameter corresponding to `List` must be a concrete type derived (possibly indirectly) from the abstract `List` type exported by the package serving as the actual parameter to `AI_One_Way_List`. Note that in a component instantiation diagram, a single implementation parameter corresponds to the pair of generic parameters required to express the *uses* relationship. For example, in Figure 2.5, the single implementation parameter `AT_One_Way_List` expresses the *uses* relationship just described.

At the programming language level, this encoding of *uses* ensures that the imported List type comes with implementations for all of the operations specified in `AT_One_Way_List`. In terms of component relationships, this strategy encodes the dependency that `AT_Queue.CT_2` has on the behavior described by `AT_One_Way_List` without making `AT_Queue.CT_2` dependent on a *specific* implementation of `AT_One_Way_List`.

2.4. An Example: AT_Queue.CT_2

This section includes the RESOLVE/Ada95 code for the kernel concrete component `AT_Queue.CT_2`. This component is a concrete template which *implements* the abstract template `AT_Queue`. This implementation *uses* the abstract template `AT_One_Way_List` which describes the interface and behavior of a list subsystem. (A `One_Way_List` may only be traversed in one direction, like a singly linked list.) This implementation uses the imported list as the data representation for the queue. The `AT_Queue.CT_2` implementation consists of a package specification and a package body. The first listing that follows is the package specification. The second listing is the package body. The component instantiation diagram shown in Figure 2.5 summarizes the following code.

Package Specification for AT_Queue.CT_2

```
---------------------------------------------------------------------------
--
-- Component: AT_Queue.CT_2
-- Relations: implements AT_Queue, uses AT_One_Way_List
--  Comments: queue implemented with a One_Way_List representation
--
---------------------------------------------------------------------------
-- Global Context ---------------------------------------------------------
---------------------------------------------------------------------------

with AT_One_Way_List;

---------------------------------------------------------------------------

generic

    -----------------------------------------------------------------------
    -- Implementation Parameters ------------------------------------------
    -----------------------------------------------------------------------

    with package AI_One_Way_List is new AT_One_Way_List (Item => Item);

    type List is new AI_One_Way_List.List with private;

    -----------------------------------------------------------------------

package AT_Queue.CT_2 is

    -----------------------------------------------------------------------
    -- Interface ----------------------------------------------------------
    -----------------------------------------------------------------------

    type Queue is new AT_Queue.Queue with private;
```

```
    --------------------------------------------------------------------------

    procedure Enqueue (
            q : in out Queue;
            x : in out Item
        );

    --------------------------------------------------------------------------

    procedure Dequeue (
            q : in out Queue;
            x : in out Item
        );

    --------------------------------------------------------------------------

    procedure Get_Length (
            q      : in out Queue;
            length : in out Integer
        );

    --------------------------------------------------------------------------

    procedure Swap (
            left  : in out Queue;
            right : in out Queue
        );

    --------------------------------------------------------------------------

private

    --------------------------------------------------------------------------
    -- Representation --------------------------------------------------------
    --------------------------------------------------------------------------

    type Queue is new AT_Queue.Queue with
        record
            rep : List;
        end record;

    --! convention
    --!     true
    --! correspondence
    --!     q = q.rep.left * q.rep.right

    --------------------------------------------------------------------------
    -- Implicit Operations ---------------------------------------------------
    --------------------------------------------------------------------------

    -- Inherited null-bodied Initialize and Finalize

    --------------------------------------------------------------------------

end AT_Queue.CT_2;
```

Package Body for AT_Queue.CT_2

```
-----------------------------------------------------------------------------
--
-- Component: AT_Queue.CT_2
-- Relations: implements AT_Queue, uses AT_One_Way_List
--  Comments: queue implemented with a One_Way_List representation
--
-----------------------------------------------------------------------------
-- Global Context -----------------------------------------------------------
-----------------------------------------------------------------------------

-----------------------------------------------------------------------------

package body AT_Queue.CT_2 is

    -------------------------------------------------------------------------
    -- Implicit Operations --------------------------------------------------
    -------------------------------------------------------------------------

    -- Inherited null-bodied Initialize and Finalize

    -------------------------------------------------------------------------
    -- Interface Operations -------------------------------------------------
    -------------------------------------------------------------------------

   procedure Enqueue (
            q : in out Queue;
            x : in out Item
        ) is
    begin
        Move_To_Finish (q.rep);
        Add_Right (q.rep, x);
    end Enqueue;

    -------------------------------------------------------------------------

    procedure Dequeue (
            q : in out Queue;
            x : in out Item
        ) is
    begin
        Move_To_Start (q.rep);
        Remove_Right (q.rep, x);
    end Dequeue;

    -------------------------------------------------------------------------

    procedure Get_Length (
            q      : in out Queue;
            length : in out Integer
        ) is
    begin
        Move_To_Start (q.rep);
        Get_Right_Length (q.rep, length);
    end Get_Length;
```

```
    ----------------------------------------------------------------------------

    procedure Swap (
           left  : in out Queue;
           right : in out Queue
   ) is
    begin
        Swap (left.rep, right.rep);
    end Swap;

    ----------------------------------------------------------------------------

end AT_Queue.CT_2;
```

3. Extension Components

Sections 1 and 2 describe RESOLVE/Ada95 abstract and concrete kernel components used to specify and implement software subsystems. Once a software system has been designed and implemented, changes to requirements are likely to call for improved performance and enhanced functionality. As Section 2.1 explains, improvements in performance which do not alter functionality may be addressed by replacing one concrete component with another. This section introduces a new software component relationship and describes how to add new functionality to existing RESOLVE/Ada95 components.

3.1. The *Extends* Relationship

If component designers had perfect foresight they might be tempted to provide kernel components with all the functionality that clients could ever want. If this were possible, then the *implements* and *uses* relationships discussed in Section 2 might be sufficient for describing component dependencies. Unfortunately component designers do not have perfect foresight. No matter how much forethought designers apply, new requirements almost always expose some desirable and unforeseen functionality. Clearly some method for extending the functionality of kernel components is needed.

Lack of perfect foresight, however, is not the only reason for providing a means to extend the functionality of components. Some disciplines for designing components advocate providing a wide assortment of possibly useful operations in their equivalent of kernel components. The problem with this approach is that the resulting component interfaces are more complex and thus more difficult for clients and component implementers to understand. Furthermore, this approach either leads to code bloat resulting from many unused operations or requires assumptions about optimizations which attempt to expunge code of unused operations.

The approach used by the RESOLVE discipline is to carefully design kernel components with a minimally sufficient set of ***primary operations***. New functionality is then added (usually one operation at a time) with components which extend the behavior of existing components. The ideal choice of primary operations for a kernel component is seldom obvious. One possible approach is to design kernel components with no operations (just an exported type) and then to add *all* operations as extensions. The RESOLVE/Ada95 discipline currently takes the more conventional approach of including primary operations with the kernel component.

The ***extends*** relationship expresses a dependency between two components, one of which extends the functionality of the other. Like *uses*, the name *extends* is overloaded. *Extends* applies to two different, yet closely related, component relationships. One *extends* expresses a dependency between two abstract components and the other expresses a dependency between two concrete components. Both *extends* relationships may be defined informally as follows:

> Component X *extends* component Y if and only if all of the interface and behavior described by Y is included in the interface and behavior described by X.

This definition conveys the intuitive meaning of *extends*, that is, component X extends the interface and behavior of component Y. While the definition leaves open the possibility that X may add nothing to Y, in general, component X will include a description of functionality not present in Y. Another way of characterizing the *extends* relationships is in terms of the *implements* and *uses* relationships:

> Abstract component X *extends* abstract component Y if and only if any component which *implements* X also *implements* Y.
>
> Concrete component X *extends* concrete component Y if and only if X *uses* Y and re-exports all the interface and behavior of Y.

These definitions convey the essential property of behavioral substitutability discussed in Section 2.1. If abstract component X *extends* abstract component Y, concrete component X1 *implements* X, and concrete component Y1 *implements* Y, then X1 is behaviorally substitutable for Y1 with respect to Y. Note that Y1 is *not* behaviorally substitutable for X1 with respect to X .

Figure 3.1 is a component coupling diagram which depicts the *extends* relationship between two components: `AT_Queue.With_Reverse` *extends* `AT_Queue`. This means that `AT_Queue.With_Reverse` describes the same interface and behavior as `AT_Queue` plus some additional interface and behavior. In this case, the additional functionality is that of an operation which reverses the order of items in the queue. The mathematical model describing a queue and the behavioral descriptions of operations to enqueue and dequeue items are provided by `AT_Queue.With_Reverse` just as they are by `AT_Queue`. `AT_Queue.With_Reverse` additionally includes the signature of an operation to reverse a queue and a formal description of its behavior specified in terms of the queue's mathematical model.

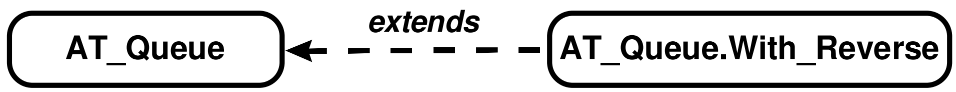

Figure 3.1 — The *Extends* Relationship Between Abstract Components

To readers familiar with object-oriented techniques, the *extends* relationship may look very much like an inheritance relationship. It is very important to understand that *extends* is *not* an inheritance relationship. Object-oriented analysis and design methodologies use inheritance as a design relationship which conveys many different meanings depending on how it is used and who is using it. The *extends* relationship has the very precise meaning given above. The inheritance *design relationship* is often equated with the ***is-a*** relationship. Although there is no single agreed-upon and formally defined meaning of *is-a*, many in the object-oriented community do associate this relationship with behavioral substitutability. Even so, *is-a* does not convey the specific dependency relationships and component usage intent conveyed by *implements* and *extends*. In fact, both *implements* and *extends* as well as the *specializes* and *checks* relationships described in the next two sections could be interpreted as special cases of the *is-a* relationship.

The inheritance *programming language mechanism* happens to be a convenient tool for expressing the *extends* relationship in an object-oriented language such as Ada or C++. However, *extends* may be encoded without the use of inheritance. For example, the abstract template `AT_Queue.With_Reverse` could explicitly include a description identical to that of `AT_Queue`. While duplication of a specification such as this may not be ideal from a maintenance perspective, it demonstrates the point that *extends* and "*inherits from*" are not the same.

The component coupling diagram in Figure 3.2 adds two implementations to the relationship shown in Figure 3.1. The *implements* relationship on the left side of Figure 3.2 is the same relationship shown in Figure 2.1 and discussed in Section 2.1. The *implements* relationship on the right side of Figure 3.2 is a relationship between an abstract extension component and the implementation of the abstract component. While this use of *implements* was not explicitly discussed in Section 2, it is the same relationship and expresses the same meaning as all other uses of *implements*. The concrete component `AT_Queue.With_Reverse.CT_1` provides the interface and behavior described abstractly by `AT_Queue.With_Reverse`. Although not shown explicitly in Figure 3.2, the *extends* and right side *implements* relationships imply that `AT_Queue.With_Reverse.CT_1` also *implements* `AT_Queue` as shown in Figure 2.2. `AT_Queue.With_Reverse.CT_1` is therefore behaviorally substitutable for `AT_Queue.CT_2` with respect to `AT_Queue`.

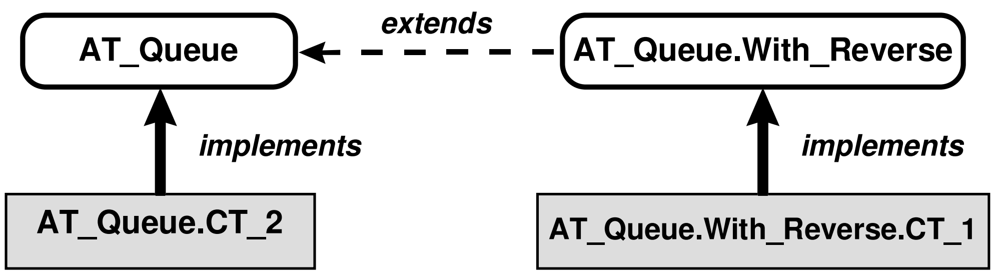

Figure 3.2 — Direct Implementation of an Extension

If component coupling diagrams explicitly depicted all derivable relationships between components, the diagrams would become very complex and difficult to read. One of the benefits of precisely defining these component relationships is that relatively simple component coupling diagrams may express a lot of precise and useful information. The graphical depictions of software component relationships used by other design and implementation methodologies do not appear to convey information as precisely defined as that expressed by these diagrams. Furthermore, component relationships intended to express implementation-level information in other methodologies are usually programming language-specific and express little or no useful design-level information. The component relationships described in this report are language-independent concepts that convey design-level intent and dependency information useful to system implementers and maintainers.

Return to Figure 3.2 to consider how `AT_Queue.With_Reverse.CT_1` might be implemented. There are at least three ways in which this implementation might be constructed: ***direct implementation***, ***coupled implementation***, and ***layered implementation***. A direct implementation does not depend on any other implementations of the abstract concept being extended. Thus a direct implementation would require that `AT_Queue.With_Reverse.CT_1` implement the interface and behavior described by `AT_Queue` in addition to providing an implementation of the additional behavior and interface described by `AT_Queue.With_Reverse`. While there might be a legitimate justification for taking this approach, in general this would be a poor choice since the new functionality (to reverse a queue) cannot be used with other implementations of `AT_Queue`. Furthermore, this is costly since it does not benefit from reuse of existing implementations of `AT_Queue`. Since Figure 3.2 does not show any relationship between these two components, direct implementation must be assumed.

Figure 3.3 is the same as Figure 3.2 except that it also includes a concrete to concrete *extends* relationship from `AT_Queue.With_Reverse.CT_1` to `AT_Queue.CT_2`. Figure 3.3 depicts either a coupled or layered implementation of `AT_Queue.With_Reverse.CT_1`, but does not distinguish between the two. A slightly more complicated notation which explicitly depicts template parameters does distinguish between these two possibilities. In Figure 3.3, the implementation on the right side *uses* the implementation on the left side to provide implementations for all operations described in `AT_Queue`. Therefore, the component `AT_Queue.With_Reverse.CT_1` only explicitly needs to provide the additional implementation for the queue reverse operation.

Coupled implementations require that the extension implementation have full access to the implementation being extended. This includes direct access to the *representation* of the implementation being extended. As noted, the *extends* relationship alone does *not* imply access to the representation. If Figure 3.3 depicts a coupled implementation of `AT_Queue.With_Reverse.CT_1`, then the implementation of the queue reverse operation may be expressed directly in terms of the queue representation implemented in `AT_Queue.CT_2`. The advantage of coupled implementations over direct implementations is that they reuse an existing implementation and thus should be less expensive to construct. However, like direct implementations, the added functionality may only be used with a single implementation. In this example, the queue reverse operation could only be applied to queues with the specific implementation described in `AT_Queue.CT_2`.

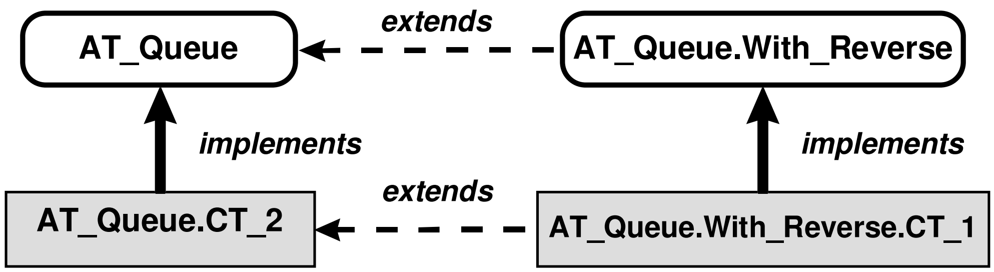

Figure 3.3 — A Coupled or Layered Implementation of an Extension

While both coupled and direct implementation of extensions might be justified for performance reasons, coupled implementations have additional disadvantages which direct implementations avoid. First of all, depending on the privileges granted by the owner of an implementation to the implementer of the extension, a coupled implementation may or may not be possible. When coupled implementations are possible, they require breaking the encapsulation of the implementation being extended. To ensure that an extension implementation is behaviorally substitutable for the implementation being extended requires careful implementation based on a detailed understanding of the implementation being extended. In particular, the implementer must ensure that the representation invariant and abstraction relation of the implementation being extended are respected by the extension implementation.[1]

The layered approach requires that new functionality be achieved using the operations provided by an existing implementation without access to its representation. Thus a layered implementation depends on the behavior described by the implementation to be extended, but not on the specifics of any implementation of that behavior. Assuming that Figure 3.3 depicts a layered implementation of `AT_Queue.With_Reverse.CT_1`, the queue reverse operation could be implemented using any of the operations described by `AT_Queue`. Any operations associated with concepts imported as implementation

---

1 A clear explanation of and solution to this problem is presented in the article “Representation Inheritance: A Safe Form of ‘White Box’ Code Inheritance” by Steven H. Edwards in *The Fourth International Conference on Software Reuse*, published by the IEEE Computer Society in April 1996.

parameters could also be used. The `AT_Queue.With_Reverse.CT_1` component shown in Section 3.5 imports a stack component as an implementation parameter to help realize the queue reverse operation.

The layered approach allows the new functionality of the extension to be used with *any* implementation of the abstract component being extended. This makes the resulting implementation much more flexible and easier to use in a wider variety of contexts. Furthermore, the layered approach provides the reuse benefits of coupled implementation without the serious problems described above. Finally, since layered implementation does not require re-implementing or understanding existing implementations, a layered implementation is likely to be the least costly to construct of the alternatives.[1] The one disadvantage of the layered approach is that layered operations may not be as efficient as those implemented using either the direct or coupled approaches. With carefully designed kernel components, however, the performance penalty can usually be kept low enough so that the advantages of layering usually far outweigh the disadvantages.

Layering implementations is the primary approach used to extend components built following the RESOLVE discipline. Like the concrete to abstract version of *uses*, the layered implementation version of *extends* conveys a polymorphic relationship. The polymorphism exists between operations named in the concrete extension component and implementations of those operations. As noted in Section 2.1, the RESOLVE discipline uses parameterization to express this polymorphism. This approach is also used in RESOLVE/Ada95.

3.2. Encoding Abstract Extension Components

Encoding both the abstract and concrete extension components using the language mechanisms of Ada presents challenges. For example, there is no completely satisfying way of using Ada to express all of the desirable relationships involved in all three approaches to implementing extension components. As noted in the preface, this report presents one of many possible strategies for applying the RESOLVE discipline to Ada. Variations on this strategy ultimately may be prove to be more effective for building large complex systems.

---

1 Empirical evidence supporting gains in productivity and software quality from using the layering approach is presented in the article “The Effects of Layering and Encapsulation on Software Development Cost and Quality” by Stuart H. Zweben and others in the March 1995 issue of *IEEE Transactions on Software Engineering*.

One of the difficulties encountered when attempting to encode component relationships in Ada involves the expression of multiple dependencies. For example, Ada's inheritance mechanism is useful for expressing both *implements* and *extends* relationships. However, Ada does not directly support multiple inheritance which has proven useful in RESOLVE/C++. Multiple inheritance is useful for expressing the simultaneous *implements* and *extends* relations of a layered or coupled extension implementation as depicted in Figure 3.3.

The current strategy used by RESOLVE/Ada95 extension components was developed primarily to support layered implementations. This strategy uses ***mixin inheritance*** to express multiple dependencies. Mixin inheritance is expressed in Ada by deriving an exported type from another type imported as a generic parameter. In order to apply multiple extensions to a component, use of mixin inheritance requires ***chaining*** extensions together to form a linear inheritance path. Furthermore, to support structural interface conformance of each concrete extension component to an abstract extension component requires an inheritance chain which alternates between abstract and concrete types. The encoding of this strategy in Ada does not reflect ideally the meaning of the *extends* relationship as described in Section 3.1. However, the effect achieved by RESOLVE/Ada95 does provide a reasonable level of language enforced structural conformance.

An abstract extension component is encoded in RESOLVE/Ada95 as a generic child unit of the abstract component it extends. By convention, the package name of the child unit is the name of the parent unit followed by a period, followed by the string `With_` followed by a name describing the new functionality. For example, the abstract component `AT_Queue.With_Reverse` *extends* the abstract component `AT_Queue` by a description of the queue reverse functionality. In general, it might be possible for a single abstract extension component to extend more than one abstract component. In RESOLVE/Ada95, however, each abstract extension is permanently linked to a single specification that it *extends*. Thus the *extends* relationship is encoded directly into each abstract extension component in RESOLVE/Ada95.

Figure 3.4 shows the RESOLVE/Ada95-specific component instantiation diagram depicting the parent-child relationship between a kernel abstract component and a abstract extension component. In this example, the abstract parent unit `AT_Queue` (first shown in Figure 1.2) is enclosed in `AT_Queue.With_Reverse`, the abstract child unit.

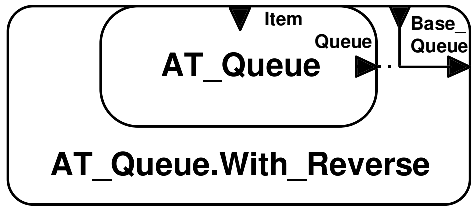

Figure 3.4 — Abstract Parent and Abstract Child Extension

The specification parameters section of an abstract extension component has a single specification parameter which serves to support chaining of extensions. This parameter is encoded as a formal generic derived type. The name of the formal parameter is the name of the exported type prefixed with the string `Base_`. The actual parameter corresponding to this formal parameter is constrained to be a *concrete* type derived from the abstract type exported by the abstract parent unit being extended. For example, `AT_Queue.With_Reverse` shown in Figure 3.4 has the specification parameter `Base_Queue`. The dotted line between the type exported from `AT_Queue` and the line extending from the specification parameter conveys the constraint on the actual parameter for `Base_Queue`. Note that the actual parameter need not be derived directly from the abstract type exported by (an instance of) the parent unit. The actual parameter may be the result of many extensions to that type. This flexibility allows multiple extensions of the same abstract component to be chained together.

The interface section of an abstract extension component includes an exported abstract type publicly derived from the formal generic type parameter. The name of the exported type is the same as the name of the type from which it is derived. (The exported type's full name includes its package name thus distinguishing it from its parent type). This derivation provides the mixin inheritance. Furthermore, the exported type is a null record extension of the type imported by the generic parameter. The null record extension means that the abstract extension component does not augment the representation of the imported type. In Figure 3.4, the line connecting the `Base_Queue` specification parameter to the abstract type exported by `AT_Queue.With_Reverse` depicts the mixin inheritance link. The abstract type exported by an instance of `AT_Queue.With_Reverse` has as primitive operations the abstract queue reverse operation specified in the extension plus all *concrete* operations of the type used as the actual parameter for `Base_Queue`.

The remainder of the interface section includes usually one, but possibly more, abstract subprogram specifications. Each subprogram specification corresponds to an operation providing new functionality to the abstract component being extended. As with all abstract components, structured comments formally describe the functional behavior of the abstract operations. Section 3.3 provides an example of a RESOLVE/Ada95 abstract extension component.

3.3. An Example: AT_Queue.With_Reverse

This section includes the RESOLVE/Ada95 package specification for the abstract extension component `AT_Queue.With_Reverse` (no package body is required). `AT_Queue.With_Reverse` is an abstract template which *extends* the abstract template `AT_Queue`. Normally the added operation to reverse a queue would be named `Reverse` instead of `Reverse_Queue`. However, since `reverse` is a reserved word in Ada, it cannot be used as an operation name. Also note that `reverse` is a built-in operation of the string theory mathematics in RESOLVE's specification sub-language. This explains how the ensures clause of `Reverse_Queue` may be expressed so concisely. The component instantiation diagram shown in Figure 3.4 summarizes the following code.

Package Specification for AT_Queue.With_Reverse

```
-----------------------------------------------------------------------------
--
-- Component: AT_Queue.With_Reverse
-- Relations: extends AT_Queue
--  Comments: 'Reverse' is an Ada reserved word, hence Reverse_Queue
--
-----------------------------------------------------------------------------
-- Global Context -----------------------------------------------------------
-----------------------------------------------------------------------------

-----------------------------------------------------------------------------

generic

    -------------------------------------------------------------------------
    -- Specification Parameters ---------------------------------------------
    -------------------------------------------------------------------------

    type Base_Queue is new AT_Queue.Queue with private;

    -------------------------------------------------------------------------

package AT_Queue.With_Reverse is

    -------------------------------------------------------------------------
    -- Interface ------------------------------------------------------------
    -------------------------------------------------------------------------

    type Queue is abstract new Base_Queue with null record;

    -------------------------------------------------------------------------
    -- Added Operations -----------------------------------------------------
    -------------------------------------------------------------------------

    procedure Reverse_Queue (
            q : in out Queue              --! preserves q
        ) is abstract;

    --! ensures
    --!     q = reverse(#q)

    -------------------------------------------------------------------------

end AT_Queue.With_Reverse;
```

3.4. Encoding Concrete Extension Components

This section and the example which follows in Section 3.5 only describe how RESOLVE/Ada95 provides layered implementations for concrete extension components. Currently, the RESOLVE/Ada95 encodings of direct and coupled implementations of extensions do not provide a language enforcement of the *implements* relationship. With abstract extension components encoded as described in Section 3.2, layered implementations do provide language enforcement of the structural compatibility expressed by the *implements* relationship.

The layered implementation of a concrete extension component is encoded in RESOLVE/Ada95 as a generic child unit of the component it *implements*. The prefix of child unit's package name is the name of the abstract extension component (the parent unit name) followed by a period. Just like implementations of kernel concepts, the name ends with the string `CT_` followed by a number identifying the specific implementation. For example, the concrete extension component name `AT_Queue.With_Reverse.CT_1`, denotes the "first" concrete template that *implements* the reverse component that *extends* the queue abstract template. Again, the component name is best interpreted from right to left.

Figure 3.5 shows the RESOLVE/Ada95-specific component instantiation diagram depicting the parent-child relationships between a kernel abstract component, an abstract extension component, and a layered concrete extension component. In this example, the abstract extension component `AT_Queue.With_Reverse` (first shown in Figure 3.4) is enclosed in `AT_Queue.With_Reverse.CT_1`, the concrete child unit.

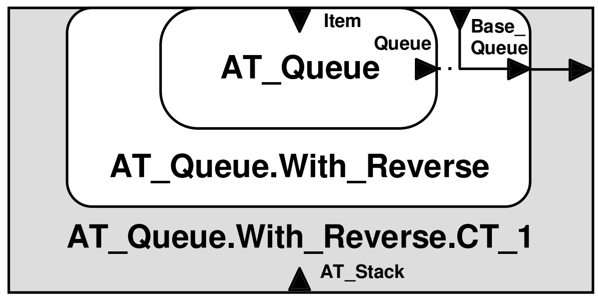

Figure 3.5 — Abstract Parent and Concrete Child Extension

A concrete extension component consists of a package specification and body which are similar in structure to those of a kernel concrete component. The package specification includes an interface section, a ***representation extension section***, and possibly an implementation parameters section. The interface section includes a concrete exported type and concrete subprogram specifications. Just as with kernel concrete components, the exported type is a private type extension of the abstract type exported by its parent unit. This inheritance link is shown in Figure 3.5 as the line connecting the abstract type exported from `AT_Queue.With_Reverse` to the concrete type exported by `AT_Queue.With_Reverse.CT_1`. In this example, both types are named `Queue` (as is the type exported by `AT_Queue`).

Derivation of the exported concrete type in the public part of the package provides a partial view of the type and ensures that the *concrete* operations imported by the parent unit are re-exported along with the concrete exported type. For example, the actual parameter for `Base_Queue` in Figure 3.5 may include many extensions to `AT_Queue`. However, the body of `AT_Queue.With_Reverse.CT_1` only has visibility to the operations described by `AT_Queue.With_Reverse` (which include those of `AT_Queue`).

The interface section also includes one concrete subprogram specification corresponding to each abstract operation added by the parent unit to describe new functionality. The representation extension section is in the private part of the package and contains the declaration of the exported type providing its full view. The exported type is a null record extension of the abstract type exported by the parent unit. The implementation parameters section, if present, serves the same role as in kernel concrete components as discussed in Section 2.3. Figure 3.5 shows the implementation parameter `AT_Stack`.

The package body of a concrete extension component includes one subprogram body corresponding to each added operation. Like its kernel component counterpart, it also may include a local operations section to provide local subprograms used to implement the exported operations.

3.5. An Example: AT_Queue.With_Reverse.CT_1

This section includes the RESOLVE/Ada95 package specification and body for the concrete extension component `AT_Queue.With_Reverse.CT_1`. This component is a concrete template which *implements* the abstract template `AT_Queue.With_Reverse` shown in Section 3.3. Since it is a layered implementation, the concrete component it *extends* must be provided as a parameter. The component instantiation diagram shown in Figure 3.5 summarizes the following code.

Package Specification for AT_Queue.With_Reverse.CT_1

```
----------------------------------------------------------------------------
--
-- Component: AT_Queue.With_Reverse.CT_1
-- Relations: implements AT_Queue.With_Reverse, uses AT_Stack
--  Comments: Reverse_Queue reverses a queue using a stack
--
----------------------------------------------------------------------------
-- Global Context ----------------------------------------------------------
----------------------------------------------------------------------------

with AT_Stack;

----------------------------------------------------------------------------

generic

    ------------------------------------------------------------------------
    -- Implementation Parameters -------------------------------------------
    ------------------------------------------------------------------------

    with package AI_Stack is new AT_Stack (Item => Item);

    type Stack is new AI_Stack.Stack with private;

    ------------------------------------------------------------------------

package AT_Queue.With_Reverse.CT_1 is

    ------------------------------------------------------------------------
    -- Interface -----------------------------------------------------------
    ------------------------------------------------------------------------

    type Queue is new AT_Queue.With_Reverse.Queue with private;

    ------------------------------------------------------------------------
    -- Added Operations ----------------------------------------------------
    ------------------------------------------------------------------------

    procedure Reverse_Queue (
            q : in out Queue
        );

    ------------------------------------------------------------------------

private

   ------------------------------------------------------------------------
   -- Representation Extension --------------------------------------------
   ------------------------------------------------------------------------

   type Queue is new AT_Queue.With_Reverse.Queue with null record;

   ------------------------------------------------------------------------
end AT_Queue.With_Reverse.CT_1;
```

Package Body for AT_Queue.With_Reverse.CT_1

```
-----------------------------------------------------------------------------
--
-- Component: AT_Queue.With_Reverse.CT_1
-- Relations: implements AT_Queue.With_Reverse, uses AT_Stack
--  Comments: Reverse_Queue reverses a queue using a stack
--
-----------------------------------------------------------------------------
-- Global Context -----------------------------------------------------------
-----------------------------------------------------------------------------

-----------------------------------------------------------------------------

package body AT_Queue.With_Reverse.CT_1 is

    -------------------------------------------------------------------------
    -- Added Operations -----------------------------------------------------
    -------------------------------------------------------------------------

    procedure Reverse_Queue (
          q : in out Queue
        ) is
          s      : Stack;
          x      : Item;
          length : Integer := 0;
    begin
        Get_Length (q, length);
        while length > 0 loop
            --! alters    q, s, length
            --! consumes  x
            --! maintains reverse(s) * q = reverse(#s) * #q  and  length = |q|
            --! decreases |q|
            Dequeue (q, x);
            Push (s, x);
            length := length - 1;
        end loop;
        Get_Length (s, length);
        while length > 0 loop
            --! alters    q, s, length
            --! consumes  x
            --! maintains q * s = #q * #s  and  length = |s|
            --! decreases |q|
            Pop (s, x);
            Enqueue (q, x);
            length := length - 1;
        end loop;
    end Reverse_Queue;

    -------------------------------------------------------------------------

end AT_Queue.With_Reverse.CT_1;
```

4. Specialization Components

The *implements*, *uses*, and *enhances* relationships described in Sections 2 and 3 express fundamental dependencies that all software component methodologies should address in some form or another. The *specializes* and *checks* relationships described in this and the next section are very useful for describing dependencies which may exist between components built using the RESOLVE framework. The reader unfamiliar with RESOLVE may find these two relationships, especially *checks*, new and interesting.

4.1. The *Specializes* Relationship

One of the principles of the RESOLVE discipline is that implementations should be ***fully parameterized***. This means that each dependency between a concrete component and a component it *uses* should be expressed in terms of the *abstract* component used instead of an implementation. Section 2.2 explains how this principle supports the ability to reason modularly about component-based software. Another benefit of this approach is that it supports the ability to parametrically adjust the performance characteristics of implementations. For example, the performance of the queue implementation AT_Queue.CT_2 shown in Section 2.4 depends on which `One_Way_List` implementation a client provides as an implementation parameter. The client of this component may ***tune*** the performance of a concrete component by choosing among different implementations of the components it *uses*.

The disadvantage of fully parameterized concrete components is that they are more difficult for clients to use. For example, most client programmers who want to use a queue component in their application will not want to be bothered with selecting a list implementation. (The list implementation might also require its own implementation parameters, and so on.) The RESOLVE framework neatly solves this problem with a ***specialization component***. A specialization component is a concrete component for which some and usually all of the *uses* relationships are fixed to specific implementations. It is produced by internally instantiating implementation parameters of a fully parameterized component and then re-exporting the interface and behavior of the resulting instantiation. A RESOLVE component library may contain several specialization components associated with each fully parameterized kernel concrete component.

The ***specializes*** relationship expresses a dependency between two components and is defined as follows:

> Component X *specializes* template component Y if and only if (a) X *uses* Y, (b) X re-exports the interface and behavior of Y, and (c) X's template parameters are a subset of Y's.

This definition applies to abstract components as well as concrete components. Specialization components are concrete components that *specialize* other concrete components.

The component coupling diagram in Figure 4.1 depicts the *specializes* relationship between two implementations: `AT_Queue.CT_2a` and `AT_Queue.CT_2`. Section 2 describes the specific *implements* and *uses* relationships also depicted in the diagram. `AT_Queue.CT_2a` is a specialization component which depends on the fully parameterized implementation `AT_Queue.CT_2`. In this example, the *specializes* relationship indicates that `AT_Queue.CT_2a` has been created by internally instantiating `AT_Queue.CT_2`. Within `AT_Queue.CT_2a`, one particular component that *implements* `AT_One_Way_List` must be used as the implementation parameter to instantiate `AT_Queue.CT_2`. The interface and behavior exported by the internal instantiation of `AT_Queue.CT_2` is re-exported as the complete interface and behavior of `AT_Queue.CT_2a`.

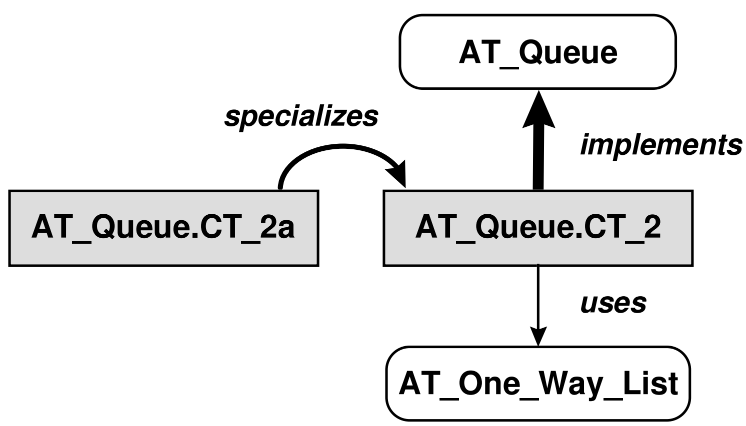

Figure 4.1 — The *Specializes* Relationship

Several implicit relationships not shown in Figure 4.1 may be deduced from those shown. First, `AT_Queue.CT_2a` *uses* `AT_Queue.CT_2` by the definition of *specializes*. Second, `AT_Queue.CT_2` also *uses* some component which *implements* `AT_One_Way_List` because of the *uses* relationship shown. Third, and most important, `AT_Queue.CT_2a` *implements* `AT_Queue` because of the *implements* relationship shown. Thus `AT_Queue.CT_2a` is behaviorally substitutable for `AT_Queue.CT_2` with respect to `AT_Queue`. The difference between these two implementations of `AT_Queue` is that the non-functional characteristics of `AT_Queue.CT_2a` have been fixed while some of the non-function characteristics of `AT_Queue.CT_2`, may still be adjusted.

4.2. Encoding Specialization Components

A specialization component is encoded in RESOLVE/Ada95 as a generic child unit package specification. No corresponding package body is needed. The parent unit is the abstract component which the component being specialized (and thus also the specialization component itself) *implements*. By convention, the name of a specialization component is the same as that of the component it *specializes* appended with a single letter used to distinguish between multiple specializations of the same component. For example, the name `AT_Queue.CT_2c` identifies the "third" specialization of the "second" implementation of the queue abstract template.

A specialization component contains a global context section, an interface section, a ***local instantiations*** section, and a representation section. Typically all of the parameters of the component being specialized are fixed. In this case, there will be no implementation parameters section and no generic formal parameters. The fixed dependencies on the component being specialized and on the components selected to serve as implementation parameters appear as `with` clauses in the global context section. The interface section of a specialization component declares the exported concrete type as a private type extension of the abstract type exported by the parent unit. This partial view of the exported type ensures that a client of this component has visibility to all operations described in the abstract parent unit. The implementations of these operations are provided in the private part and thus no operations need to be declared in the interface section.

The private part of the package contains the local instantiations and representation sections. The local instantiations section contains all package instantiations necessary to supply actual parameters to the final instantiation in this section. The final instantiation creates an instance of the component being specialized. This package instance supplies the concrete type to be re-exported by the specialization component. The representation section consists of a null record extension of the concrete type to be re-exported. This type extension creates the full view of the concrete exported type. The example in Section 4.3 demonstrates these techniques.

4.3. An Example: AT_Queue.CT_2a

This section includes the RESOLVE/Ada95 package specification for specialization component `AT_Queue.CT_2a`. This component is a concrete template which *specializes* `AT_Queue.CT_2` (shown in Section 2.4) and *uses* the specialization `AT_One_Way_List.CT_1a` to instantiate `AT_Queue.CT_2`. By the definition of *specializes*, the component shown also *implements* `AT_Queue`.

A component instantiation diagram summarizing the code shown in this section could simply appear as the diagram shown in Figure 2.5 with an "a" appended to the concrete component name and with the `AT_One_Way_List` implementation parameter removed. (This is how `AT_Queue.CT_2a` appears in Figure 6.1.) The component instantiation diagram shown in Figure 4.2 provides more detail, showing *how* `AT_Queue.CT_2a` is implemented.

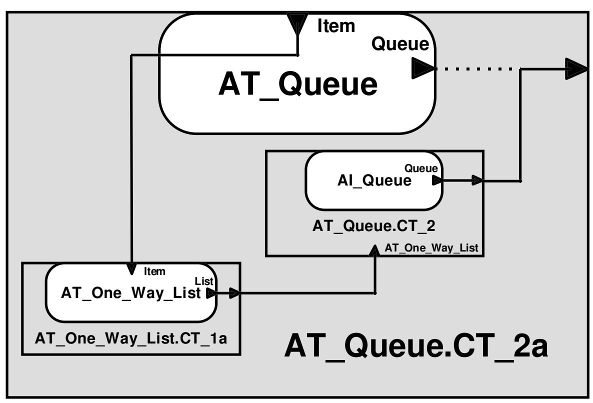

Figure 4.2 — A Detailed View of AT_Queue.CT_2a

The local instantiations section of AT_Queue.CT_2a includes three package instantiations. The first creates `AI_One_Way_List`, an abstract instance of `AT_One_Way_List` instantiated with the same `Item` type as `AT_Queue`. This instantiation is depicted in Figure 4.2 by the line connecting the `Item` specification parameter of `AT_Queue` to that of `AT_One_Way_List` on the left side of the diagram. The second instantiation creates `CI_One_Way_List`, a concrete instance of `AI_One_Way_List` formed by selecting the "1a" implementation of `AT_One_Way_List`. This instantiation is depicted by showing the implementation `AT_One_Way_List.CT_1a` around the specification `AT_One_Way_List` in the lower left corner of Figure 4.2. The final instantiation creates `CI_Queue`, a concrete instance of `AT_Queue.CT_2`, the component being specialized. This instantiation is depicted in Figure 4.2 by placing the `AT_Queue.CT_2` implementation around the `AI_Queue` specification and by the line connecting the type exported by `AT_One_Way_List.CT_1a` to the implementation parameter of AT_Queue.CT_2.

Note that the component name `AI_Queue` in Figure 4.2 refers to an instance of `AT_Queue` with the type `Item` fixed. Within the child unit `AT_Queue.CT_2a`, the package name `AT_Queue` implicitly refers to this package instance, an instance of its generic parent unit. In Figure 4.2, the line from the type exported by `AT_Queue.CT_2` to the type exported by `AT_Queue.CT_2a` depicts the inheritance link between these two types. The null record type extension at the end of `AT_Queue.CT_2a` encodes this link. The component instantiation diagram shown in Figure 4.2 summarizes the following code.

Package Specification for AT_Queue.CT_2a

```
-----------------------------------------------------------------------------
--
-- Component: AT_Queue.CT_2a
-- Relations: specializes AT_Queue.CT_2, uses AT_One_Way_List.CT_1a
--  Comments: -
--
-----------------------------------------------------------------------------
-- Global Context -----------------------------------------------------------
-----------------------------------------------------------------------------

with AT_Queue.CT_2;
with AT_One_Way_List.CT_1a;

-----------------------------------------------------------------------------

generic

package AT_Queue.CT_2a is

    -------------------------------------------------------------------------
    -- Interface ------------------------------------------------------------
    -------------------------------------------------------------------------

    type Queue is new AT_Queue.Queue with private;

    -------------------------------------------------------------------------

private

    -------------------------------------------------------------------------
    -- Local Instantiations -------------------------------------------------
    -------------------------------------------------------------------------

    package AI_One_Way_List is new
        AT_One_Way_List (Item => Item);

    package CI_One_Way_List is new
        AI_One_Way_List.CT_1a;

    package CI_Queue is new
        AT_Queue.CT_2 (
                AI_One_Way_List => AI_One_Way_List,
                           List => CI_One_Way_List.List
            );

    -------------------------------------------------------------------------
    -- Representation -------------------------------------------------------
    -------------------------------------------------------------------------

    type Queue is new CI_Queue.Queue with null record;

    -------------------------------------------------------------------------

end AT_Queue.CT_2a;
```

5. Checking Components

This section describes checking components and the associated *checks* relationship. Checking components are very useful components within the RESOLVE framework. Their utility largely results from the layered way in which RESOLVE components are designed and implemented. This section concludes with a sample RESOLVE/Ada95 checking component for the queue kernel component.

5.1. The *Checks* Relationship

One of the principles of the RESOLVE discipline is that clients should be responsible for checking the preconditions of operations. This approach avoids unnecessary inefficiency and simplifies component implementations. Furthermore, it respects the contractual relationship expressed by the *implements* relationship. A component implementer is responsible for providing all behavior described by the specification and no more. If a concrete component must handle exceptional events, then the abstract template should describe what behavior is required for exceptional events. If a client uses an operation when its precondition is not satisfied, then the concrete component exporting that operation is free to do anything or nothing from that point on.

During the testing and debugging of new implementations, unreliable code may erroneously call operations with violated preconditions. Since the behavior of the implementations is unspecified once a violation takes place, locating errors while debugging may be very difficult. Thus checking the preconditions of operations is useful when there is a lack of confidence in the correctness of implementations within a software system. A checking component addresses this problem by checking operation preconditions before calling unprotected operation implementations.

A checking component is a concrete component with characteristics similar to both an extension component and a concrete component. Like an extension component, it adds new functionality to another component. The additional behavior consists of checking to see if a precondition is satisfied and, if not, reporting the violation and halting execution of the program immediately. Unlike an extension component, however, a checking component provides no additional *specified* behavior. Like a typical concrete component, a checking component provides the interface and behavior specified by an abstract template. However, unlike the implementations described in Section 2, a checking component does not directly provide the specified functionality it exports. Instead, a checking component *uses* the abstract template which it *implements* to provide its exported behavior. That is, a checking component is implemented by layering the checking functionality on top of a concrete component that provides the desired specified behavior. Therefore, a single checking component may be used to check *any* concrete component which *implements* the abstract component being checked.

The ***checks*** relationships expresses *part* of the dependency between a concrete component and an abstract component. *Checks* is defined as follows:

> Concrete component X *checks* abstract component Y if and only if (a) X *implements* Y and (b) X reports violations of all operation preconditions described by Y.

In terms of this relationship, a checking component is a concrete component which *checks* and *uses* the same abstract component. It is possible to have a concrete component which *checks* but does not *use* an abstract component. However, such a component, one which directly implements the behavior of the component it *checks*, would not be nearly as "useful" as a checking component.

The component coupling diagram in Figure 5.1 depicts the *checks* relationship between the checking component `AT_Queue.CT_0`, and the abstract component `AT_Queue`. The only precondition specified by `AT_Queue` is that the `Dequeue` operation may not be called with an empty queue object. Therefore, if a client of `AT_Queue.CT_0` tries to dequeue an item from an empty queue, an error report will be issued and the program will halt. Note that in the case of a checking component, the *uses* relationship is omitted from the component coupling diagram. A slightly more complex notation which includes template parameters makes explicit this *uses* relationship for checking components.

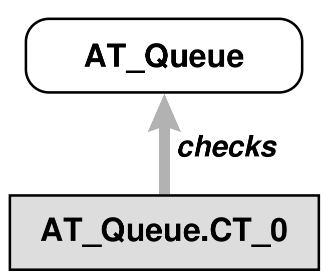

Figure 5.1 — The *Checks* Relationship

5.2. Encoding Checking Components

A checking component is encoded in RESOLVE/Ada95 as a generic child unit of the abstract template which it checks. By convention, the package name `CT_0` prefixed by the name it its parent unit. Since the child unit has direct visibility to its parent, the *uses* relationship with its parent is implicit. Thus the global context section in a checking component is empty. The implementation parameters section contains a single generic formal derived type. The name of this type is the name of the exported concrete type prefixed by the string `Base_`. The actual parameter for this formal type must be a type derived from the abstract type exported by the parent unit. Thus, the type exported by any component which *implements* the parent unit may serve as an actual parameter.

Figure 5.2 shows the RESOLVE/Ada95-specific component instantiation diagram depicting the parent-child relationship between a kernel abstract component and its concrete checking component. In this example, `AT_Queue.CT_0` is the checking component for `AT_Queue`. The implementation parameter `Base_Queue`, like all other implementation parameters, is shown along the bottom edge of the concrete component. The dotted line conveys the constraint that the actual parameter supplied for `Base_Queue` must be a type derived (possibly indirectly) from the type exported from an instance of `AT_Queue`.

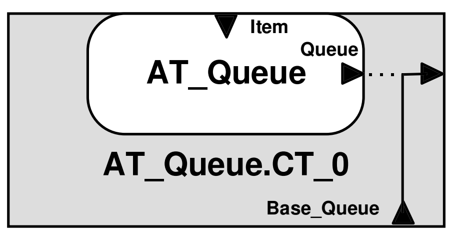

Figure 5.2 — Abstract Parent and Concrete Checking Child

The interface section of a checking component package specification includes the declaration of the concrete exported type. The exported type is a null record extension of the imported generic formal type. Thus the representation of the component on which the checking component is layered is not altered. The inheritance link established by this type extension is depicted in Figure 5.2 by the line connecting the `Base_Queue` implementation parameter to the type exported by `AT_Queue.CT_0`.

For each operation with precondition (aside from "true") in the parent unit, the child unit includes a corresponding concrete subprogram signature. These operations *override* those with matching signatures inherited from the generic type parameter. This is another use of mixin inheritance. While overriding inherited operations is generally a threat to preserving behavioral substitutability, this overriding is safe as long as the RESOLVE/Ada95 discipline is followed. It is safe because the overriding operation provides the identical *specified* functionality as the overridden operation.

The package body of a checking component provides the implementations of each overriding operation. The body of each overriding operation first checks to see if the precondition for that operation is satisfied. One of RESOLVE's guidelines for the selection of primitive operations is that they include any operations necessary to check all preconditions. Therefore, the code to check the precondition may be layered. Once the check is made, some mechanism must be used to report the error and halt the program if necessary. The ideal mechanism used depends upon the run-time environment in which the RESOLVE/Ada95 programs will be run. The code shown in Section 5.3 uses the Assert pragma of the GNAT Ada compiler to report precondition violations and terminate execution.

If there are no precondition violations, the body of the overriding operation calls the operation it has overridden. This requires use of a ***view conversion***. Inside the body of the overriding operation, a formal parameter of the exported type is converted to the type of its parent when used as an actual parameter the call to the overridden operation. This technique is applied to the `Dequeue` operation as shown in Section 5.3.

5.3. An Example: AT_Queue.CT_0

This section includes the RESOLVE/Ada95 package specification and body for the checking component `AT_Queue.CT_0.` This component is a concrete template which *checks* the kernel abstract component `AT_Queue` shown in Section 1.3. The implementation shown here depends on the `Assert` pragma as implemented by the GNAT Ada compiler. The component instantiation diagram shown in Figure 5.2 summarizes the following code.

Package Specification for AT_Queue.CT_0

```
-----------------------------------------------------------------------------
--
-- Component: AT_Queue.CT_0
-- Relations: checks AT_Queue
--  Comments: GNAT -a switch must be on when compiling this and client units
--
-----------------------------------------------------------------------------
-- Global Context -----------------------------------------------------------
-----------------------------------------------------------------------------

-----------------------------------------------------------------------------

generic

    -------------------------------------------------------------------------
    -- Implementation Parameters --------------------------------------------
    -------------------------------------------------------------------------

    type Base_Queue is new AT_Queue.Queue with private;

    -------------------------------------------------------------------------

package AT_Queue.CT_0 is

    -------------------------------------------------------------------------
    -- Interface ------------------------------------------------------------
    -------------------------------------------------------------------------

    type Queue is new Base_Queue with null record;

    -------------------------------------------------------------------------
    -- Overriding Operations ------------------------------------------------
    -------------------------------------------------------------------------

    procedure Dequeue (
            q : in out Queue;
            x : in out Item
        );

    -------------------------------------------------------------------------

end AT_Queue.CT_0;
```

Package Body for AT_Queue.CT_0

```
----------------------------------------------------------------------------
--
-- Component: AT_Queue.CT_0
-- Relations: checks AT_Queue
--  Comments: GNAT -a switch must be on when compiling this and client units
--
----------------------------------------------------------------------------
-- Global Context ----------------------------------------------------------
----------------------------------------------------------------------------

----------------------------------------------------------------------------

package body AT_Queue.CT_0 is

    ------------------------------------------------------------------------
    -- Interface Operations --------------------------------------------------
    ------------------------------------------------------------------------

    procedure Dequeue (
            q : in out Queue;
            x : in out Item
        ) is
            length : Integer := 0;
    begin
        Get_Length (q, length);
        pragma Assert (length > 0,
            "Dequeue pre-condition (q /= empty_string) violated");
        Dequeue (Base_Queue(q), x);
    end Dequeue;

    ------------------------------------------------------------------------

end AT_Queue.CT_0;
```

6. Instantiation of Components

This section briefly explains how the components described in Sections 1-5 may be instantiated for use in applications. Section 6.2 includes sample RESOLVE/Ada95 code that shows how a queue concrete instance may be built from components described in earlier sections.

6.1. Building Concrete Instances

A concrete component is a subsystem implementation for which all parameters have been fixed. Thus a concrete component exports a concrete type which may be used directly by another component or application program. Section 4.3 explained the instantiation of several components within the package specification of `CT_Queue.CT_2a`. Instantiation of components for use by application programs is similar.

Before building a *new* concrete instance from existing components, an implementer should check first to see that no available concrete instances already provide the desired functional and non-functional subsystem requirements. A component library suitable for production use is likely to contain a wide variety of ready-to-use concrete instances, especially for common data structures such as queues.

The process of building a concrete instance generally proceeds as follows. First, a kernel abstract instance is produced by an instantiation which binds the specification parameters of a kernel abstract template. Then a kernel concrete instance is produced by an instantiation which binds a specific implementation to the kernel abstract instance. This instantiation may involve supplying actual parameters for implementation parameters. Then, if necessary, the functionality of the kernel concrete instance may be augmented through a sequence of abstract and concrete extensions. For each abstract extension, the type exported from the most recently constructed concrete instance serves as the actual parameter for the specification parameter. Each abstract instance created by an abstract extension must then be supplied with an implementation in the same manner as the kernel abstract instance. Checking concrete instances may be introduced after the kernel concrete instance has been created. A checking component is produced by supplying a concrete instance as a realization parameter to a checking concrete template.

6.2. An Example: CI_Enhanced_Integer_Queue_1

This section includes the RESOLVE/Ada95 package specification for a concrete instance providing an integer queue extended with operations to reverse and replicate (copy the value of) the queue. This code provides a simple example of how components described in earlier sections may be composed to build a new component. As noted in the last section, a client application programmer should rarely need to build a component as simple as this. A component library suitable for production use would probably contain several ready-to-use queue components incorporating a number of generally useful extensions such as operations to reverse and replicate queues. Those components could be built following the approach shown in this section.

The component instantiation diagram in Figure 6.1 summarizes the code which follows it. One new notation introduced in this diagram is the striped rectangle containing "Integer". This notation is used for Ada built-in types such as Integer, Character, and Boolean. Since built-in types do not have a `Swap` operation, the special package `CI_Scalar_Operations` is used to provide `Swap` for built-in types.

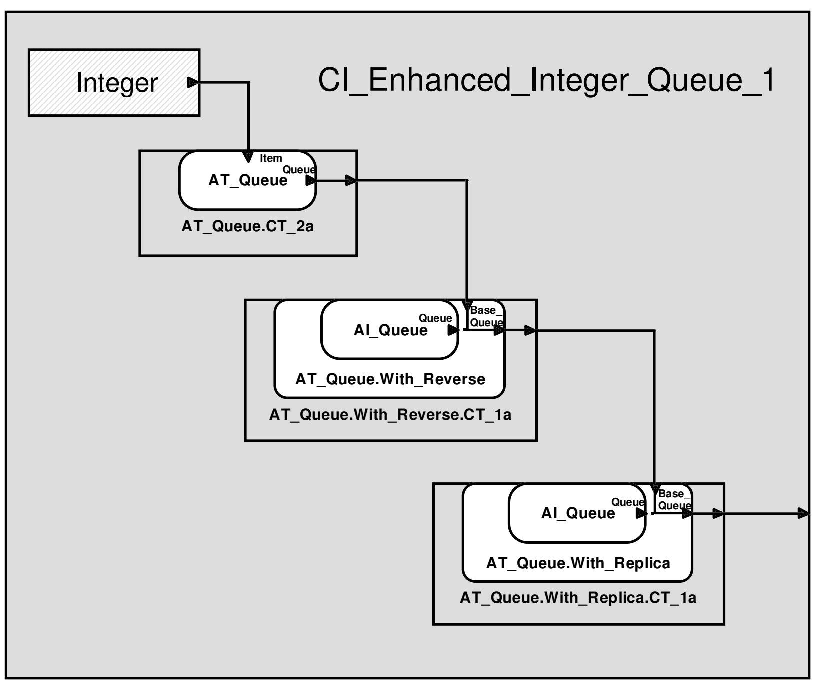

Figure 6.1 — A Detailed View of CI_Enhanced_Integer_Queue_1

Package Specification for CI_Enhanced_Integer_Queue_1

```
----------------------------------------------------------------------------
--
-- Component: CI_Enhanced_Integer_Queue_1
-- Relations: uses AT_Queue.CT_2a, uses AT_Queue.With_Reverse.CT_1a,
--            uses AT_Queue.With_Replica.CT_1a
--  Comments: CI_Scalar_Oparations provides Swap for built-in type Integer
--
----------------------------------------------------------------------------
-- Global Context ----------------------------------------------------------
----------------------------------------------------------------------------

with CI_Scalar_Operations;
use  CI_Scalar_Operations;

with AT_Queue.CT_2a;
with AT_Queue.With_Reverse.CT_1a;
with AT_Queue.With_Replica.CT_1a;

----------------------------------------------------------------------------

package CI_Enhanced_Integer_Queue_1 is

    ------------------------------------------------------------------------
    -- Local Instantiations ------------------------------------------------
    ------------------------------------------------------------------------

    -- instantiate AT_Queue with Integer

    package AI_Integer_Queue is new
        AT_Queue(Item => Integer);

    ------------------------------------------------------------------------

    -- implement AI_Integer_Queue with CT_2a

    package CI_Integer_Queue is new
        AI_Integer_Queue.CT_2a;

    ------------------------------------------------------------------------

    -- extend AI_Integer_Queue with Reverse

    package AI_Integer_Queue_With_Reverse is new
        AI_Integer_Queue.With_Reverse (
            Base_Queue => CI_Integer_Queue.Queue);

    ------------------------------------------------------------------------

    -- implement AI_Queue_With_Reverse with CT_1a

    package CI_Integer_Queue_With_Reverse is new
        AI_Integer_Queue_With_Reverse.CT_1a;
```

```
    ------------------------------------------------------------------------

    -- extend AI_Integer_Queue with Replica

    package AI_Integer_Queue_With_Replica is new
        AI_Integer_Queue.With_Replica (
            Base_Queue => CI_Integer_Queue_With_Reverse.Queue);

    ------------------------------------------------------------------------

    -- implement AI_Queue_With_Replica with CT_1a

    package CI_Integer_Queue_With_Reverse_And_Replica is new
        AI_Integer_Queue_With_Replica.CT_1a;

    ------------------------------------------------------------------------
    -- Interface -----------------------------------------------------------
    ------------------------------------------------------------------------

    type Queue is new CI_Integer_Queue_With_Reverse_And_Replica.Queue
        with null record;

    ------------------------------------------------------------------------

end CI_Enhanced_Integer_Queue_1;
```