CHAPTER III

RESOLVE

RESOLVE (an acronym for REusable SOftware Language with Verifiability and Efficiency) is a programming language and environment specifically created to encourage the design and implementation of reusable software components. RESOLVE is an imperative language, with control structures similar to those found in most structured languages such as Pascal and Ada. A program is organized as a collection of separately-compiled modules. The behavior of each module is formally specified in a *conceptualization*, with the structures and code implementing each conceptualization contained in a *realization*. RESOLVE permits (and even encourages) several realizations for any conceptualization, allowing multiple implementations, each with potentially different performance characteristics.

An interesting feature of RESOLVE is that no types are built-in to the language. Instead, every type is provided by some conceptualization, including those normally built-in such as integer, character, and boolean. Likewise, structured types such as arrays and records are not defined in RESOLVE, but are defined by conceptualizations. A similar approach is taken with respect to pointer variables. This design makes RESOLVE very regular if a bit primitive.

Another feature of RESOLVE is that data is moved by swapping the contents of two variables, rather than copying the contents of one variable to another. The ability to make a copy of a data value is not a built-in operation in RESOLVE. This unique approach offers several significant advantages.

Finally, every type has an initial value defined for it, and every variable is automatically initialized to an initial value of its type before it is referenced in any executable statement. This facilitates verification and eliminates the possibility of program bugs caused by uninitialized variables.

This chapter informally defines RESOLVE and argues why it encourages the design and implementation of reusable software components, as defined in Section 2.3. RESOLVE

is defined by presenting examples of RESOLVE modules, with an accompanying discussion of the motivation and implications with respect to reusability.

3.1 Conceptualizations and Type Parameters

The functionality of a RESOLVE component is defined within a *conceptualization*. This section presents some basic characteristics of conceptualizations. Advanced features of conceptualizations are presented in Sections 3.5 and 3.6. The example used throughout this section is the LIFO stack.

3.1.1 LIFO Stacks and Conceptualization Stack_Template

The Last In First Out stack is perhaps the most studied abstract data type in computer science. It seems that everyone uses it as a benchmark in describing approaches to data specification and abstraction.

There are several reasons why the stack is a logical choice for describing an approach to data abstraction. First, it is a good reference point, since the general intuitive notion of a stack is understood by the vast majority of practitioners of computer science. Even though the precise definition of stacks varies somewhat among camps, the typical person reading the literature will most likely understand the problem being addressed. Second, stacks are simple enough to describe in a relatively small amount of space, yet are surprisingly non-trivial. And finally, stacks have two simple implementations — an array with a top index, and a list.

A RESOLVE conceptualization for stacks is presented in Figure 2.

```
conceptualization Stack_Template
  parameters
    type Item
  end  parameters

  auxiliary
    math  facilities
      String_Theory is String_Theory_Template (math[Item])
        renaming
          String_Theory.String as String
          String_Theory.Lambda as Lambda
          String_Theory.Post as Post
        end  renaming
    end  math  facilities
  end  auxiliary

  interface
    type Stack is modeled by String
      exemplar s
      initially "s = Lambda"
    end Stack

    procedure Push
      parameters
        alters s : Stack
        consumes x : Item
      end  parameters
      ensures "s = Post(#s, #x)"
    end Push

    procedure Pop
      parameters
        alters s : Stack
        produces x : Item
      end  parameters
      requires "s ≠ Lambda"
      ensures "#s = Post(s, x)"
    end Pop

    control Is_Empty
      parameters
        preserves s : Stack
      end  parameters
      ensures Is_Empty iff "s = Lambda"
    end Is_Empty
  end  interface
```

Figure 2

Specification for a Module Providing the Generic Type Stack

Figure 2 (continued)

```
    description

      Stack_Template provides the type family "Stack of Item",
      where Item is any type.  In the formal specifications
      above, an abstract stack is a string of Items, with the top
      of the stack at the right end of the string.  Initially,
      every Stack is empty.

      • "Push(s,x)" pushes x onto stack s.  Since x is a consumes
        parameter, it has an initial value for its type upon
        return.

      • "Pop(s,x)" pops the top element off stack s and returns
        it in x.

      • "Is_Empty(s)" returns yes if and only if s is an empty
        stack.

    end  description

end Stack_Template
```

Perhaps one of the most striking characteristics of a RESOLVE module is its verboseness, especially with respect to keywords. This was a conscious design decision, made without apologies. RESOLVE modules are written primarily to be read and understood by programmers, and it is felt that cryptic abbreviations for keywords thwart this goal. If an appropriate editing environment is used for module construction — such as the one presented in Chapter 5 — a programmer never types keywords. Thus verbosity need not imply a penalty on the time necessary to construct a program.

A RESOLVE conceptualization provides a formal specification as well as an informal description of a software component. The majority of the text of the conceptualization is devoted to formal specification, which provides the “official” definition of the component. Informal prose descriptions of a component are also useful, since they can provide the human reader with a “feel” for what the component does. The informal description is contained within the **description** section of a conceptualization. It must be understood that descriptions are “unofficial,” and in the event that the informal

description contradicts the formal specification of a component, the specification is *always* used as the component definition.

A type is formally defined in RESOLVE by modeling it as a mathematical type from some mathematical theory, and operations are formally defined by pre- and post-conditions written as assertions in one or more mathematical theories. The reader should recognize this as the model-based approach to formal specification, discussed in Section 2.4.2.3.

Let's take a closer look at what exactly is defined by Stack_Template. The interface section contains definitions of all items provided by the conceptualization to a client. It should be apparent that Stack_Template is exporting (i.e., providing) a type (called Stack) and operations to add an item to a stack (procedure Push), remove an item from a stack (procedure Pop), and determine if a stack is the empty stack (control Is_Empty).

In this example, stacks are modeled as mathematical strings from string theory, with the right end of a string containing the top item on the stack. Operations Push, Pop, and Is_Empty are defined by assertions in string theory. The empty stack is modeled as the empty string, operation Push concatenates an item onto the right end of a string, Pop removes and returns the rightmost item from a string, and Is_Empty determines if a string is the empty string.

3.1.2 Generic Conceptualizations and Type Parameters

The conceptualization in Figure 2 is for homogeneous stacks, where all items contained in a stack are the same type. However, the actual type of items to be contained in the stack is not specified in the conceptualization, but is indicated as conceptualization parameter Item. For this reason Stack_Template is a *generic* conceptualization. There are no restrictions on what type Item may be, so this one conceptualization actually defines an infinite number of stack types, including, for example, stacks of integers, stacks of characters, and even stacks of stacks of integers. Stack_Template defines a *type family* called Stack and *operation families* called Push, Pop, and Is_Empty. An actual type (e.g., stack of integers) and actual operations (e.g., Push, Pop, and Is_Empty for stacks of integers) are created only when the conceptualization is *instantiated*, as discussed in Section 3.2.3.

As demonstrated by this example, formal parameters to a conceptualization are declared in the parameters section. Two kinds of parameters can be passed to create a conceptualization instance — types and facilities. The role that type parameters play is discussed in this example, while facility parameters are discussed in Section 3.5.

3.1.3 Mathematical Theory Modules

Conceptualization Stack_Template is defined in terms of mathematical string theory. But what exactly is mathematical string theory? Anyone who has taken a course in discrete mathematics probably has encountered string theory at least enough to have an intuitive understanding of it. String theory defines a type for strings over some alphabet, along with a constant representing the empty string, and definitions such as concatenation of two strings. It should be clear that string theory is parameterized (i.e., generic) because it is defined with respect to some alphabet.

Relying upon a person's intuition of string theory (or any mathematical theory for that matter) is not precise enough for formal specification, and in fact would lead to ambiguities — the very problem formal specification is supposed to solve! For this reason, mathematical theories must themselves be formally specified. In RESOLVE this is accomplished within a theory module, which is a module that specifies mathematical types and mathematical functions. Theory modules are similar to conceptualizations, except that theory modules provide *mathematical* types and functions, whereas conceptualizations provide *program* types and operations. The precise contents and syntax of theory modules have not been finalized, and in fact are not a part of the research presented here.

Nonetheless, the syntactic slot for specifying mathematical theories is included in conceptualizations. Specifically, theories are instantiated within the **auxiliary** section of a conceptualization, in a manner similar to conceptualization instantiation discussed in Section 3.2.3. In the Stack_Template example, theory String_Theory_Template is instantiated to create a math *facility* called String_Theory. This provides a mathematical type for strings over (the math model of) type Item, called String_Theory.String (renamed String[27]). A function that concatenates an item onto the right end of a string

---

27 In essence, renaming an item provides a shorthand identifier for it. Here, for example, the type provided by this instance can be called either String_Theory.String or String.

called String_Theory.Post (renamed Post), and the empty string constant String_Theory.Lambda (renamed Lambda), are also provided. This mathematical type is used as the model for stacks, as indicated in the declaration for type Stack.

3.1.4 Specification of Types

The exemplar clause in a type specification indicates an identifier that represents a prototypical variable of that type. The exemplar is referenced in the assertions of the type specification, and is also treated as a program variable in initialization and finalization routines, as discussed in Section 3.7.2.

Every program type in RESOLVE has an initial value specification defined in the initially clause of the type definition. Every variable is guaranteed to have a value that meets the initial value specification for its type before the variable's first reference. The motivation for initial values was discussed in Section 2.5.3.6. In this example every variable s of type Stack initially satisfies the assertion "`s = Lambda`"; in other words, all stacks are initially modeled by the empty string, and hence are empty stacks.

3.1.5 RESOLVE Operations

Three kinds of operations can be defined within a RESOLVE program — procedures, functions, and controls. The differences among these operations are the manner in which information is exchanged between the invoker and the operation. With procedures, all information is exchanged via parameters. This is similar to procedures in most structured languages such as Pascal and Ada, except that RESOLVE has no global variables, so indeed parameters are generally the *only* means of exchanging information between an invoker and procedure. (There are, however, shared module variables available to operations provided by a particular module instance.)

RESOLVE functions are similar to procedures, except that a function returns exactly one value to the invoker, and parameters are used exclusively to provide the function with information from the invoker. Because functions cannot access global variables and parameters cannot be used to return information to the invoker, functions in RESOLVE do not have side effects. A function can be invoked only within an assignment statement, which is discussed in Section 3.3.1.4.

Conceptualization Stack_Template does not define any functions. However, Figure 3 contains the definition for a function Top that returns a copy of the top item of a stack without effectively removing it from the stack[28]. In this example, it is assumed that Item, Stack, String, Lambda, and Post are defined as in Stack_Template in Figure 2.

```
function Top returns topitem : Item
  parameters
    preserves s : Stack
  end  parameters
  requires "s ≠ Lambda"
  ensures "∃ r : String, s = Post(r,topitem)"
end Top
```

Figure 3

Specification of Function Top

Controls are similar to functions in that exactly one piece of information is returned to the invoker, and parameters are used solely to provide the control with information from the invoker. The difference between functions and controls is that a control does not return a data value, but a state value used exclusively to determine the action of an if or while statement. The implications are obvious — controls can be invoked only within an if or while statement, and controls return one of two possible state values. As presented in Sections 3.3.2 and 3.3.3 execution of a control terminates when one of two return statements is executed within the control — return yes or return no. The state value returned by the control is determined by which return statement is executed.

The motivation for including control operations in RESOLVE centers around the design goal of not having any types built-in to the language, per the discussion in Section 2.5.3.7. In most languages, the alternation and iteration constructs (e.g., if and while statements) require a type be built-in to the language (e.g., boolean). Controls permit RESOLVE to be defined with no built-in types. (It is crucial to understand that “yes” and “no” are *not* data values of some type, but rather state values used exclusively to control the action taken by if and while statements. For example, the result of a control invocation cannot be assigned to a variable.)

---

28 The reason Top is not included as an operation in Stack_Template is because it is a secondary operation, as defined in Section 2.4.1.3.

3.1.6 Parameter Modes

An operation's formal parameters are declared within the parameters section of the operation. In addition to an identifier and type, each formal parameter has an associated *mode* — consumes, produces, alters, or preserves. The parameter mode describes how a parameter is used in the exchange of information between invoker and operation, and also helps streamline pre- and post-conditions for the operation. Parameter modes do *not*, however, describe the *parameter passing mechanism* used to exchange information between invoker and operation. As discussed in Section 3.3.1.3, all parameters are passed by swapping.

Information flows strictly from the invoker to the operation via a consumes parameter. As the name implies, the information provided by the invoker (in the actual parameter) is "consumed" by the operation, and in fact the actual parameter contains an initial value for its type when the operation returns. For example, the item to be pushed onto a stack is supplied to procedure Push by the invoker, and information is flowing strictly from the invoker to procedure Push via parameter x; thus, x is a consumes parameter. The motivation for specifying this unusual behavior has to do with efficient implementation of generic modules, discussed in Section 3.8.3.

A produces parameter is in a sense just the opposite of consumes, since information flows strictly from the operation to the invoker. Information originally in the actual parameter is discarded (i.e., finalized) by the operation. For example, the item removed from a stack by procedure Pop is returned to the invoker through parameter x. Since x is not used to supply Pop with information from the invoker, it is a produces parameter.

An alters parameter indicates that useful information is passed to the operation by the invoker, and also returned to the invoker by the operation. The information returned might not be the same information originally sent to the operation. In other words, the invoker gives information to the operation, the operation possibly alters that information, and then gives it back to the invoker. For example, procedure Push is given a stack via parameter s, which it modifies (by concatenating a new item to it), and then gives back to the invoker. Thus, parameter s is an alters parameter.

A preserves parameter is similar to alters except that the value returned by the operation is guaranteed to be the same as the value sent to it. From the invoker's point of view, it

lets the operation use a value, and the operation agrees not to change it. For example, control Is_Empty is given a stack via parameter s, determines if s is the empty stack, and then returns it to the invoker unchanged. Thus, s is a preserves parameter.

It is important to note that an operation is allowed to change a preserves parameter during its execution, *as long as it restores the parameter to its original value before returning*. For example, an operation is permitted to pop items from a stack that is passed to it as a preserves parameter, provided that all items are pushed back onto the stack before the operation returns. This is a subtle yet important characteristic of preserves parameters.

It should also be noted that preserves is the only mode that does not potentially alter the value of the actual parameter. RESOLVE functions and controls are not allowed to alter their parameters. Therefore, all parameters to functions and controls must be preserves parameters.

3.1.7 Operation Specification

The effect of an operation is formally defined using pre- and post-conditions. A requires clause, if present, specifies the pre-condition of the operation. If a requires clause is not present, the pre-condition is assumed to be true (indicating the operation does not have a pre-condition). Similarly, the post-condition of an operation is specified in an ensures clause. Since each operation is assumed to have some effect, every operation must have an ensures clause.

The requires clause is an assertion that the operation assumes is true at the time it is invoked. Normally the requires clause specifies restrictions placed upon the values passed to the operation by the invoker. For example, it is not meaningful to pop from an empty stack, so the requires clause of procedure Pop specifies that parameter s must not be the empty stack (which is modeled as the empty string).

The ensures clause is an assertion the operation guarantees to be true when it returns, *provided the requires clause was true when the operation was invoked*. The implications of this last part are discussed shortly. For now, let's assume the requires clause was met when the operation was invoked. The ensures clause usually relates the values of parameters at the end of the operation to the original values of parameters

when the operation was invoked. In other words, it is necessary to reference the values of parameters at two points in time — the value at the beginning of the operation and the value at the end. Within an ensures clause, a "#" preceding a parameter identifier (e.g., #s) denotes the value of that parameter when the operation is invoked. A parameter identifier without a "#" (e.g., s) denotes the value of that parameter when the operation returns.

For example, the ensures clause of procedure Pop is "`#s = Post(s,x)`", meaning the string created by concatenating the value of stack s (modeled as a string) with item x when Pop returns equals the string contained in s when Pop was invoked. This is a somewhat roundabout way of saying that Pop has the effect of removing the rightmost element of the string modeling stack s. This example also demonstrates that ensures clauses are indeed assertions and not assignment statements.

Each parameter mode can affect requires and ensures clauses in two ways — it may implicitly add a clause to the ensures clause, and it may place restrictions upon how a parameter may be used in assertions. For example, a produces parameter is not used to pass information to an operation, so it may not be mentioned in a requires clause.

A consumes parameter always has an initial value when the operation returns. In effect, the conjunction "`and init(x)`" is implicitly part of the ensures clause for any consumes parameter x. It is never necessary (and is not valid) to mention the new value of a consumes parameter within an ensures clause.

Similarly, the value of a preserves parameter at the end of the operation is the same as it was at the beginning, so the conjunction "`and x = #x`" is implicitly part of the ensures clause for any preserves parameter x. This means there is no difference between x and #x in the ensures clause. By convention only x may appear there.

The value of a produces parameter at the beginning of an operation cannot have an effect upon the outcome of that operation. Thus, "x" cannot appear in the requires clause and "`#x`" cannot be mentioned in an ensures clause for any produces parameter x. A similar restriction holds for the identifier representing the value returned by a function (e.g., topitem in Figure 3).

Finally, let's return to an issue raised in the previous discussion — what happens if an operation is invoked and its requires clause is not met? In this situation nothing is assumed about the operation's effect, so the operation can do anything, including crash the system, return bogus results, or commence World War Three. The designer of an operation does not need to specify what happens if the requires clause is violated. Likewise, the programmer implementing the operation does not have to worry about what to do in this situation — anything is considered valid! Put another way, the requires clause specifies under what conditions it is meaningful to call an operation. Invoking an operation when the requires clause is violated is meaningless, and therefore the results of that invocation are meaningless. The problem is not in the operation definition, but rather in the client invoking the operation.

3.1.8 Another Example: Conceptualization One_Way_List_Template

For a second example of a RESOLVE conceptualization, let's examine a *one-way list*, which is a structure useful for storing elements that are accessed sequentially in one direction only (e.g., left to right). A one-way list can be described abstractly as a sequence of items with a marker of the "current position," which is called the *fence*, located between two of the items in the sequence. For example, `<3 9 4‡8 1>` represents a list of integers consisting of 3, 9, 4, 8, and 1, with the fence (denoted by "‡") between 4 and 8. `<3 9 4 8 1‡>` represents a list containing the same elements as before, but with the fence at the right end. `<‡3 9 4 8 1>` represents a list with the fence at the left end. The operations on a one-way list allow the contents of the sequence to be altered, the fence to move a step at a time in one direction (hence, the name one-way), and tests concerning the position of the fence.

This structure is often called a "linked list" in data structures texts. This name is inappropriate — the structure described is a list, and "linked" is simply one of several possible *representations* of lists. The abstract descriptions of a one-way list and its operations most assuredly should *not* talk about nodes and pointers, which are implementation details.

A RESOLVE conceptualization for one-way lists is presented in Figure 4. The informal description section is not included in this figure, since it is essentially presented in the accompanying text.

```
conceptualization One_Way_List_Template

  parameters
    type Item
  end  parameters

  auxiliary
    math  facilities
      String_Theory is String_Theory_Template (math[Item])
        renaming
          String_Theory.String as String
          String_Theory.Lambda as Lambda
          String_Theory.Pre as Pre
          String_Theory.Post as Post
          String_Theory.Cat as Cat
        end  renaming

      Tuple_2_Theory is Tuple_2_Theory_Template(String,String)
        renaming
          Tuple_2_Theory.Tuple as List_Model
          Tuple_2_Theory.Projection_1 as Left
          Tuple_2_Theory.Projection_2 as Right
        end  renaming
    end  math  facilities
  end  auxiliary

  interface
    type List is modeled by List_Model
      exemplar L
      initially "Left(L) = Lambda and Right(L) = Lambda"
    end List
```

Figure 4

Specification for a Module Providing the Generic Type List

Figure 4 (continued)

```
procedure Reset
  parameters
    alters L : List
  end  parameters
  ensures "Left(L) = Lambda and
             Right(L) = Cat(Left(#L),Right(#L))"
end Reset

procedure Advance
  parameters
    alters L : List
  end  parameters
  ensures "Cat(Left(L),Right(L)) =
           Cat(Left(#L),Right(#L))
           and ∃x:Item, Left(L) = Post(Left(#L),x)"
end Advance

procedure Add_Right
  parameters
    alters L : List
    consumes x : Item
  end  parameters
  ensures "Left(L) = Left(#L) and
           Right(L) = Pre(#x,Right(#L))"
end Add_Right

procedure Remove_Right
  parameters
    alters L : List
    produces x : Item
  end  parameters
  requires "Right(L) ≠ Lambda"
  ensures "Left(L) = Left(#L) and
           Pre(x,Right(L)) = Right(#L)"
end Remove_Right

procedure Swap_Rights
  parameters
    alters L1 : List
    alters L2 : List
  end  parameters
  ensures "Left(L1) = Left(#L1) and
           Right(L1) = Right(#L2) and
           Left(L2) = Left(#L2) and
           Right(L2) = Right(#L1)"
end Swap_Rights
```

Figure 4 (continued)

```
      control At_Left_End
        parameters
          preserves L : List
        end  parameters
        ensures At_Left_End iff "Left(L) = Lambda"
      end At_Left_End

      control At_Right_End
        parameters
          preserves L : List
        end  parameters
        ensures At_Right_End iff "Right(L) = Lambda"
      end Is_Empty
    end  interface

    description
      ...
    end  description

  end One_Way_List_Template
```

In this conceptualization, type family List is modeled as a cartesian product of two strings. Strings are specified in theory String_Theory_Template, which provides math type String, math constant Lambda for the empty string, and math functions Pre, Post, and Cat for constructing strings. (Pre defines concatenation of an item onto the left of a string, Post defines concatenation of an item onto the right end of a string, and Cat defines concatenation of two strings.)

Cartesian products of two (parameter) types are formally defined in theory Tuple_2_Theory_Template, which provides a math type called Tuple, and math functions for projecting either part of a Tuple, called Projection_1 and Projection_2. Here, the instance of Tuple_2_Theory_Template is called Tuple_2_Theory, and the type defined is renamed List_Model, with the projection functions renamed Left and Right.

One of the strings of a List holds the items to the left of the fence, and the other string holds the items to the right of the fence. These two strings are projected from a list with math functions Left and Right, respectively. A list is initially empty, represented by both Left and Right of the List being the empty string.

Procedure Reset places the fence at the left end of a List. Advance moves the fence one item to the right. Add_Right inserts an item into a List immediately to the right of the fence, and Remove_Right removes and returns the item immediately to the right of the fence. Procedure Swap_Rights exchanges the right parts of two lists. Controls At_Left_End and At_Right_End determine if the fence is at the left or right end of a List, respectively.

3.1.9 Summary

In this section RESOLVE conceptualizations for two common data structures were presented — the LIFO stack and the one-way list. These conceptualizations demonstrate some fundamental characteristics of RESOLVE conceptualizations, including the role type parameters play in the specification of generic conceptualizations, the motivation for math facilities in formal specification, the method used to formally specify types and operations, the three kinds of RESOLVE operations (procedures, functions, and controls), and the role of parameter modes.

3.2 Simple Realizations

A RESOLVE realization contains the data structures and algorithms that implement the functionality specified in a conceptualization. This section discusses realizations by presenting a realization of Stack_Template using a one-way list. This discussion actually addresses two issues — how to implement a conceptualization, and how to make use of the types and operations specified in a conceptualization by instantiating it.

3.2.1 Realization Stack_Real_1 of Stack_Template

A realization of Stack_Template using One_Way_List_Template is presented in Figure 5.

```
realization of Stack_Template by Stack_Real_1

  conceptualization  auxiliary
    renaming
      String_Theory.Reverse as Reverse
    end  renaming
  end  conceptualization  auxiliary

  realization  auxiliary
    facilities
      List_Facility is One_Way_List_Template (Item)
        realized by List_Real_1
        renaming
          List_Facility.List as List
          List_Facility.Add_Right as Add_Right
          List_Facility.Remove_Right as Remove_Right
          List_Facility.At_Right_End as At_Right_End
          List_Facility.Right as Right
          List_Facility.Left as Left
        end  renaming
    end  facilities
  end  realization  auxiliary

  interface
    type Stack is represented by List
      exemplar s_rep
      conventions "Left(s_rep) = Lambda"
      correspondence "Right(s_rep) = Reverse(s)"
    end Stack

    procedure Push
      parameters
        alters s : Stack
        consumes x : Item
      end  parameters
      performance "O(1)"
      begin
        Add_Right (s,x)
    end Push

    procedure Pop
      parameters
        alters s : Stack
        produces x : Item
      end  parameters
      performance "O(1)"
      begin
        Remove_Right (s,x)
    end Pop
```

Figure 5

Realization Stack_Real_1 of Stack_Template Using One_Way_List_Template

Figure 5 (continued)

```
        control Is_Empty
          parameters
            preserves s : Stack
          end parameters
          performance "O(1)"
          begin
            if At_Right_End(s) then
              return yes
            else
              return no
            end if
        end Is_Empty
    end interface

    description
      ...
    end description

end Stack_Real_1
```

A realization does not exist in isolation, but as a realization of a particular conceptualization. The realization heading indicates the conceptualization that is being realized as well as the name of the realization. For example, the realization presented in Figure 5 is for conceptualization Stack_Template, and is called Stack_Real_1.

Some of the text in a realization is identical to corresponding text in the conceptualization, for example operation names and formal parameters. These portions of a realization could easily be included by the editing environment as unmodifiable text. In the realizations presented in this dissertation, text included from the conceptualization is italicized.

Every name defined in a conceptualization is available for use within a realization of it. For example, Item is defined as a formal type parameter to conceptualization Stack_Template, and is therefore implicitly defined as a type within realization Stack_Real_1. Similarly, String_Theory is defined as a mathematical theory within Stack_Template, and is known within Stack_Real_1. Every name in the scope of the conceptualization is in the scope of its realization.

In addition to definitions made in the conceptualization, realizations also define many things themselves. In the next sections realization Stack_Real_1 is discussed in detail, with the intent of demonstrating the flavor of RESOLVE realizations by presenting a rather simple example first. Advanced features of realizations are presented in Section 3.7.

3.2.2 Conceptualization Auxiliary Section

It is possible that the mathematical functions provided to a realization from the conceptualization are not sufficient, or some of the definitions provided by the conceptualization may not be convenient. For example, a realization may need mathematical theories in addition to those defined in the conceptualization, or a realization may want to rename a function from the conceptualization for convenience.

Conceptual declarations are made in the **conceptualization auxiliary** section of a realization. Declarations in this section do not replace or override declarations made in the conceptualization. Rather, they add to those in the conceptualization.

For example, the correspondence assertion for type Stack in realization Stack_Real_1 uses the definition of string reversal, defined in math facility String_Theory in conceptualization Stack_Template. This assertion could have been written as "`Right(s_rep) = String_Theory.Reverse(s).`" However, it is somewhat more convenient to rename the definition of string reversal from String_Theory as Reverse, which is accomplished in the conceptualization auxiliary section of Stack_Real_1. Note this renaming is not done in conceptualization Stack_Template since the specification does not use string reversal, and renaming it there is unnecessary (although it would have been legal).

3.2.3 Realization Auxiliary Section

The **realization auxiliary** section contains declarations necessary to explain the implementation of types and operations defined by the conceptualization. (Note the parallel between this and the auxiliary section of a conceptualization, which contained declarations of items necessary for the *specification* of types and operations defined by the conceptualization.) This section contains instantiations of conceptualizations needed

for the realization, as well as declarations of variables and operations local to the realization.

Before we go any further with this discussion, let's review the definitions of some terms. A realization that makes use of a conceptualization must *instantiate* that conceptualization. The instance of a conceptualization is called a *facility*, and the realization is said to be a *client* of that conceptualization. The phrases "declare a facility" and "instantiate a conceptualization" are synonymous.

All facilities are declared within the **facilities** subsection of the realization auxiliary section of a realization. A facility declaration states the name of the facility along with the name of the conceptualization being instantiated and the name of a realization of that conceptualization. Actual parameters must also be provided for all formal parameters to the conceptualization and realization.

For example, realization Stack_Real_1 represents stacks using one-way lists, so it instantiates conceptualization One_Way_List_Template. Facility List_Facility is declared as an instance of One_Way_List_Template using List_Real_1 as the realization. (It is assumed here that List_Real_1 is a realization of One_Way_List_Template.) Type Item, which is the formal parameter to conceptualization Stack_Template representing the type of item contained in the stack, is passed as the actual parameter to One_Way_List_Template. Thus List_Facility is a facility exporting the type one-way List of Items.

An instance of a conceptualization makes available to the client all types and operations specified in the interface section of the conceptualization, as well as all math names declared in the auxiliary section of the conceptualization[29]. Referencing any of these is accomplished by prefixing the local name by the facility name followed by a dot. For example, List_Facility.List is the name of the type provided by the instance of

---

29 The instance also provides names for all formal parameters. For example, List_Facility.Item is a name for the parameter to One_Way_List_Template in the instantiation that creates List_Facility, and is also the name of the type of the second parameter to operations List_Facility.Add_Right and List_Facility.Remove_Right. Normally, these formal parameter identifiers provided by the instance are not referenced in the client, since it is much easier to simply reference the actual parameter (e.g., Item). However, this is important for the discussion of type equivalence in Section 3.4.

One_Way_List_Template in Figure 5. This naming scheme is necessary to disambiguate identically named items provided by two different instances[30].

For convenience, it is possible to alias any "dotted name" to a more descriptive identifier. This aliasing is described in a renaming section of the facility declaration. For example, List_Facility.List is renamed as List, so identifiers List and List_Facility.List both stand for the type provided by facility List_Facility. Similarly, the definition List_Facility.Right is renamed Right.

It is important to understand that facilities are static entities defined at compile-time, as opposed to dynamic entities created at run-time. All facilities exist for the entire execution of the program. It is not possible to declare a facility within an operation, have that facility come into existence only when that operation is invoked, and disappear when the operation returns.

3.2.4 Interface Section

The **interface** section of a realization contains the data structures and code that implement the types and operations defined in the interface section of the conceptualization. The types and operations defined in the interface section of a realization are exactly those types and operations specified in the interface section of the conceptualization[31]. For example, the interface section of Stack_Real_1 contains the representation for type Stack, and implementations of operations Push, Pop, and Is_Empty.

3.2.4.1 Type Representations

In realization Stack_Real_1, a Stack is represented by a List of Items, which is simply type List_Facility.List (renamed List). This is indicated in the first line of the type declaration for Stack.

The remainder of the declaration for type Stack specifies exactly how a one-way list is used to represent a stack. In this realization, the items on a stack are contained in the

---

30 For consistency, this naming scheme is enforced even if all names provided by the facilities are unambiguous without being qualified with a facility name.

31 For this reason, much of the text appearing in the interface section can be automatically included in a realization by the editing environment.

right portion of a one-way list, with the top item on the stack immediately to the right of the fence in the one-way list. For example, a stack of integers whose model is the string <4 2 7 5> (with 5 as the topmost item) would be represented by the one-way list <‡5 7 2 4>. In other words, the right string of a one-way list is the reverse of the string that models the stack. This correspondence between the representation of a type and its abstract model is stated formally in the correspondence clause of the type declaration. In this clause, the identifier s_rep is an exemplar denoting a list used to represent a stack, and s is an exemplar (defined in the conceptualization) denoting the abstract stack.

The conventions clause in a type declaration describes an invariant that is guaranteed to be true before and after any operation invocation. In this example, the conventions clause states that the left portion of any one-way list representing a stack will always be the empty string. Conventions and correspondence clauses play a crucial role in formal verification of the realization, as discussed in [Krone 88].

Recall from the discussion in Section 3.1.4 that every type has an associated initial value described as part of the type specification in the conceptualization. In the case of stacks, every variable of type Stack is initially an empty stack. What one-way list represents an empty stack? The conventions and correspondence clauses from the declaration of type Stack indicate that an empty list represents an empty stack. Since the initial value of a one-way list (as defined in One_Way_List_Template in Figure 4) is the empty list, nothing need be done to an initial one-way list for it to represent an empty stack. However, this is not the case in general, and as discussed in Section 3.7.2, it is generally necessary to have code within the type declaration that creates the representation for an initial value.

Similarly, as discussed in Section 3.7.2, it may be necessary to have code that finalizes a variable, releasing memory occupied by its representation. In the case of stacks, finalizing the one-way list representing the stack is sufficient, and no code is explicitly required within the type declaration to accomplish this.

3.2.4.2 Operation Implementations

Implementation of the stack operations is straightforward. The operation name and formal parameter list is simply duplicated from the conceptualization. The performance

clause is an assertion indicating performance characteristics of the operation's implementation, and is typically in "big-O" notation. Here, all operations take a constant amount of time to execute, assuming the one-way list operations realized by List_Real_1 execute in constant time.

The code implementing an operation is introduced by the keyword begin. Procedure Push simply inserts the item being pushed into the one-way list immediately to the right of the fence. Pop removes and returns the item immediately to the right of the fence. Is_Empty returns yes if the fence is at the right end of the one-way list, and returns no otherwise.

It is important to note that every variable (and formal parameter) of type Stack is considered a variable of type List within the code portions of realization Stack_Real_1. This is the reason there isn't a type compatibility problem when a Stack is passed as an actual parameter to a one-way list operation. Type equivalence is discussed in Section 3.4.

3.2.5 Summary

In this section a simple implementation of conceptualization Stack_Template was presented — one that represents stacks using one-way lists. Although the realization seems trivial, it demonstrates two important features of RESOLVE — the relationship between a conceptualization and a realization of it, and the relationship between a conceptualization and a client of it. This example also demonstrates that a facility is an instance of a conceptualization with actual values bound to formal conceptualization parameters and a specific realization chosen. A conceptualization must be instantiated in order for a client to reference the types and operations specified by the conceptualization.

3.3 Data Movement and Control Structures

The code implementing the operations in realization Stack_Real_1 in Figure 5 is very simple, consisting solely of operation invocations and one if statement. Although RESOLVE has a minimal set of control and data movement primitives, this example obviously does not demonstrate its entire repertoire! This section discusses control structures and data movement in RESOLVE.

The control structures defined in RESOLVE are those found in most modern block-structured languages, such as Ada and Pascal. Specifically, RESOLVE's control structures include an if statement for alternation, a while statement for iteration, a procedure invocation statement, and a return statement to return to the invoker from an operation. In addition, the function assignment and swap statements each effect the movement of data.

Data movement in RESOLVE is discussed in Section 3.3.1, along with operation invocation, assignment, and swap statements. The if and while statements are discussed in Section 3.3.2, and the return statements are discussed in Section 3.3.3.

3.3.1 Swapping — RESOLVE's Data Movement Primitive

One of the fundamental actions performed during execution of a program is the movement of data within the execution environment. In modern imperative languages, this movement occurs as a direct result of assignment statements and procedure invocations. As discussed in Section 2.5.2, these generally involve making *copies* of data, which is inherently inefficient.

The problems inherent to copying data are addressed quite simply by RESOLVE — namely, copying is not defined as a built-in operation. Instead, *swapping* the values of two variables is the *only* data movement primitive. This is a rather radical approach, and is one of the most unique and interesting features of RESOLVE.

It may not be intuitively obvious that swapping is indeed powerful enough to warrant its inclusion as the sole data movement primitive in the language. The justification for this decision is presented in the following subsections.

3.3.1.1 The Swap Statement

The first embodiment of the swap primitive is the swap statement. The BNF description of this statement is:

```
<SWAP-STMT>:   <VAR-NAME> :=: <VAR-NAME>
```

When a swap statement is executed, the obvious happens — the values of the two variables are exchanged. For example, assume variables x and y are integer variables, and that x = 5 and y = 10. After executing the statement "`x :=: y`" we have x = 10 and y = 5.

No restrictions are placed upon the types of variables that can be swapped, except that the two variables must be the same type. For example, it is perfectly legal to swap two stacks of integers, but it is not legal to swap an integer with a character.

3.3.1.2 Swapping Is Efficient

One justification for including swapping as the only data movement primitive is that it is very efficient to implement. In fact, it can always be executed in constant time. In other words, swapping two stacks each containing a million elements takes no more time than swapping two integers.

At first glance this doesn't seem possible. However, all that is needed is a well-known implementation trick. The important thing to keep in mind is that each variable *appears* to always have a value from the domain of its type. From the programmer's viewpoint, the swap statement exchanges the values of two variables, as shown in Figure 6.

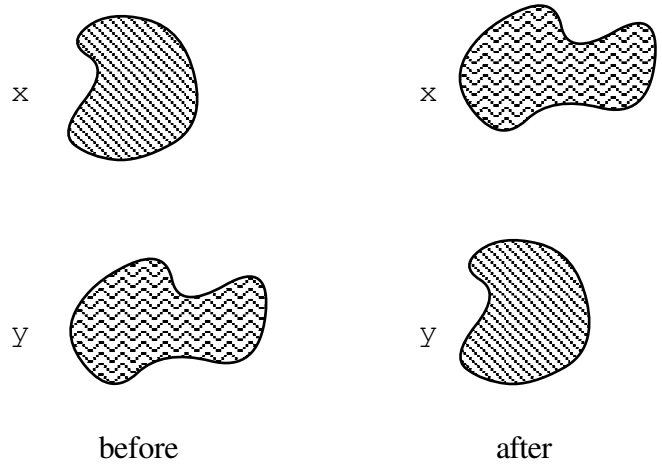

Figure 6

Abstract Effect of Swap Statement "x :=: y"

It is possible to represent each variable as a pointer to a data structure that represents its value[32]. This pointer is part of the implementation of RESOLVE, and is completely invisible to the programmer. Figure 7 describes the action of the swap statement "`x :=: y`" from the implementation (i.e., run-time environment) viewpoint. It is apparent from this figure that swapping the values of two variables simply involves swapping two pointers. The time required to swap these pointers is independent of the sizes of structures they point to. Therefore swapping two variables is a constant-time operation. Representing variables in this way has other advantages, which are discussed in Section 3.8.3.

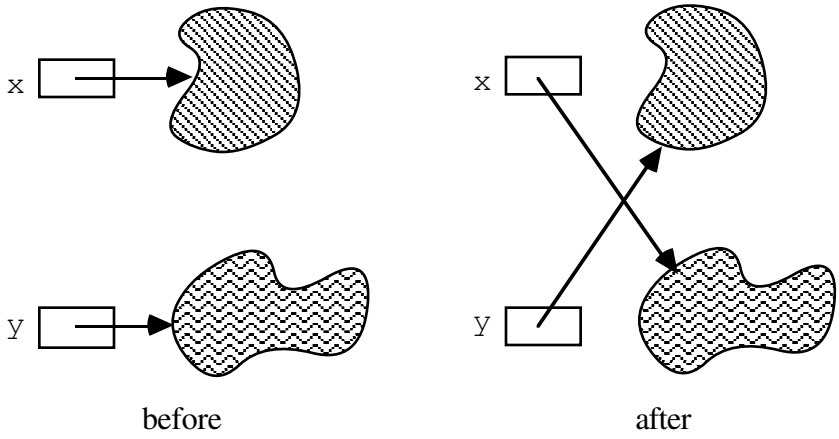

Figure 7

Implementation of Swap Statement "x :=: y"

3.3.1.3 Operation Invocation and Parameter Passing

Passing information between an invoker and an operation via parameters also involves the movement of data. In RESOLVE this is also accomplished by swapping. To see how this works, it is important to understand the role of formal and actual parameters, and the action taken by the RESOLVE run-time environment when an operation is invoked.

It is also imperative that parameter modes (i.e., consumes, produces, alters, and preserves) are not confused with parameter passing mechanisms. Recall from Section

---

[32] Actually, if the data structure representing a value fits into the memory required for a pointer, the pointer is not necessary. For example, integers might be implemented without pointers.

3.1.6 that parameter modes describe how each parameter is used in the exchange of information between an invoker and an operation. They also help streamline pre- and post-conditions. Parameter modes do *not* describe a mechanism for this exchange of information. In RESOLVE, swapping is the mechanism used in the exchange of information between invoker and operation, regardless of parameter mode. Thus, RESOLVE's parameter passing mechanism is "call-by-swapping."

For this discussion, let's assume that all actual parameters are variables (i.e., not function invocations). In fact, this is the restriction placed upon actual parameters in the current version of RESOLVE, although it would not be difficult to relax this somewhat.

When an operation is invoked, the formals may be assumed to have unknown values of their types. Upon invocation, the actual parameters are swapped with corresponding formal parameters. This swapping occurs sequentially, but the order is not specified (i.e., one may *not* assume that parameters are swapped left to right). See Figure 9b. After the parameters have been exchanged the code for the operation is executed. When the operation returns, actual parameters are again swapped with their corresponding formal parameters. See Figure 9d.

Figures 8 and 9 demonstrate the effect of call by swapping by showing the contents of actual and formal parameters at critical times during the operation invocation sequence. In these figures it is assumed that zero is the initial value for type Int.

```
procedure proc
  parameters
    consumes a : Int
    produces b : Int
    alters c : Int
    preserves d : Int
  end  parameters
```

(a) Definition of Procedure Proc

```
proc (w,x,y,z)
```

(b) Client Invocation of Proc

Figure 8

Definition and Invocation of Sample Procedure

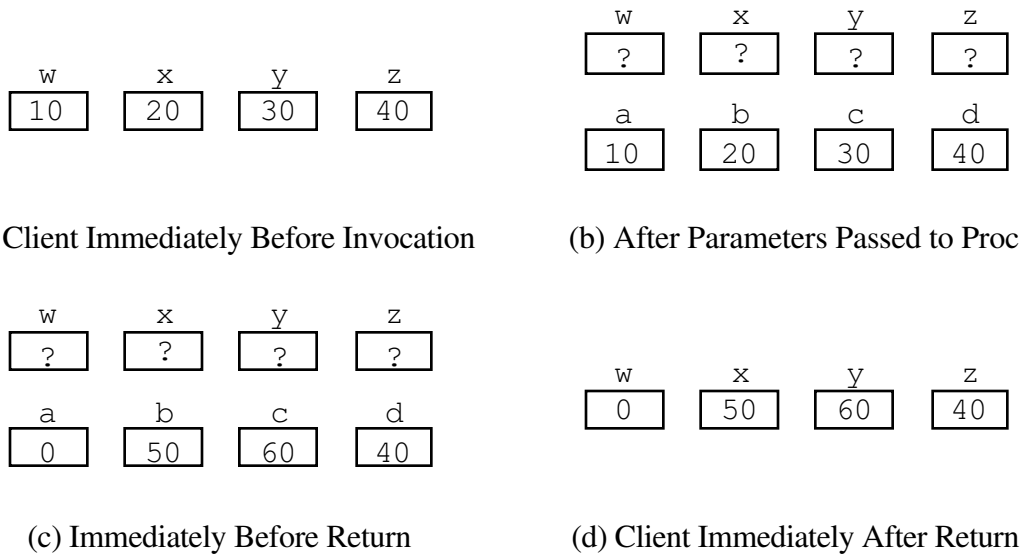

Figure 9

Effect of Sample Procedure Invocation

Note that operation call/return overhead in RESOLVE is constant, since all parameters are passed by swapping, and swapping is a constant-time operation. This is a significant performance advantage over some traditional parameter passing mechanisms such as call by value and call by value-result.

A restriction is placed upon actual parameters in RESOLVE — namely, a variable cannot appear more than once in an argument list. This restriction is necessary because actual parameters are swapped sequentially with formal parameters when an operation is invoked. After an actual parameter is swapped with the corresponding formal parameter, the value of the actual parameter is unknown. If that variable appeared elsewhere in the actual parameter list, its current value (i.e., an unknown value) would be swapped with the corresponding formal parameter, which is most likely not the intention. Note that this problem is not unique to swapping, as discussed in Section 2.6.1.

3.3.1.4 The Function Assignment Statement

The assignment statement in RESOLVE places the result of a function invocation into a variable. The BNF description of this statement is:

```
<ASSIGN-STMT>: <VAR-NAME> := <FUNC-CALL>
```

Note that the right-hand side of an assignment statement *must* be a function invocation — specifically, it cannot be a variable. In other words, the assignment statement cannot be used to implicitly copy a variable. (Making copies of variables is discussed in Section 3.3.1.5.) Also, the assignment statement is the only context where a function invocation is allowed, and the assignment target variable can be an actual parameter to the function call (e.g., the assignment "`x := f(x)`" is legal).

When an assignment statement is executed, RESOLVE's run-time environment invokes the function in the manner described in the previous section. The function's return value identifier (e.g., topitem in Figure 3) is initialized like all other local variables of the function. When the function returns, the value in the function's return value identifier is swapped with the value of the assignment target variable, and the return value identifier is finalized. The effect of this is that the original value in the target variable is finalized, and its new value is the return value of the function.

3.3.1.5 Copying a Variable

RESOLVE does not provide the programmer with a built-in feature to make copies of variables. There are two primary reasons for this, as discussed in Section 2.5.2. First, copying is usually expensive. Including it as a language primitive permits (and possibly encourages) programmers to unwittingly make copies of variables, thereby paying an implicit performance penalty. Second, copying a variable demands type-specific code. Since no types are built-in to RESOLVE, the compiler cannot even generate the code to make a copy of something as simple as an integer. The actual representation of integers is known only within some realization! Of course, there are times when it is necessary to make a copy of a variable, and RESOLVE would be quite useless if there were no way to accomplish this.

If it should be possible for a client to make a copy of a variable of a particular type, it is the responsibility of the writer of the conceptualization providing that type (or one that uses it) to explicitly specify an operation that accomplishes this. For example, Figure 10 contains the specification of a function Replica that creates and returns a copy of a variable of type T. The assignment statement "`a := Replica(b)`" would have the effect of the traditional assignment statement "`a := b`."

```
function Replica returns clone : T
  parameters
    preserves x : T
  end  parameters
  ensures "clone = x"
end Replica
```

Figure 10

Specification of Function Replica for Type T

If a conceptualization does not provide an operation that copies a variable of a provided type, there is no direct way to make a copy for that type[33]. In other words, RESOLVE does not require every type have a copy operation.

3.3.2 Ifs, Whiles, and Control Invocations

This section discusses RESOLVE's definition of the two most elementary control structures in any block-structured language — the if and while statements — and the special relationship these statements have with control operations.

---

33 This does not necessarily mean there is no way at all to make a copy for that type. For example, it is possible to make a copy of a stack by popping all items from the stack into a temporary stack, then popping each item from the temporary stack, making a copy of that item, pushing the copy onto the duplicate stack, and pushing the original item onto the original stack. This algorithm relies on the existence of an operation to copy an item, but is independent of the actual representation of a stack. See Sections 3.5.2 and 3.7.1.

The syntax of the if statement is relatively straightforward:

```
<IF-STMT>:     if [ not ] <CTRL_CALL> then
                  <CODE>
               [ else
                  <CODE> ]
               end  if
```

When an if statement is encountered during execution, the control operation called within the if statement is invoked, as discussed in Section 3.3.1.3. Recall from the discussion in Section 3.1.5 that a control operation returns a control state to the invoker by executing either a return yes statement or a return no statement. If the indicated control returns via a return yes statement, the statements following **then** are executed. Otherwise the statements following **else** are executed if present, or execution continues at the statement following **end if** if **else** is not present. If **not** is placed in the if statement, the control state returned by the control is inverted, so that return yes acts as if it were return no, and vice versa.

The BNF description of the while statement is:

```
<WHILE-STMT>:  <LOOP-ASRT>
               while [ not ] <CTRL-CALL> do
                  <CODE>
               end  while

<LOOP-ASRT>:   maintaining <ASSERTION>     |
               ensuring <ASSERTION>
```

The operational semantics of the while statement are what you'd expect — the control operation is invoked (as discussed in Section 3.3.1.3), and the statements within the body of the loop are executed if and only if the control returns via a return yes statement (or return no for a while not statement). This continues until the control returns via a return no statement.

The loop assertion must be present, and formally specifies the effect of the loop. The assertion is either part of a maintaining clause or part of an ensuring clause. If the loop assertion is contained in a maintaining clause, the assertion is a loop invariant. If the assertion is contained in an ensuring clause, it is a post-condition for the loop, and relates the values variables have before loop execution (denoted by a '#' before the variable name, e.g., #x) to the values they have after loop execution (denoted by the

variable name, e.g., x). The loop assertion has no effect upon the execution of the loop, but is used for verification, as described in [Krone 88].

The syntax and semantics of both statements are similar to most if and while statements. However, a control invocation determines the action taken. Indeed, controls can only be invoked within these two contexts. The motivation for this scheme is simple — the complete separation of data (and data types) from control structures, discussed in Section 2.5.3.7. RESOLVE control structures are defined independently of all data types (even type "boolean"), meaning RESOLVE can be completely defined without any types built-in to the language.

3.3.3 Return Statements

The return statement does the obvious, namely passes control from an operation back to its invoker. Actual and formal parameters are also swapped when an operation returns, as discussed in Section 3.3.1.3. There are actually three forms of the return statement in RESOLVE:

```
<RETURN-STMT>: return        |
               return  yes   |
               return  no
```

Use of the simple return statement is permitted only within procedure and function operations (i.e., it is not permitted within control operations). When return is executed, parameters are swapped as discussed in Section 3.3.1.3, and control passes back to the operation's invoker. Note that the return value of a function is contained in the function's return value identifier, and is *not* specified as part of the return statement.

Use of return yes and return no statements is permitted only within control operations (i.e., they are not permitted within procedure or function operations). When executed, parameters are swapped with the invoker as discussed in Section 3.3.1.3, and control passes back to the control's invoker (which must be in an if statement or while statement). As discussed in Section 3.3.2, the action taken by the if or while statement is determined by which return statement the control executed.

Every procedure and function has an implicit return statement at the end of the code, so an explicit return statement is not necessary. For example, see Figure 5. On the other

hand, control operations do not have an implicit return statement. Reaching the end of a control without executing either a return yes or return no statement is illegal.

3.3.4 Summary

In this section data movement within RESOLVE programs was discussed, as well as the control structures defined by RESOLVE. Data movement is accomplished by swapping the values of two variables. The swap and assignment statements explicitly involve data movement (i.e., swapping); in addition, all parameters are passed between an invoker and an operation by swapping actual with formal parameters.

Copying the value of a variable is not a built-in operation in RESOLVE, but is an operation specified in a conceptualization, just like all other operations except swapping. RESOLVE does not require that every type be copyable (i.e., a conceptualization does not have to specify a copy operation); thus, it may not be possible to copy some variables.

The control structures defined by RESOLVE are the if statement, while statement, operation invocation, and return statement. The definitions of these control structures are straightforward. This is a minimal set of control structures compared to RESOLVE's ancestors such as Pascal and Ada. However, these control structures are sufficiently powerful and easy to understand.

3.4 Types and Type Equivalence

RESOLVE is a statically-typed language, meaning that every variable has a type that is fixed and known at compile time. As discussed in Section 2.5.3, the purpose of types is to check that variables are used in legal contexts, e.g., that the two variables in a swap statement have the same type. If a variable appears in a context that is illegal for its type, that statement is simply invalid, and no meaning is defined for it.

But what precisely are types in RESOLVE, what is the relationship between program types and mathematical types, and what does it mean for two types to be equivalent? This section addresses these questions and discusses the relationships among types, domains, and module instances.

3.4.1 Domains

In RESOLVE a *math domain* is an anonymous set of anonymous values defined by an instance of a *theory*. It is defined implicitly by the axioms of the theory. For example, instance Number_Theory of Number_Theory_Template in Figure 15 provides the axioms defining the domain of integers, and instance String_Theory of String_Theory_Template in the same figure provides the axioms defining the domain of strings over integers.

Similarly, a *program domain* is defined by an instance of a *conceptualization*. Each element in a program domain is modeled by some element in a corresponding math domain. The elements in a program domain are those values that are "reachable" by executing the operations defined in the conceptualization. For example, instance List_Facility in Figure 5 defines a program domain whose elements are modeled by 2-tuples of strings. The elements in this domain are exactly those elements whose models are reachable by executing all possible combinations of one-way list operations provided by List_Facility.

When a facility is declared, the instantiated theory or conceptualization is chosen from a library of theories and conceptualizations. Even though these libraries contain a finite number of items, there are an infinite number of possible instances that can be created from them, due to the fact that many of the theories and conceptualizations are generic. For example, it is possible to declare a math facility providing a domain for integer, one providing a domain for strings of integers, another providing a domain for strings of strings of integers, still another providing a domain for strings of strings of strings of integers, etc.

Given a library of available theories, there exists an infinite collection of math domains provided by all possible instantiations of these theories, shown as set d_m in Figure 11. Similarly, given a library of available conceptualizations and realizations for them, there exists an infinite collection of program domains defined by all possible instantiations, shown as set d_p in Figure 11. Every math domain defined by a math facility is an element of set d_m, and every program domain defined by a program facility is an element of set d_p.

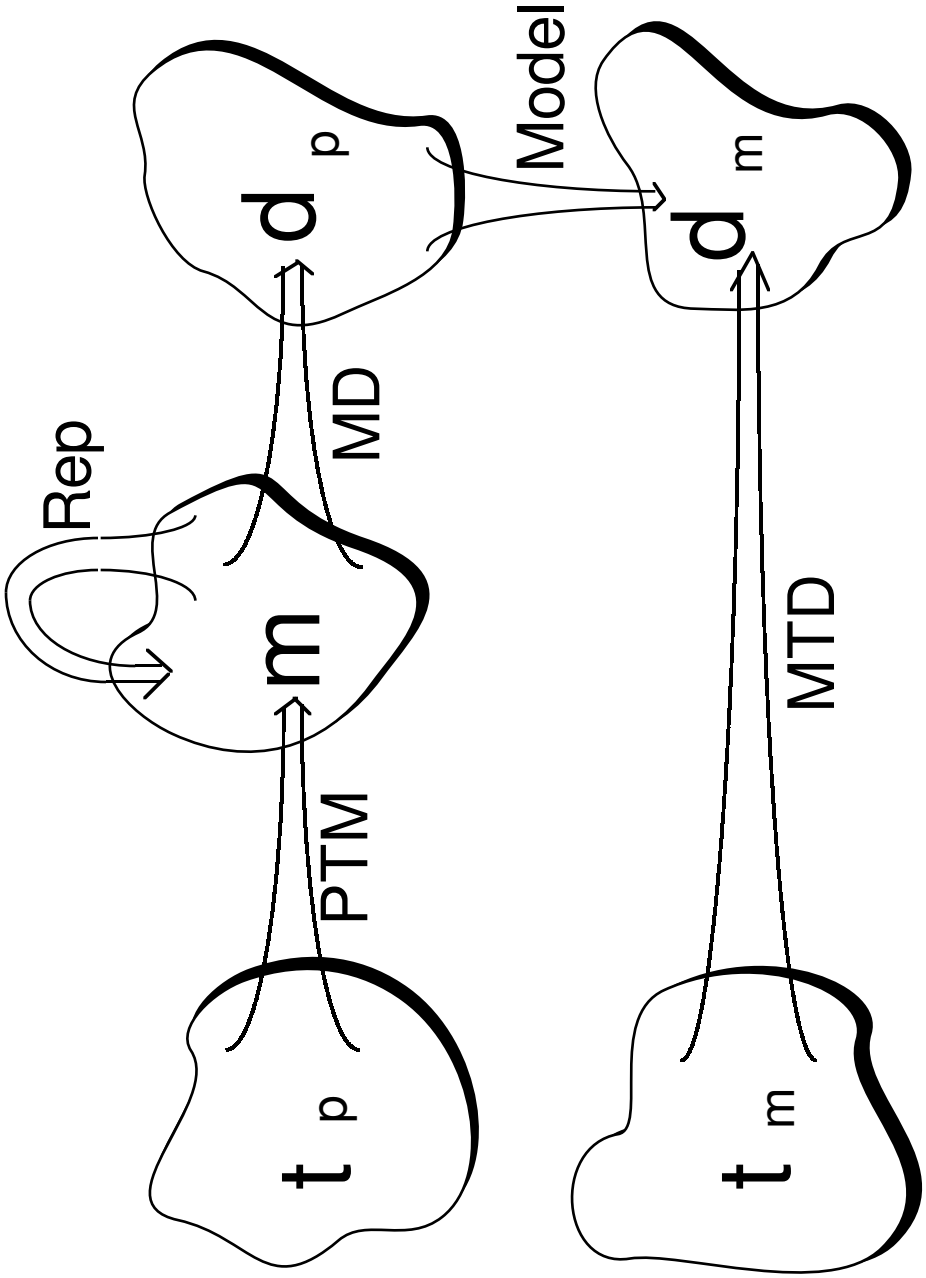

Figure 11
Relationship Between Types, Markers, and Domains

Total function $\mathsf{Model}:\mathsf{d}_p \rightarrow \mathsf{d}_m$ indicates the math domain that models each program domain, and is shown as a hollow arrow in Figure 11. This mapping for a particular program type is specified in the "is modeled by" clause of the type declaration within a conceptualization. This function is neither one-to-one nor onto, because more than one program domain may be modeled by a particular math domain, and there exist math domains that do not model any program domain.

3.4.2 Math Types

When a client module instantiates a theory, every math domain defined by the resulting math facility is given a name that is unique within the client. A *math type* is simply the name given to a math domain by a client. Each client has an associated set containing all math types defined within it, shown as set t_m in Figure 11.

Total function $\mathsf{MTD}:\mathsf{t}_m \rightarrow \mathsf{d}_m$ (which stands for Math Types-to-Domain) indicates the math domain associated with each math type. This function is not one-to-one since several math types may name the same math domain.

As an example of how the MTD mapping is created for a particular client, consider the facilities section of a realization shown in Figure 12. The mappings between the various sets is shown in Figure 13. (Conceptualizations for Stack_Template and Bounded_Integer_Template are presented in Figures 2 and 21, respectively.) Here, two math domains are referenced — integers and strings of integers. However, there are seven math types declared that name these math domains.

```
facilities
  Int_Facility1 is Bounded_Integer_Template
    realized by Standard_Int_Real
    renaming
      Int_Facility1.Int as Int
    end  renaming

  Int_Facility2 is Bounded_Integer_Template
    realized by Standard_Int_Real

  IntStack_Facility is Stack_Template (Int)
    realized by Stack_Real_1
    renaming
      IntStack_Facility.Stack as Stack
    end  renaming
end  facilities
```

Figure 12

Example Type Declarations

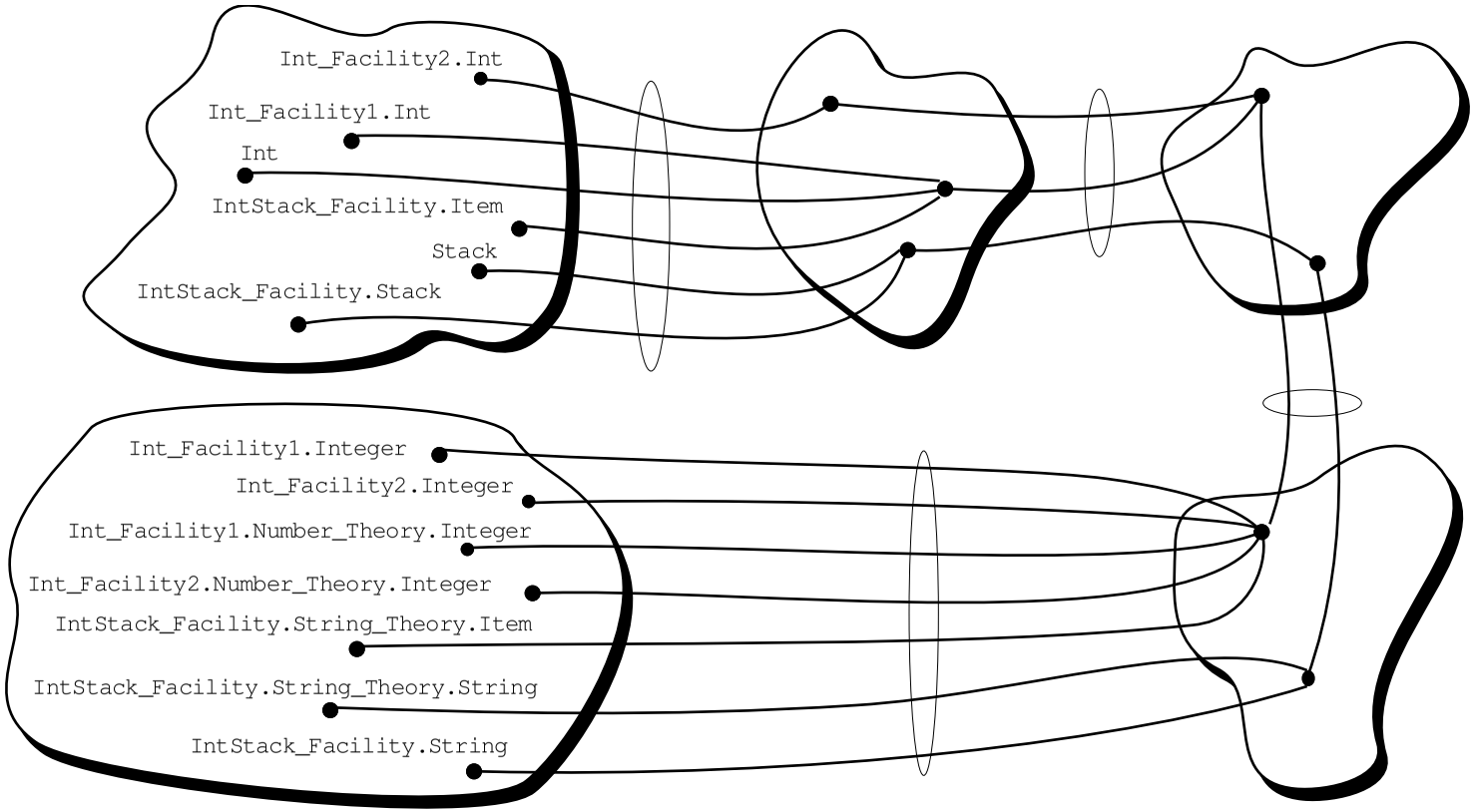

Figure 13
Type Mappings for Example Type Declarations

3.4.3 Program Types and Markers

As with math types, *program types* are simply names of things. However, because of type equivalence issues discussed shortly, program types do not directly name program domains, but instead name *markers*. When a conceptualization is instantiated by a client, a marker is created for each program domain defined. A marker is also created for each type parameter to the client and for each type provided by the client. Each client has an associated set containing all its markers, shown as set m in Figure 11.

A program type is the name for a marker, which in turn designates a program domain. Each client has an associated set containing all program types declared in it, shown as set $\mathsf{t_p}$ in Figure 11.

As an example, consider the facilities section of the client shown in Figure 12, with the corresponding mappings shown in Figure 13. Three markers are declared — for Int provided by Int_Facility1, Int provided by Int_Facility2, and Stack (of Int) provided by IntStack_Facility — with six names for them, which are the program types known in this client.

Total function $\mathsf{PTM}{:}\mathsf{t_p} \rightarrow \mathsf{m}$ (which stands for Program Types-to-Markers) indicates the marker associated with each program type. This function is onto, because each marker has at least one name. This function is not one-to-one, because a particular marker may have more than one name associated with it, as demonstrated in Figures 12 and 13.

Total function $\mathsf{MD}{:}\mathsf{m} \rightarrow \mathsf{d_p}$ (which stands for Markers-to-Domains) indicates the program domain associated with each marker. Markers created for identically-declared facilities map to the same program domain, as demonstrated by Int_Facility1 and Int_Facility2 in Figures 12 and 13. Because of this MD is not one-to-one. Of course, it is not onto because m is the finite set of markers for a particular program module, while $\mathsf{d_p}$ is the infinite set of all possible program domains creatable from the entire library of conceptualizations.

Partial function $\mathsf{Rep}{:}\mathsf{m} \rightarrow \mathsf{m}$ is defined only on the markers whose program types are provided by the client, and indicates the marker used to represent the provided type. This function is defined by the “is represented by” clause of a type declaration within a

realization. For example, Figure 14 shows the sets and mappings for realization Stack_Real_1 from Figure 5. Here, function Rep shows that provided type Stack is represented by type List.

Figure 14 also demonstrates two other important characteristics of type mappings. First, it demonstrates the mapping of type parameters (e.g., Item) and generic types (e.g., Stack and List). In the case of a type parameter the formal parameter maps to a point in m, but the mapping of this point to d_p (and d_m) is unknown, as indicated by question marks. Similarly, the name of a generic program type maps to a point in m, but this point maps to an unknown point in d_p and d_m. Although the mathematical model of a generic program type is unknown, a math type can be assigned to it, indicated by function MTD mapping t_m to the same unknown point in d_m. For example, Figure 14 indicates that program type Stack is modeled by math type String (among others, all of which are names for the same math domain).

Second, Figure 14 demonstrates that all math types defined by identical instances of a theory map to the same point in d_m. For example, String (defined by the instance of String_Theory_Template in conceptualization Stack_Template) and List_Facility.String_Theory.String (defined in the instance of One_Way_List_Template) map to the same point in d_m (as do many other names).

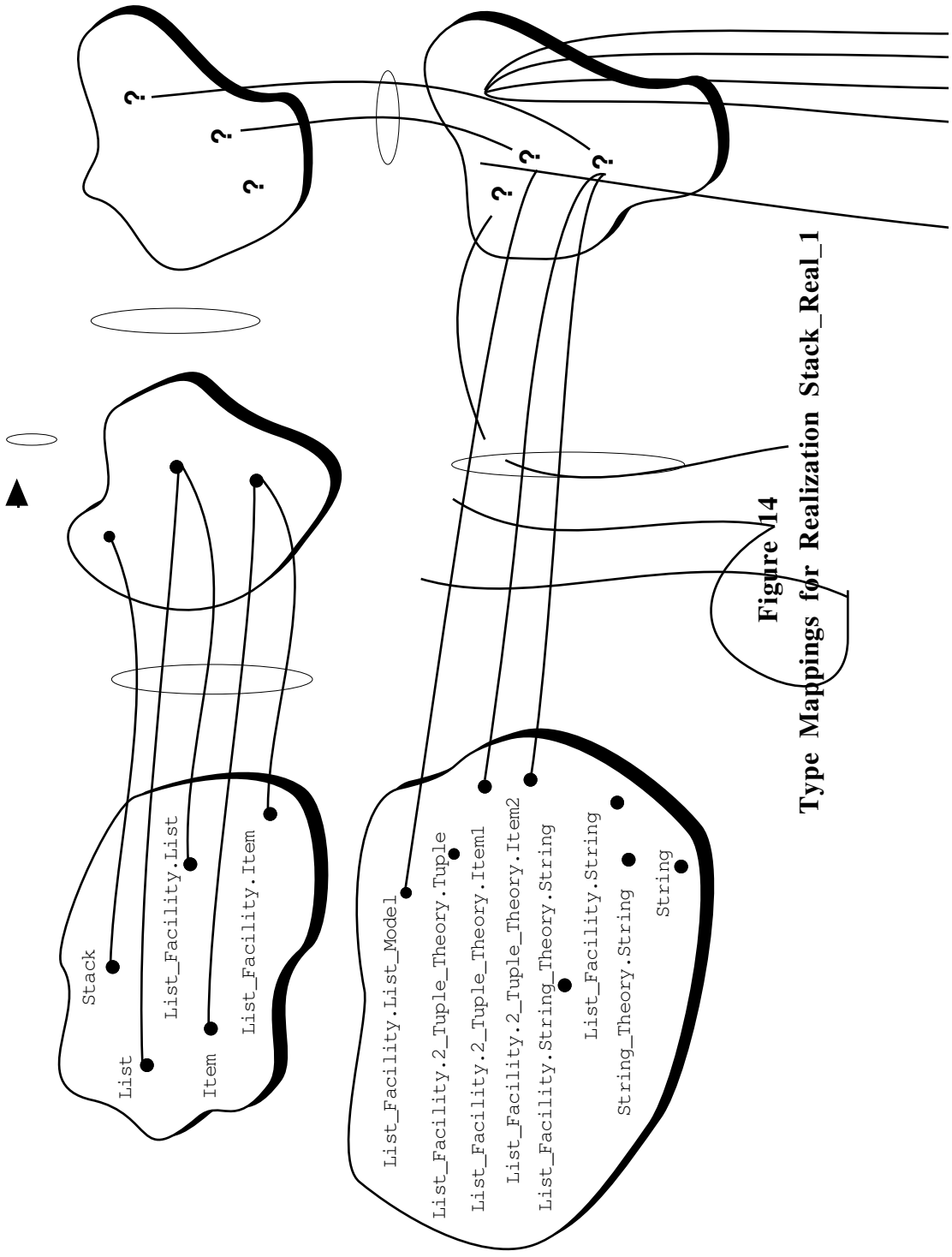

Figure 14

Type Mappings for Realization Stack_Real_1

3.4.4 Type Equivalence

Using the definition of types presented in the previous section, type equivalence is a matter of determining if two types map to the same point in some set. This may seem rather straightforward (and it is), yet some interesting issues are raised, which are discussed here.

The obvious question arising from this approach to type equivalence is which set to use to determine type equivalence. For math types there is only one choice — set d_{m}. Thus, type equivalence for math types t1 and t2 is defined as:

```
t1 ≡ t2 ⇔ MTD(t1) = MTD(t2)
```

For example, types List_Facility.String and String_Theory.String in Figure 14 are equivalent, since they both map to the same point in DM. However, neither of these types is equivalent to List_Facility.List_Model, since it maps to a different point.

The mathematical model of a program type t, written **math**[t], is defined implicitly by the following:

```
MTD(math[t]) = Model(MD(PTM(t)))
```

It is thus legitimate to talk about type equivalence between the mathematical model of a program type and other mathematical types. For example, the mathematical model of program type Int_Facility2.Int in Figure 13 is equivalent to the mathematical model of program type Int. Similarly, the mathematical model of program type Stack in Figure 14 is equivalent to mathematical type List_Facility.String. However, the mathematical models of program types Stack and List in Figure 14 are not equivalent.

Type equivalence of program types t1 and t2 is defined as:

```
t1 ≡ t2 ⇔
     PTM(t1) = PTM(t2) ∨
     Rep(PTM(t1)) = PTM(t2) ∨
     PTM(t1) = Rep(PTM(t2))
```

In other words, two program types are equivalent iff they both map to the same marker, or if one of the types is represented by the other. For example, program type Int in Figure 13 is equivalent to type Int_Facility1.Int, but is not equivalent to type

Int_Facility2.Int. Similarly, program type List in Figure 14 is equivalent to type List_Facility.List, and is also equivalent to type Stack.

The justification for defining program type equivalence with markers rather than program domains is based partly upon our design philosophy that realizations determine performance characteristics of the operations and do not affect the contexts where those operations can be legally invoked. In other words, changing the realization of a facility should not alter the syntactic correctness of the module. Note that changing the realization of a facility does not alter function PTM, yet it does alter function MD because realizations are part of the definition of elements of $\mathcal{D}_p$. Thus, if program type equivalence were defined in terms of $\mathcal{D}_p$, changing a realization of a facility could potentially alter the syntactic acceptability of the module, which is not a problem when defining program type equivalence in terms of markers.

3.4.5 Influence on Compiler Implementation

The sets and functions defined in the previous sections are entirely abstract, and their use was solely for the purpose of defining types and type equivalence. However, this framework is also very useful in implementing the symbol tables for a RESOLVE compiler, which is discussed in this section.

Sets $\mathcal{D}_m$ and $\mathcal{D}_p$ are infinite, so it is not possible to actually represent them within the symbol table. Because $\mathcal{D}_p$ is not used in determining type equivalence though, the symbol table need not represent any elements from it. However, mathematical type equivalence is defined in terms of elements of $\mathcal{D}_m$, so a subset of it must be maintained during compilation.

The compiler assigns a tag to each element of DM that is referenced in the client it is compiling. When a mathematical type is defined in that client by instantiation of a theory, the compiler determines the type's signature, which includes the type identifier and all actual parameters of the math facility. If the symbol table already contains a math type with that signature, the new math type is assigned the same tag as the math type already in the symbol table. Otherwise the new math type is assigned a new tag and is added to the symbol table. The tag of a type is also assigned to all types that rename it. Two mathematical types are equivalent iff their tags are the same.

In this scheme, the "identifier" fields within the symbol table represent set t_m, the tags are elements of d_m, and the "tag" fields within the symbol table represent function MTD.

A similar approach can be used for program types, with the compiler basically constructing set m for the client it is compiling. An unique marker is assigned to each formal type parameter of the client (e.g., Item in Figure 14), to each type provided by the client (e.g., Stack in Figure 14), and to each program type provided by instances of conceptualizations within the client (e.g., List in Figure 14). A type that is a formal parameter to an instantiated conceptualization (e.g., List_Facility.Item in Figure 14) is assigned the marker of the actual parameter. The marker of a type is also assigned to all types that rename it.

The symbol table includes a "represented by" field for all types provided by the client, which is assigned the marker of the type that represents the provided type. The symbol table also includes a "math model" field which is meaningful for all program types except formal type parameters. This field is assigned the tag of the mathematical type that models the program type.

Two program types are equivalent iff they have the same marker, or the marker of one is the "represented by" marker of the other.

In this scheme, the "identifier" fields within the symbol table represent set t_p, the markers are elements of m, the "marker" fields within the symbol table represent function PTM, the "represented by" fields represent function Rep, and the "math model" fields represent the composition of function MD with Model.

3.4.6 Summary

In this section a domain was defined as an unnamed set of values, and a type as a name given to a domain. A mathematical type was defined by an instance of a theory, while a program type was defined by an instance of a conceptualization. Given a finite library of theories, there is an infinite set d_m that contains all possible math domains definable by these theories. Similarly, given a finite library of conceptualizations, there is a set called d_p that contains all possible program domains definable by these

conceptualizations. Given a module there are sets t_p and t_m that contain respectively all program types and math types defined within that module.

Every program domain (i.e., every element of d_p) has a corresponding math domain that models it. This is formally defined as total function Model that maps elements of d_p to d_m.

Every module has associated with it a finite set m of markers that contains an element for each program domain declared in that module. A total function PTM maps elements of t_p to m. Partial function Rep maps m to m, and indicates the representation of all types provided by a realization module.

Two mathematical types are equivalent if and only if they both map to the same point in d_m. Similarly, two program types are equivalent if and only if both map to the same point in m, or one of the types is represented by the other.

These definitions for type and type equivalence are formal, simple to understand and to implement, and maintain the advantages of static type checking. By contrast, definitions surveyed by [Danforth 88] are complicated and generally do not apply for static typing.

3.5 Conceptual Facility Parameters

Ordinarily, every program type referenced within a conceptualization must be equivalent to some program type in the client. If this were not the case it might be impossible to invoke some operations because the client could not pass an actual parameter of an equivalent type. Thus, all types referenced in a conceptualization are either defined by the conceptualization (e.g., Stack in conceptualization Stack_Template) or somehow passed to the conceptualization via its parameters (e.g., Item in conceptualization Stack_Template).

Type parameters are used to pass types from the client to the conceptualization. The conceptualization cannot place any restrictions upon the type that is passed to it, and likewise it cannot make any assumptions about the type. This mechanism is used to define generic types where the component type is provide by the client.

There are situations, however, where a conceptualization needs the client to pass it a particular type, such as Int provided by an instance of Bounded_Integer_Template.

Facility parameters must be used in these situations. This section presents three examples that demonstrate the need for and use of facility parameters — bounded stacks, copying stacks, and arrays.

3.5.1 Conceptualization Bounded_Stack_Template

Conceptualization Stack_Template in Figure 2 contains a specification for unbounded stacks where no *a priori* limit is placed on the number of items that can be contained in a stack. A different conceptualization of stacks places a finite limit (or bound) on the maximum number of items that a stack can hold. A conceptualization that captures this notion, called Bounded_Stack_Template, is presented in Figure 15.

```
conceptualization Bounded_Stack_Template
  parameters
    type Item
    facility Int_Facility is Bounded_Integer_Template
      renaming
        Int_Facility.Int as Int
      end  renaming
  end  parameters

  auxiliary
    math  facilities
      Number_Theory is Number_Theory_Template
        renaming
          Number_Theory.Integer as Integer
        end  renaming

      String_Theory is String_Theory_Template (math[Item])
        renaming
          String_Theory.String as String
          String_Theory.Lambda as Lambda
          String_Theory.Post as Post
          String_Theory.Length as Length
        end  renaming

      Tuple_2_Theory is Tuple_2_Theory_Template
          (String,Integer)
        renaming
          Tuple_2_Theory.Tuple as Bounded_Stack_Model
          Tuple_2_Theory.Projection_1 as Stack_Items
          Tuple_2_Theory.Projection_2 as Stack_Max_Size
        end  renaming
    end  math  facilities
  end  auxiliary

  interface
    type Bounded_Stack is modeled by Bounded_Stack_Model
      exemplar s
      initially "Stack_Items(s) = Lambda and
                 Stack_Max_Size(s) = 0"
    end Stack

    procedure Set_Max_Size
      parameters
        alters s : Bounded_Stack
        consumes Max_Size : Int
      end  parameters
      requires "Max_Size > 0"
      ensures "Stack_Items(s) = Lambda and
               Stack_Max_Size(s) = #Max_Size"
    end Set_Max_Size
```

Figure 15

Specification for a Module Providing Generic Type Bounded_Stack

Figure 15 (continued)

```
    function Get_Max_Size returns Max_Size : Int
      parameters
        preserves s : Bounded_Stack
      end  parameters
      ensures "Max_Size = Stack_Max_Size(s)"
    end Get_Max_Size

    function Get_Size returns Size : Int
      parameters
        preserves s : Bounded_Stack
      end  parameters
      ensures "Size = Length(Stack_Items(s))"
    end Get_Size

    procedure Push
      parameters
        alters s : Bounded_Stack
        consumes x : Item
      end  parameters
      requires "Length(Stack_Items(s)) < Stack_Max_Size(s)"
      ensures "Stack_Items(s) = Post(Stack_Items(#s),#x) and
               Stack_Max_Size(s) = Stack_Max_Size(#s)"
    end Push

    procedure Pop
      parameters
        alters s : Bounded_Stack
        produces x : Item
      end  parameters
      requires "Stack_Items(s) ≠ Lambda"
      ensures "Stack_Items(#s) = Post(Stack_Items(s),x) and
               Stack_Max_Size(s) = Stack_Max_Size(#s)"
    end Pop

  description
    ...
  end  description

end Bounded_Stack_Template
```

Each bounded stack is modeled as a 2-tuple consisting of a string (referenced with projection function Stack_Items) that contains the items currently in the bounded stack, and an integer (referenced with projection function Stack_Max_Size) that contains the maximum allowable size of the bounded stack. The number of items in a bounded stack at any time is simply the length of the string.

Functions Get_Max_Size and Get_Size return the maximum size and current size of a bounded stack, respectively. Operations Push and Pop accomplish the obvious. Note that a requires clause is specified for procedure Push, whereas one is not specified for procedure Push in conceptualization Stack_Template.

A client of Bounded_Stack_Template is presented in Figure 16, and the corresponding program type mappings are shown in Figure 17. In this example type BStack is a bounded stack of characters, and the size and maximum size of a bounded stack are integers of type Int.

```
  ...
facilities
  Int_Facility1 is Bounded_Integer_Template
    realized by Standard_Int_Real
    renaming
      Int_Facility1.Int as Int
    end  renaming

  Int_Facility2 is Bounded_Integer_Template
    realized by Standard_Int_Real

  Char_Facility is Char_Template
    realized by Standard_Char_Real
    renaming
      Char_Facility.Char as Char
    end  renaming

  Char_Stack_Facility is Bounded_Stack_Template
    (Char,Int_Facility)
    realized by BStack_Real_1
    renaming
      Char_Stack_Facility.Bounded_Stack as BStack
      Char_Stack_Facility.Get_Size as Get_Size
    end  renaming
end  facilities
  ...
local  variables
  i : Int
  j : Int_Facility2.Int
  s : BStack
end  local  variables
  ...
```

Figure 16

Bounded_Stack_Template Client

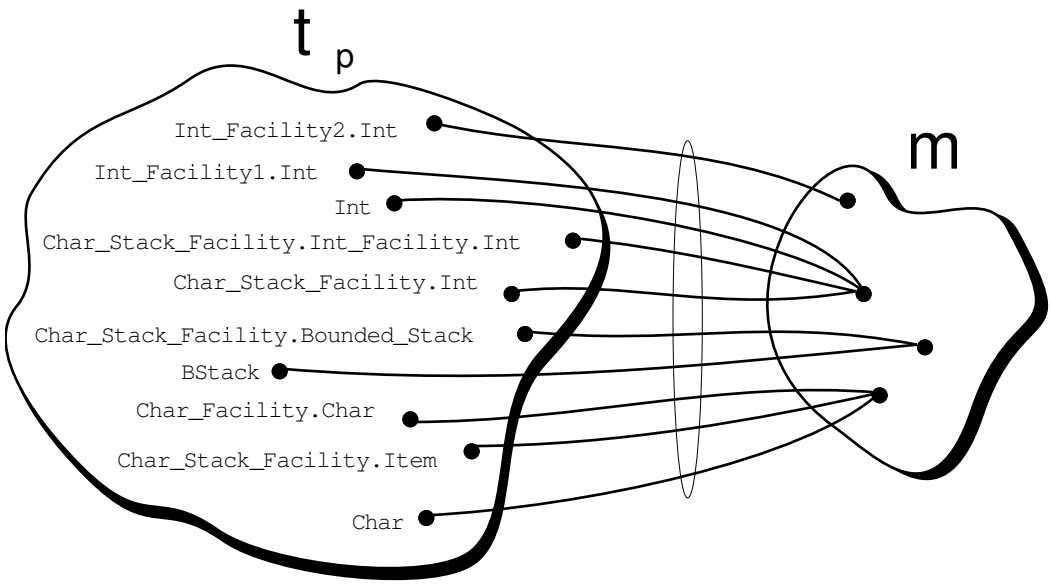

Figure 17

Program Type Mappings for Bounded_Stack_Template Client

In the above client, the type of variable i (i.e., Int) is equivalent to the type of function Get_Size (i.e., Char_Stack_Facility.Int) because both types have the same marker. Therefore, the assignment statement "`i := Get_Size(s)`" is legal. However, the statement "`j := Get_Size(s)`" is illegal, because types Int_Facility2.Int and Char_Stack_Facility.Int are not equivalent.

3.5.2 Conceptualization Copy_Stack_Template

Conceptualization Copy_Stack_Template in Figure 18 defines procedure Copy_Stack that copies one stack into another.

```
conceptualization Copy_Stack_Template
  parameters
    facility Stack_Facility1 is Stack_Template
    facility Stack_Facility2 is Stack_Template

    restrictions
      Stack_Facility1.Item = Stack_Facility2.Item
    end  restrictions
  end  parameters

  interface
    procedure Copy_Stack
      parameters
        preserves s1 : Stack_Facility1.Stack
        produces s2 : Stack_Facility2.Stack
      end  parameters
      ensures "s2 = #s1"
    end Copy_Stack
  end  interface

  description
    . . .
  end  description

end Copy_Stack_Template
```

Figure 18

Specification for a Module Providing Procedure Copy_Stack

The interesting characteristic of procedure Copy_Stack (and in fact the motivation for creating this conceptualization) is that the stack types need not be equivalent. In other words, it is possible to copy a stack into a stack with a different realization. Two facilities are passed to Copy_Stack_Template — the first provides the type of the "source" stack, and the second provides the type of the "destination" stack.

The only restriction placed on the stacks is that the types of items contained in them must be equivalent. This restriction is specified in the restrictions section of the conceptualization parameters. Restriction clauses must be of the form "`type1 = type2`" where type1 and type2 are passed to the conceptualization from the client. Restrictions are enforced by the compiler when the conceptualization is instantiated.

An example client of Copy_Stack_Template is presented in Figure 19.[34]

---

34 The reason for the realization parameter in this example (i.e., Char_Copy) is discussed in Section 3.7.1.

```
  .
  .
  .
facilities
  Char_Facility is Char_Template
    realized by Standard_Char_Real
    renaming
      Char_Facility.Char as Char
      Char_Facility.Copy as Char_Copy
    end  renaming

  Stack_Facility1 is Stack_Template (Char)
    realized by Stack_Real_1

  Stack_Facility2 is Stack_Template (Char)
    realized by Stack_Real_2

  Copy_Stack_Facility is Copy_Stack_Template
    (Stack_Facility1,Stack_Facility2)
    realized by Copy_Stack_Real_1 (Char_Copy)
    renaming
      Copy_Stack_Facility.Copy_Stack as Copy_Stack
    end  renaming
end  facilities
  .
  .
  .
local  variables
  s1 : Stack_Facility1.Stack
  s2 : Stack_Facility2.Stack
end  local  variables
  .
  .
  .
  Copy_Stack(s1,s2)
  .
  .
  .
```

Figure 19

Copy_Stack_Template Client

In this example, procedure Copy_Stack copies a stack realized by realization Stack_Real_1 into a stack realized by realization Stack_Real_2. If the client also needed to copy stacks in the other direction (i.e., from Stack_Real_2 to Stack_Real_1) it would be necessary to declare another instance of Copy_Stack_Template, passing it facilities Stack_Facility2 and Stack_Facility1 in that order.

3.5.3 Conceptualization Array_Template

As a final demonstration of facility parameters, consider conceptualization Array_Template in Figure 20.

```
conceptualization Array_Template
  parameters
    type Item

    facility Integer_Facility is Bounded_Integer_Template
      renaming
        Integer_Facility.Int as Int
      end renaming
  end parameters

  auxiliary
    math facilities
      Number_Theory is Number_Theory_Template
        renaming
          Number_Theory.Integer as Integer
        end renaming

      Function_Theory is Function_Theory_Template
          (Integer, math[Item])
        renaming
          Function_Theory.Function as Integer_To_Item
          Function_Theory.Delta as Delta
        end renaming

      Tuple_2_Theory is Tuple_2_Theory_Template
          (Integer,Integer_To_Item)
        renaming
          Tuple_2_Theory.Tuple as Array_Model
          Tuple_2_Theory.Projection_1 as size
          Tuple_2_Theory.Projection_2 as map
        end renaming
    end math facilities
  end auxiliary
```

Figure 20

Specification for a Module Providing Generic Type Array

Figure 20 (continued)

```
    interface
      type Array is modeled by Array_Model
        exemplar a
        initially "size(a) = 0 and
                   ∀i:Integer, Item.init ((map(a)) (i))"
        lemma "∀i:Integer, (i < 0 or i ≥ size(a)) ⇒
               Item.init ((map(a)) (i))"
      end Array

      procedure Set_Size
        parameters
          alters a : Array
          consumes n : Int
        end  parameters
        requires "n ≥ 0"
        ensures "size(a) = #n and
                 ∀i:Integer, Item.init ((map(a)) (i))"
      end Set_Size

      function Get_Size returns this_size : Int
        parameters
          preserves a : Array
        end  parameters
        ensures "this_size = size(a)"
      end Get_Size

      procedure Access
        parameters
          alters a : Array
          preserves i : Int
          alters x : Item
        end  parameters
        requires "0 ≤ i < size(a)"
        ensures "size(a) = size(#a) and Delta(map(a),{i}) and
                 (map(a)) (i) = #x and x = (map(#a)) (i)"
      end Access
    end  interface

    description
      . . .
    end  description

  end Array_Template
```

Array_Template defines type Array which is a structure encapsulating the notion of quasi-static arrays. A quasi-static array is one where the size is set at execution time. However, once the size is set, it effectively cannot be changed. The indices of the array are the first size non-negative integers (i.e., 0 to size-1).

An Array is formally defined as a pair: the size (called size), and a function (called map) from integers to type Item, the component type of the array. In the abstract, map is a total function, but the requires clause of procedure Access (which is the operation that alters the contents of an Array) restricts the index to be non-negative and less than the array's size. Therefore, function map is meaningful only on this interval.

An Array initially has a size of zero, and its function maps all integers to an initial value of type Item. Procedure Set_Size sets the size of an Array and also resets the Array's function so it maps all integers to an initial value of type Item. In other words, the old contents of the Array are lost when the size is changed. Procedure Get_Size returns the current size of an Array. Procedure Access swaps the previous value of an element of Array with the previous value of parameter x.

Array_Template also introduces two notations — lemmas and deltas. A lemma, such as specified in the specification for type Array, is an invariant (i.e., an assertion that is true at all times) that can be proved using other assertions defined in the conceptualization. For example, the lemma defined in Array states that an Array's function maps all integers less than zero or greater than or equal to size to an initial value of type Item. This is easy to prove, given that an Array's function maps all integers to an initial value of type Item when the Array is created and after invoking Set_Size (by Array's initially clause and Set_Size's ensures clause), and that the only other operation that changes this mapping is Access, which cannot change the mapping of integers less than zero or greater than or equal to size (by Access's requires and ensures clauses).

Math function Delta, defined by theory Function_Theory, is a convenient notation indicating that a mathematical function's mapping potentially changes on only a subset of its domain. Specifically, the following is the definition for Delta, where f is a mathematical function and s is a set of values from f's domain:

```
Delta(f,s) ≡ ∀ i:Domain(f), i ∉ s ⇒ f(i) = #f(i)
```

For example, Access's ensures clause states (among other things) that function map potentially changes only on the element indexed.

3.5.4 Summary

As a consequence of RESOLVE's definition of type equivalence, it is often necessary for a client to pass an actual type to an instance of a conceptualization. Facility parameters accomplish this in RESOLVE, where an actual instance of a specific conceptualization is passed to the conceptualization being instantiated. Three examples of conceptualizations that have facility parameters were presented in this section — bounded stacks, a conceptualization providing an operation to copy stacks, and arrays.

3.6 Conceptual Constants and Variables

Conceptualizations are defined in terms of formal conceptualization parameters and declarations made within the auxiliary section. In all conceptualizations presented so far, the auxiliary sections contained only instances of theories. However, some conceptualizations are defined in terms of constant values defined by the realization or in terms of conceptual variables that contain state information.

This section demonstrates auxiliary constants and variables with three examples — conceptualization Bounded_Integer_Template, conceptualization Single_Link_Ref_Template, and conceptualization ADO_Stack_Template.

3.6.1 Conceptualization Bounded_Integer_Template

Conceptualization Bounded_Integer_Template, presented in Figure 21, defines "computational integers" which are mathematical integers with lower and upper bounds defined.

```
conceptualization Bounded_Integer_Template

  auxiliary
    math facilities
      Number_Theory is Number_Theory_Template
        renaming
          Number_Theory.Integer as Integer
          Number_Theory.Add as Math_Add
          Number_Theory.Sub as Math_Sub
          Number_Theory.Mult as Math_Mult
          Number_Theory.Abs as Math_Abs
        end renaming
    end math facilities

    math constants
      min_int : Integer
      max_int : Integer
    end math constants

    constraint "min_int ≤ 0 < max_int"
  end auxiliary

  interface
    type Int is modeled by Integer
      exemplar i
      initially "i = 0"
      lemma "min_int ≤ i ≤ max_int"
    end Int

    function Get_Min_Int returns min : Int
      ensures "min = min_int"
    end Get_Min_Int

    function Get_Max_Int returns max : Int
      ensures "max = max_int"
    end Get_Max_Int

    procedure Increment
      parameters
        alters i : Int
      end parameters
      requires "i < max_int"
      ensures "i = Math_Add(#i,1)"
    end Increment
```

Figure 21

Specification for a Module Providing Type Int

Figure 21 (continued)

```
    function Add returns Sum : Int
      parameters
        preserves i : Int
        preserves j : Int
      end  parameters
      requires "min_int ≤ Math_Add(i,j) ≤ max_int"
      ensures "Sum = Math_Add(i,j)"
    end Add

    function Subtract returns Diff : Int
      parameters
        preserves i : Int
        preserves j : Int
      end  parameters
      requires "min_int ≤ Math_Sub(i,j) ≤ min_int"
      ensures "Diff = Math_Sub(i,j)"
    end Subtract

    function Multiply returns Prod : Int
      parameters
        preserves i : Int
        preserves j : Int
      end  parameters
      requires "min_int ≤ Math_Mult(i,j) ≤ max_int"
      ensures "Prod = Math_Mult(i,j)"
    end Multiply

    function Divide returns Quo : Int
      parameters
        preserves i : Int
        preserves j : Int
      end  parameters
      requires "(j ≤ 0) ⇒
                (Math_Mult(j,Math_Add(max_int,1)) < i and
                i < Math_Mult(j,Math_Sub(min_int,1))"
      ensures "Math_Abs(Math_Mult(j,Quo)) ≤ Math_Abs(i) and
                Math_Abs(Math_Sub(i,Math_Mult(j,Quo))) ≤
                Math_Abs(j)"
    end Divide

    control Less_Than_Or_Equal
      parameters
        preserves i : Int
        preserves j : Int
      end  parameters
      ensures Less_Than_Or_Equal iff "i ≤ j"
    end Less_Than_Or_Equal
  end  interface

end Bounded_Integer_Template
```

Constants min_int and max_int, which represent the minimum and maximum representable value, respectively, are defined by a realization of Bounded_Integer_Template. These constants are part of the auxiliary section of the conceptualization, and are used only to define operations and types in the conceptualization. Specifically, they are not directly available to a client. However, Bounded_Integer_Template defines functions Get_Min_Int and Get_Max_Int that return the minimum and maximum integer values.

The auxiliary section of Bounded_Integer_Template contains a constraint clause, which is an invariant (i.e., an assertion that is true at all times). In this respect constraints are similar to lemmas, discussed in Section 3.5.3. Unlike lemmas, constraints cannot be proved using other assertions defined in the conceptualization. Rather, the realization guarantees that constraints are met at all times.

Finally, a brief explanation of the definition of procedure Divide is in order. First, it is useful to rewrite the requires and ensures clauses using standard infix notation:

```
requires "(j ≤ 0) ⇒ (j·(max_int + 1) < i < j·(min_int – 1))"
ensures "|j·Q| ≤ |i| and |i – j·Q| < |j|"
```

The requires clause places two restrictions on the parameters — the divisor cannot be zero, and the quotient must be between min_int and max_int, inclusive. The first restriction (i.e., division by zero) is expressed by the fact that when j is zero, no value of i satisfies the requires clause. The second restriction (i.e., representable quotient) is interesting only when j is negative, for when j is positive the quotient is representable since it is between the dividend and zero, inclusive. Thus, the requires clause restricts the parameters only when j is less than or equal to 0.

3.6.2 Conceptualization Single_Link_Ref_Template

Conceptualization Single_Link_Ref_Template, presented in Figure 22, provides type Reference that is useful for implementing traditional “singly-linked” data structures involving nodes consisting of information and a next field. The motivation for presenting this example is twofold. First, it declares conceptualization auxiliary variables that hold module state information, and second it demonstrates that it is possible to formally define a module providing the functionality traditionally associated with “pointers” even though RESOLVE does not define pointers as a built-in type.

```
conceptualization Single_Link_Ref_Template
  parameters
    type Item
  end parameters

  auxiliary
    math facilities
      Number_Theory is Number_Theory_Template
        renaming
          Number_Theory.Integer as Integer
        end renaming

      Function_Theory_1 is Function_Theory_Template
        (Integer,math[Item])
        renaming
          Function_Theory_1.Function as Integer_To_Item
          Function_Theory.Delta as Delta
        end renaming

      Function_Theory_2 is Function_Theory_Template
        (Integer,Integer)
        renaming
          Function_Theory_2.Function as Integer_To_Integer
        end renaming
    end math facilities

    math variables
      Unused : Integer
      Info : Integer_To_Item
      Next : Integer_To_Integer
    end math variables

    initially "Unused = 1 and
               ∀i:Integer, Item.init(Info(i)) and Next(i) = 0"

    lemma "Unused ≥ 0 and ∀i:Integer, (i ≤ 0 or i ≥ Unused) ⇒
           (Next(i) = 0 and Item.init(Info(i)))"
  end auxiliary
```

Figure 22

Specification for a Module Providing Generic Type Reference

Figure 22 (continued)

```
interface
  type Reference is modeled by Integer
    exemplar r
    initially "r = 0"
    initialization
      ensures "Unused = #Unused and
               Info = #Info and Next = #Next"
    finalization
      ensures "Unused = #Unused and
               Info = #Info and Next = #Next"
    lemma "r ≥ 0"
  end Reference

  procedure New_Ref
    parameters
      alters r : Reference
      consumes x : Item
    end  parameters
    ensures "r = #Unused and Unused = #Unused + 1 and
             Delta(Info,{r}) and Info(r) = #x and
             Next = #Next"
  end New_Ref

  procedure Swap_Info
    parameters
      preserves r : Reference
      alters x : Item
    end  parameters
    requires "r ≠ 0"
    ensures "Delta(Info,{r}) and Info(r) = #x
             and x = #Info(r) and Unused = #Unused
             and Next = #Next"
  end Swap_Info

  procedure Advance_Next
    parameters
      alters r : Reference
    end  parameters
    ensures "r = Next(#r) and Unused = #Unused and
             Info = #Info and Next = #Next"
  end Advance_Next

  procedure Change_Next
    parameters
      preserves r1 : Reference
      preserves r2 : Reference
    end  parameters
    requires "r1 ≠ 0"
    ensures "Delta(Next,{r1}) and Next(r1) = r2 and
             Info = #Info and Unused = #Unused"
  end Change_Next
```

Figure 22 (continued)

```
      procedure Copy
        parameters
          preserves r1 : Reference
          alters r2 : Reference
        end parameters
        ensures "r2 = r1 and Unused = #Unused and
                 Info = #Info and Next = #Next"
      end Copy

      control Equal
        parameters
          preserves r1 : Reference
          Preserves r2 : Reference
        end  parameters
        ensures Equal iff "r1 = r2"
      end Equal

      control Is_Null
        parameters
          preserves r : Reference
        end  parameters
        ensures Is_Null iff "r = 0"
      end Is_Null
    end  interface

    description
      . . .
    end  description

  end Single_Link_Ref_Template
```

A variable of type Reference is modeled as a mathematical Integer. A Reference variable initially contains zero (which corresponds to 'nil' in the traditional view of pointers). Information (of type Item) and a next reference (of type Reference) are associated with each Integer by mathematical functions Info and Next, respectively. These functions must be defined globally to the module instance rather than locally to each Reference variable since they define the mappings for *all* Integers (i.e., all variables of type Reference). In other words, these functions (along with other items discussed shortly) define the state of the module instance, which is altered by operations provided by the facility.

The conceptual state of a facility is defined by variables declared in the auxiliary section of the conceptualization. In this example the state is defined by one Integer (Unused) and two functions over Integers (Info and Next).

Unused is the smallest positive Integer that has never been the value of a Reference variable. At module initialization, Unused = 1, and its value is non-decreasing throughout execution of the client. The actual value of Unused is not really important. What matters is that, at any given time during execution, no Reference variable has a value as large as Unused. This implies that when a Reference is given a new value by New_Ref, it is certainly a value not equal to any other Reference value at that time. Keeping track of Unused is simply one way to guarantee this property.

Info is a mapping from Integers to Items that associates each Reference value with some Item value. (In the traditional view of pointers, Info(p) corresponds to the data field of the record pointed to by p.) Info is formally defined as a total function, although only a portion of its domain is actually accessible.

Next is a function from Integers to Integers that associates each Reference value with another Reference value. (In the traditional view of pointers, Next(p) corresponds to the next field of the record pointed to by p.) Next is defined as a total function, although only a portion of its domain is actually accessible.

The initially clause in the auxiliary section is an assertion involving auxiliary variables that is true before any variable of type Reference is initialized, and before any operation defined by the conceptualization is invoked. In this example, Unused = 1, Info maps all Integers to an initial value of type Item, and Next maps all Integers to zero. Initialization code in the realization is responsible for accomplishing module-level initialization.

The initially clause for Reference indicates that each Reference variable initially contains zero, which corresponds to ‘nil’ in the traditional view of pointers. The initialization ensures clause specifies that the state of the facility does not change when a variable is initialized.

The distinction between these two clauses is subtle yet important. The initially clause is an assertion about the contents of a Reference variable, and is referenced in other assertions as `Reference.init`. This assertion may contain references to state

variables, but cannot reference ‘old’ and ‘new’ variable values (i.e., it may not reference variables with a ‘#’). The actual initialization of a variable is accomplished by executing an initialization routine defined in the realization (which is invoked automatically). The post condition of this routine is the conjunction of the initially clause with the initialization ensures clause. The initialization ensures clause relates the state after initialization to the state before initialization, and thus may reference ‘old’ and ‘new’ state variable values.

Variable finalization occurs when a variable is no longer accessible, such as at the end of the block in which the variable is declared. Actual finalization is accomplished by a finalization routine defined in the realization (which is invoked automatically). The post condition of this routine is specified in the finalization ensures clause, which relates the state before a variable is finalized to the state after. In this example, variable finalization does not change the state.

Variable finalization has no conceptual effect on the variables being finalized (since they are no longer accessible). The reason for including finalization in RESOLVE is to give a realization the opportunity to reclaim resources (such as memory) allocated to variables no longer needed.

Seven operations are defined by Single_Link_Ref_Template. New_Ref returns an Integer that has never been assigned, and sets Info to map to the Item passed. Swap_Info exchanges an Item with the Info associated with a Reference. Advance_Next advances a Reference to its Next reference. Change_Next changes the Next mapping of a Reference to another Reference. Copy makes a copy of a Reference. Equal and Is_Null return indications of equal References and a Null Reference, respectively.

3.6.3 Conceptualization ADO_Stack_Template

Modules designed using conventional “object-oriented” design guidelines conceptually incorporate the data object within the conceptualization. Although it is not recommended to design modules this way, these modules can be specified in RESOLVE using conceptualization variables, as demonstrated by conceptualization ADO_Stack_Template, presented in Figure 23.

```
conceptualization ADO_Stack_Template
  parameters
    type Item
  end  parameters

  auxiliary
    math  facilities
      String_Theory is String_Theory_Template (math[Item])
        renaming
          String_Theory.String as String
          String_Theory.Lambda as Lambda
          String_Theory.Post as Post
        end  renaming
    end  math  facilities

    variables
      Stack : String
    end  variables

    initially "Stack = Lambda"
  end  auxiliary

  interface
    procedure Push
      parameters
        consumes x : Item
      end  parameters
      ensures "Stack = Post(#Stack, #x)"
    end Push

    procedure Pop
      parameters
        produces x : Item
      end  parameters
      requires "Stack ≠ Lambda"
      ensures "#Stack = Post(Stack, x)"
    end Pop

    control Is_Empty
      ensures Is_Empty iff "Stack = Lambda"
    end Is_Empty
  end  interface

end ADO_Stack_Template
```

Figure 23

Specification for a Module Providing an "Object-Oriented" Stack

The similarities between this and Stack_Template (presented in Figure 2) are obvious. The differences are also quite obvious — every instance of ADO_Stack_Template has its own stack, and invoking the operations from an instance effects only that stack. It is

also not possible to swap two ADO_Stacks or to pass an ADO_Stack as a parameter to an operation since there are no stack variables available to a client.

3.6.4 Summary

This section discussed conceptualization constants, variables, module initialization, and type finalization. Constants permit a realization to provide conceptual information to the client. Conceptual variables contain the state of a module instance, whose initial state is specified by an initially assertion. These variables can be used to implement "object-oriented" modules, though this design is not recommended. Finally, the effect that variable initialization and finalization has on the state of the facility is specified in the initialization ensures and finalization ensures clauses of the type definition.

These constructs were presented by discussing three examples — Bounded_Integer_Template that defines computational integers, Single_Link_Ref_ Template that defines a structure corresponding to traditional "singly-linked" data structures, and ADO_Stack_Template that defines an object-oriented stack. These examples also demonstrate how types traditionally built-in to languages can be formally specified using the same mechanism used to define "user-defined" types.

3.7 Realization Parameters, Constants, and Variables

There are situations where it is necessary for a client to pass information to the realization when an instance is declared, possibly in addition to what was passed to the conceptualization. This is accomplished by means of realization parameters in RESOLVE.

Also, it is often necessary for a facility to maintain realization state information during execution. In RESOLVE, this is accomplished by declaring variables and/or constants within the realization auxiliary section of a realization.

It is important to understand that realization parameters and realization state variables do not affect the conceptual behavior of the facility. For example, changing an actual realization parameter cannot alter the correctness of a client (assuming, of course, that the actual realization parameter is syntactically legal). These constructs are necessary

only so performance goals can be met, and for other reasons dealing with *how* a conceptualization is implemented, not *what* it does.

This section discusses these RESOLVE constructs by presenting realization examples. Realization parameters are demonstrated by realization Copy_Stack_Real_1 for conceptualization Copy_Stack_Template, and realization state variables are demonstrated by realization List_Real_1 for conceptualization One_Way_List_Template.

3.7.1 Realization Copy_Stack_Real_1 of Copy_Stack_Template

Conceptualization Copy_Stack_Template, presented in Figure 18, defines procedure Copy_Stack that copies one stack into another, even if the implementations of the two stacks are different. Realization Copy_Stack_Real_1 for this conceptualization is presented in Figure 24.

```
realization of Copy_Stack_Template by Copy_Stack_Real_1

  conceptualization  parameters
    renaming
      Stack_Facility1.String_Theory.Reverse as Reverse
      Stack_Facility1.String_Theory.Cat as Cat
      Stack_Facility1.Item as Item
    end  renaming
  end  conceptualization  parameters

  realization  parameters
    procedure Copy_Item
      parameters
        preserves x : Item
        produces y : Item
      end  parameters
      ensures "y = x"
    end Copy_Item
  end  realization  parameters
```

Figure 24

Realization Copy_Stack_Real_1 of Copy_Stack_Template

Figure 24 (continued)

```
    interface
      procedure Copy_Stack
        parameters
          preserves s1 : Stack_Facility1.Stack
          produces s2 : Stack_Facility2.Stack
        end parameters

        begin
          local variables
            garbage : Stack_Facility2.Stack
            catalyst : Stack_Facility1.Stack
            temp : Item
            temp_copy : Item
          end local variables

          garbage :=: s2

          ensuring "catalyst = Cat(#catalyst,Reverse(#s1)) and
                    s1 = Lambda"
          while not Stack_Facility1.Is_Empty(s1) do
            Stack_Facility1.Pop (s1,temp)
            Stack_Facility1.Push (catalyst,temp)
          end while

          ensuring "s1 = Cat(#s1,Reverse(#catalyst)) and
                    s2 = Cat(#s2,Reverse(#catalyst))"
          while not Stack_Facility1.Is_Empty(catalyst) do
            Stack_Facility1.Pop (catalyst,temp)
            Copy_Item (temp,temp_copy)
            Stack_Facility1.Push (s1,temp)
            Stack_Facility2.Push (s2,temp_copy)
          end while
      end Copy_Stack
    end interface

    description
      . . .
    end description

  end Copy_Stack_Real_1
```

Procedure Copy_Stack must be able to make a copy of a value of type Item. Since nothing is known about this type, the client provides a copy procedure to the realization via realization parameter Copy_Item. The formal parameter declaration consists of the procedure signature (i.e., types and modes of all parameters) as well as a specification.

The compiler checks the actual procedure passed against the formal signature and specification.

The conceptual parameters section in a realization allows conceptual parameter items to be renamed for convenience. Additional conceptual parameters cannot be specified in this section!

It is also important to note that all items declared in the conceptualization can be referenced in the realization. For example, Stack_Facility1 is an instance of Stack_Template passed as a conceptual facility parameter, therefore procedure Stack_Facility1.Push is accessible within Copy_Stack_Real_1.

A client of Copy_Stack_Real_1 was presented earlier in Figure 19.

3.7.2 Realization List_Real_1 of One_Way_List_Template

Conceptualization One_Way_List_Template, presented earlier in Section 3.1.8 and Figure 4, defines a structure useful for storing information that is accessed sequentially in one direction only. Realization List_Real_1, presented in Figure 25, implements this conceptualization using Single_Link_Ref_Template, presented in Figure 22.

```
realization of One_Way_List_Template by List_Real_1

  realization  auxiliary
    facilities
      Item_Ref_Facility is Single_Link_Ref_Template (Item)
        realized by Single_Link_Real_1
        renaming
          Item_Ref_Facility.Reference as Item_Ref
          Item_Ref_Facility.New_Ref as New_Ref
          Item_Ref_Facility.Swap_Info as Swap_Info
          Item_Ref_Facility.Advance_Next as Advance_Next
          Item_Ref_Facility.Change_Next as Change_Next
          Item_Ref_Facility.Copy as Copy
          Item_Ref_Facility.Equal as Equal
          Item_Ref_Facility.Is_Null as Is_Null
        end  renaming
```

Figure 25

Realization List_Real_1 of One_Way_List_Template

Figure 25 (continued)

```
    Record_Facility is Record_2_Template (Item_Ref,Item_Ref)
      realized by Record_2_Real_1
      renaming
        Record_Facility.Record as List_Rep
        Record_Facility.Access_1 as Access_PreFirst
        Record_Facility.Access_2 as Access_Prev
      end  renaming

    StringerRef is Single_Link_Ref_Template (Item_Ref)
      realized by Single_Link_Real_1
  end  facilities

  variables
    Avail : StringerRef.Reference
  end  variables

  operations
    procedure Get_Ref
      parameters
        produces r : Item_Ref
        consumes x : Item
      end  parameters
      ensures "r ≠ 0"
      begin
        local  variables
          FirstList : Item_Ref
        end  local  variables

        if StringerRef.Is_Null(Avail) then
          New_Ref (r)
        else
          StringerRef.Swap_Info (Avail,FirstList)
          Copy (FirstList,r)
          Swap_Info (r,x)
          Advance_Next (FirstList)
          if not Is_Null(FirstList) then
            StringerRef.Swap_Info (Avail,FirstList)
          else
            StringerRef.Advance_Next (Avail)
          end  if
        end  if
      end Get_Ref
    end  operations
end  realization  auxiliary
```

Figure 25 (continued)

```
interface
  type List is represented by List_Rep
    exemplar L_rep
    correspondence "---"

    initialization
      performance "O(1)"
      begin
        local variables
          PreFirst : Item_Ref
          Prev : Item_Ref
          Init_Item : Item
        end local variables

        Get_Ref (PreFirst,Init_Item)
        Copy (PreFirst,Prev)
        Access_PreFirst (L_rep,PreFirst)
        Access_Prev (L_rep,Prev)
    end initialization

    finalization
      performance "O(1)"
      begin
        local variables
          FreeList : StringerRef.Reference
          PreFirst : Item_Ref
        end local variables

        Access_PreFirst (L_rep,PreFirst)
        StringerRef.New_Ref (FreeList,PreFirst)
        StringerRef.Change_Next (FreeList,Avail)
        FreeList :=: Avail
    end finalization
  end List

  procedure Reset
    parameters
      alters L : List
    end parameters
    performance "O(1)"

    begin
      local variables
        PreFirst : Item_Ref
        Prev : Item_Ref
      end local variables

      Access_PreFirst (L,PreFirst)
      Copy (PreFirst,Prev)
      Access_PreFirst (L,PreFirst)
      Access_Prev (L,Prev)
  end Reset
```

Figure 25 (continued)

```
procedure Advance
  parameters
    alters L : List
  end parameters
  performance "O(1)"

  begin
    local variables
      Prev : Item_Ref
    end local variables

    Access_Prev (L,Prev)
    Advance_Next (Prev)
    Access_Prev (L,Prev)
end Advance

procedure Add_Right
  parameters
    alters L : List
    consumes x : Item
  end parameters
  performance "O(1)"

  begin
    local variables
      newItem : Item_Ref
      Prev : Item_Ref
      Curr : Item_Ref
    end local variables

    Access_Prev (L,Prev)
    Copy (Prev,Curr)
    Advance_Next (Curr)

    Get_Ref (newItem,x)
    Change_Next (newItem,Curr)
    Change_Next (Prev,newItem)

    Access_Prev (L,Prev)
end Add_Right
```

Figure 25 (continued)

```
procedure Remove_Right
  parameters
    alters L : List
    produces x : Item
  end parameters
  performance "O(1)"

  begin
    local variables
      Prev : Item_Ref
      Curr : Item_Ref
    end local variables

    Access_Prev (L,Prev)
    Copy (Prev,Curr)
    Advance_Next (Curr)

    Swap_Info (Curr,x)
    Advance_Next (Curr)
    Change_Next (Prev,Curr)

    Access_Prev (L,Prev)
end Remove_Right

procedure Swap_Rights
  parameters
    alters L1 : List
    alters L2 : List
  end parameters
  performance "O(1)"

  begin
    local variables
      Prev1 : Item_Ref
      Curr1 : Item_Ref
      Prev2 : Item_Ref
      Curr2 : Item_Ref
    end local variables

    Access_Prev (L1,Prev1)
    Copy (Prev1,Curr1)
    Advance_Next (Curr1)
    Access_Prev (L2,Prev2)
    Copy (Prev2,Curr2)
    Advance_Next (Curr2)

    Change_Next (Prev1,Curr2)
    Change_Next (Prev2,Curr1)

    Access_Prev (L1,Prev1)
    Access_Prev (L2,Prev2)
end Swap_Rights
```

Figure 25 (continued)

```
      control At_Left_End
        parameters
          preserves L : List
        end parameters
        performance "O(1)"

        begin
          local variables
            PreFirst : Item_Ref
            Prev : Item_Ref
            temp1 : Item_Ref
            temp2 : Item_Ref
          end local variables

          Access_PreFirst (L,PreFirst)
          Access_Prev (L,Prev)
          Copy (PreFirst,temp1)
          Copy (Prev,temp2)
          Access_PreFirst (L,PreFirst)
          Access_Prev (L,Prev)
          if Equal(temp1,temp2) then
            return yes
          else
            return no
          end if
      end At_Left_End

      control At_Right_End
        parameters
          preserves L : List
        end parameters
        performance "O(1)"

        begin
          local variables
            Prev : Item_Ref
            Curr : Item_Ref
          end local variables

          Access_Prev (L,Prev)
          Copy (Prev,Curr)
          Advance_Next (Curr)
          Access_Prev (L,Prev)
          if Is_Null(Curr) then
            return yes
          else
            return no
          end if
      end Is_Empty
    end interface

  end List_Real_1
```

List_Real_1 implements one-way lists using a standard "linked-list" representation discussed in most data structures texts. The items contained in a List are stored in a linked structure implemented using conceptualization Single_Link_Ref_Template. A List is represented by a record containing two references — PreFirst and Prev. PreFirst is a reference to the "dummy" node that is at the head of every linked list, and Prev is a reference to the item to the left of the fence.

Part of the complexity of the above realization is a result of implementing all operations to execute in constant time, including List initialization and finalization. This is a noble and worthwhile goal for any realization, and many issues are raised in reaching it. For instance, because every variable is initialized and finalized, it is especially important that these operations have efficient implementations. Initialization is normally not difficult to implement in constant time because the initial value for a type is usually defined to be one that is easy to construct.

Finalization is another matter because the finalization routine has no control over the values it must finalize. For example, one-way list finalization must be able to finalize empty lists as well as lists containing any number of items. Also, if the value is composite (e.g., a one-way list of Items) all components should also be finalized when the structure is finalized. Algorithms for constant time finalization are seldom obvious.[35]

Realization List_Real_1 accomplishes constant time finalization by using a trick that amortizes the cost of finalizing all items on a List over subsequent List insert operations. When a List is finalized it is placed on an internal list that contains finalized Lists, which requires a constant amount of time. This internal list is implemented by variable Avail. When an Item is inserted onto a List (by the client invoking Add_Right), an item from a previously finalized list is finalized if there is one, and its reference is reused for the new Item. If there are no finalized lists, a new reference is obtained and used for the inserted Item. A complete discussion of this approach to constant time initialization and finalization can be found in [Harms 89a].

---

35 Of course, since finalization usually has no conceptual effect, it need not do anything, which takes constant time! This approach makes no attempt to recover resources (such as memory) used to represent inaccessible variables, and is therefore neither advisable nor realistic.

Several interesting features of RESOLVE realizations are demonstrated by List_Real_1. Avail is declared as a realization auxiliary variable, and contains realization state information for the facility. It is a static variable in the C sense of the word, meaning its lifetime is for the entire program. Procedure Get_Ref is a local procedure to the realization. As is true of all items declared in the realization auxiliary section, Avail and Get_Ref are accessible only within the realization.

The initialization and finalization routines for Lists are defined within the type declaration. These routines are invoked automatically, and their purpose is to create an initial value and reclaim memory occupied by an inaccessible value, respectively.

3.7.3 Other Realization Sections

Although most of the sections of RESOLVE realizations are demonstrated by the examples in this chapter, several are not. For instance, a realization may need code to initialize the realization state of the facility. In realization List_Real_1 (presented in Figure 25) Avail is the only state variable, and it just happens that an initial value for its type is exactly what is needed as the initial state, so facility initialization code was not needed. However, this is not the case in general, so it is possible to define a facility initialization routine within the realization auxiliary section, which is automatically executed when the program begins execution.

It is also possible to declare constants within the realization auxiliary section. Constants are similar to variables, except that their value can be changed only during facility initialization (i.e., only by the facility initialization routine); to all other routines constants are "read-only" variables, and can only be used as actual preserves parameters.

In addition to operations, facilities and types can be passed as realization parameters. These kinds of parameters are useful for parameterizing performance characteristics of realizations, and are discussed more fully in [Muralidharan 90].

3.7.4 Summary

In this section some very interesting and powerful features of RESOLVE realizations were discussed — realization parameters, realization state variables, and initialization

and finalization of types. This discussion centered around two non-trivial realizations for conceptualizations presented earlier — realization Copy_Stack_Real_1 for Copy_Stack_Template, and realization List_Real_1 for One_Way_List_Template.

3.8 Implementation Issues

One of the primary goals of RESOLVE is to provide the programmer with mechanisms enabling him or her to create efficient implementations of formally specified (and verifiable) software components. We have already discussed some implementation issues, such as implementing swap as a constant time operation in Section 3.3.1.2, and constant time initialization and finalization in Section 3.6.2.

In this section we'll examine three additional implementation issues — implementing primitive conceptualizations, lazy initialization, and efficient implementations of generic conceptualizations.

3.8.1 Primitive Realizations for Conceptualizations

Recall that all types referenced in a RESOLVE program must be formally defined in a conceptualization module. This holds even for types that are built-in to most languages, such as integers, characters, and booleans. In other words, there are absolutely no built-in types in RESOLVE. This approach to built-in types simplifies the definition of the language by making it very regular, and allows the programmer to select realizations that are appropriate for the constraints and performance goals of the particular client.

Realizations are a somewhat different matter. There must be a set of "primitive" realizations upon which other realizations can be built. These primitive realizations are defined in compilation units such as machine language or a "system" dialect of RESOLVE. Using one of these realizations is no different than using a regular realization, since the actual realization code is completely hidden from the client.

The programmer will have a library of "standard" realizations for commonly used conceptualizations such as integers, characters, booleans, and arrays. These will most likely be realizations built directly on the hardware, though this fact is transparent to the programmer.

3.8.2 Lazy Initialization of Variables

As mentioned in Section 3.6.2, it is desirable to implement type initialization and finalization efficiently (preferably as a constant time operations) because every variable is automatically initialized at the beginning of the block in which it is declared and finalized at the end of the block. However, the initial value in many variables is never actually referenced before the variable is finalized. For example, the initial values in variables temp and temp_copy in Figure 24 are never referenced. It might be a desirable characteristic of an implementation of RESOLVE to not waste time and memory initializing variables whose initial value is never referenced.

An implementation of RESOLVE might be able to take advantage of the fact that word addresses are even values in most modern machine architectures. If this is the case, the implementation could automatically place an odd value in each variable when it allocates memory for it. Let's assume that all variables are implemented as pointers to the actual representation of the value, and that representations always start on word boundaries. In this situation, all variables initially contain an illegal address that will cause a run-time trap when accessed. When a trap occurs, the trap routine invokes the appropriate initialization routine for the variable, and stores the address of the actual representation in the variable. All subsequent access to the variable will be legitimate.

When a variable is to be finalized, the implementation checks it for an odd value. If it is odd, the variable was never actually initialized, and thus does not need to be finalized. On the other hand, if the variable contains an even value, the appropriate finalization routine is invoked.

This implementation trick does not violate any conceptualization assertions that state all variables have an initial value. Indeed, the first time a variable's value is accessed, it is an initial value, which is exactly what the initially clause is for.

3.8.3 Efficient Implementation of Generic Modules

Implementing all RESOLVE variables as pointers has several advantages when coupled with the fact that the only data movement primitive is swapping the value of two variables. One advantage, discussed in Section 3.3.1.2, is that swap can be

implemented as a constant time operation. Another advantage, discussed here, is that the object code for a realization can be shared by all instances of it.

Consider the object code produced by the swap statement '`x :=: y`' where variables x and y are implemented as pointers. In most machine architectures this is effected by three MOVE instructions, where each MOVE moves a fixed-size bit sequence (e.g., a 32-bit address). Note that these statements do not depend on the type of x and y. So even if the type of x and y are unknown to the compiler (which is the case with type parameters in generic modules) it can still produce object code. Extending this one step further, there is no reason that each instance utilizing a particular realization must have its own copy of the object code, since the object code is identical.

The only type-specific operations that can be invoked within the generic facility are initialization and finalization of the type parameter. This can be easily implemented by creating a run-time structure for each facility that contains, among other items, the addresses of initialization and finalization routines for all types referenced in the module. When the initialization or finalization routine for a type parameter needs to be invoked, the instance-specific structure is used at execution time to invoke the proper routine.

This and other RESOLVE run-time structures are discussed more fully in [Hegazy 89].

3.9 Summary

RESOLVE is a programming language designed specifically to encourage the design and implementation of reusable software components, whose characteristics are described in Section 2.3. Specifically, each component's behavior is formally described using assertions in mathematical theories (that are themselves formally defined in theory modules). Every program type is modeled as a mathematical type, and every program variable is considered to be a value from the type's mathematical type for purposes of reasoning about behavior of the variable. Operations are formally specified with requires and ensures clauses that specify the pre- and post-conditions, respectively. Although it is possible (and strongly encouraged) to also describe each component informally, the formal description is always used as the "final word" concerning the behavior.

The behavior of a component is specified in a conceptualization while the structures and code that implement a conceptualization are contained in a realization, thus separating component specification from implementation. In addition, it is possible to have multiple realizations for any conceptualization. A client gains access to the types and operations defined in a conceptualization by instantiating it (also called declaring a facility). A facility declaration binds a conceptualization with a realization, and binds actual with formal parameters. A different realization may be bound to each instance of a particular conceptualization .

A conceptualization is generic if it has one or more type parameters. It is thus possible to define structures where the type of certain components is passed as a parameter.

RESOLVE modules can be formally verified using techniques described in [Krone 88]. Verification is feasible here because of several characteristics of RESOLVE modules. First, all types and operations have formal mathematical specifications. Also, every type has a defined initial value, so the verification system need not incorporate the notion of an "undefined" value, thus simplifying verification. Second, all loop constructs have an associated assertion that formally defines the specification of the loop, which is similar to operation specification. Third, pointers are not a built-in type in RESOLVE (although they can be defined via a conceptualization), eliminating the possibility of aliasing and other problems relating to pointers, such as "dangling references," that typically complicate verification systems.

Finally, it is possible to efficiently implement RESOLVE modules, even generic ones. This is due largely to the fact that the only data movement primitive in RESOLVE is swapping the values of two variables, which takes constant time. All operation parameters are defined in terms of swapping actual with formal parameters, so parameter passing also takes constant time. Copying a value is accomplished by an operation defined by a conceptualization just like all other operations. It may not be possible to copy every type of value since RESOLVE does not require a copy operation be defined for every type. Thus, generic modules are usually designed in such a manner that values are swapped into the structure, rather than copied, because the generic type may not have a copy operation.

RESOLVE supports the second point of the thesis, namely that it is possible to have a usable language incorporating constructs that encourage the design of reusable software parts.