

Part II: Specifying Components in RESOLVE

Stephen H. Edwards
Wayne D. Heym
Timothy J. Long
Murali Sitaraman
Bruce W. Weide

A *conceptual module* is a RESOLVE unit that formally specifies an abstract component by defining both its context and its interface structure and behavior. The purpose of this paper is to explain conceptual modules and how to organize specifications of reusable components.

Conceptual modules may export two kinds of things for use in client programs: *type families* and *operation families*. We say “families” here because every RESOLVE module is *generic*, so a client must *instantiate* a module before using it. Instantiation has two parts: First you bind all of a conceptual module’s formal parameters to actuals which match the formals both in structure and in other specified properties; then you select an implementation for the concept and fix the realization’s (additional) parameters [Part III]. An instance created this way is called a *facility*. For a typical conceptual module that defines one type family and associated operation families, every instance defines a particular *type* and some particular *operations* whose specifications result, in effect, from replacing the formal parameters of the generic specification with the actuals for that instance.

Usually, the difference between a particular type/operation and the corresponding family of which it is a member is clear, and we omit the word “family” for brevity. But the difference is important. More information about the treatment of types in RESOLVE appears elsewhere [Harms 90, Harms 91].

The organization of this paper is as follows. First we briefly discuss the kind of thinking involved in writing and reading a RESOLVE conceptual module. Then we present several examples and classify them along the lines recommended by the RESOLVE discipline. Finally, we explain the most important notational details of conceptual modules. The recommended approach to this last section (and to the corresponding sections in the companion paper for realizations [Part III]) is to skip the boxed syntax descriptions on first reading, and to use them later to answer questions about details.

1. Mathematical Modeling

A software engineer designing and specifying program behavior in RESOLVE starts by *mathematically modeling* the intended behavior. For example, suppose you “know” how

objects of program type Stack should behave. Perhaps this behavior mimics some real-world phenomena, say, the behavior of cafeteria trays. Saying “Stacks behave like cafeteria trays” is not precise enough either to convey this intent unambiguously to a reader, nor to serve as the basis for careful checking of program properties. Intuition about cafeteria trays and intentions for Stack behavior must be captured using a formal mathematical description. This involves creating and writing down a mathematical model of just the aspects of cafeteria tray behavior that Stacks should have.

This software engineering activity reflects the usual meaning of the term “mathematical modeling”. But the main purpose of mathematical modeling here is not to permit analysis — which is what, say, a civil engineer generally wants to do when claiming that a truss is modeled by a set of equations relating forces in its members. At this point the mathematical model defines the synthesized behavior of Stacks, i.e., a design.

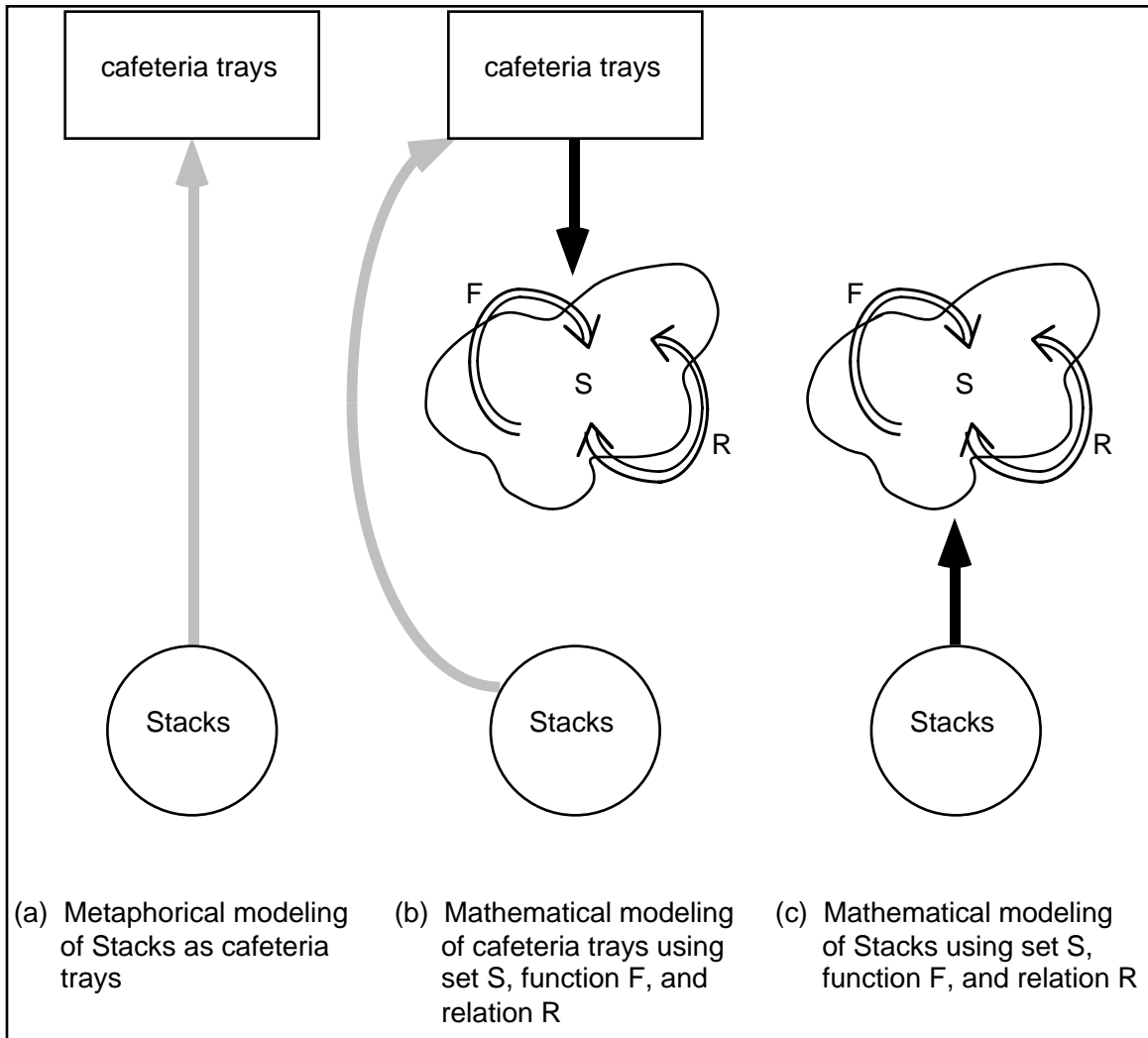
Of course the model also supports certain analyses of Stacks. The primary (but certainly not only) activity of this kind is program *verification*, or proof-of-correctness of implementation. Stacks themselves are mathematical objects, notwithstanding the recording of their descriptions on paper or disk or their ephemeral existence in a computer’s physical memory. This means that you can use formal methods to compare the abstract specification of the way Stacks are supposed to behave (as defined in terms of the mathematical model discussed here) with the actual behavior of Stacks in a program, as defined in terms of a different mathematical model of that program’s execution (as defined by the programming language semantics).

In traditional engineering, there is ordinarily no way to compare mathematically modeled behavior and actual behavior except by observation. This problem — *validating* the mathematical model (“are we building the right thing?”) — arises both in traditional engineering and in programming. But there is no directly parallel notion of formal verification (“are we building the thing right?”) for physical artifacts of engineering.

Figure 1 depicts the modeling aspect of the specification activity for the case discussed above. In step (a) you decide that you want Stacks to behave “like” cafeteria trays; this might be termed “metaphorical modeling”. Next, in (b) you develop a traditional mathematical model of the relevant aspects of cafeteria tray behavior, in terms of (say) a set of values S called the *universe of discourse* and functions (such as F) and relations (such as R) on S . Finally, in (c) you argue that this model is also a mathematical model of the intended Stack behavior, and you can forget about cafeteria trays.

As we subsequently operate entirely within the mathematical realm that includes such things as program Stacks and their formal specification, but not the likes of cafeteria trays, we restrict the term “mathematical model” to the situation in Figure 1(c). That is, mathematical models relate sets of values and functions and relations on those sets, to some program behavior that you are trying to specify. Cafeteria trays still might be useful in an informal description of Stack behavior. But they play no part in the formal mathematical “legal” contract between the specifier of Stacks and clients or implementers of Stacks.

Figure 1: Three Cases of “Modeling”



2. A Taxonomy of Concepts (With Examples)

To show specifically how mathematical models are described and used to specify programming concepts in RESOLVE, we provide a few simple but illustrative examples. Figures 2–5 depict specifications in the three main categories that we recommend as the basis for organizing abstract components in RESOLVE:

- *A kernel component* — Figure 2 defines a new type (actually, a type family) called Stack and some operations¹ for it — Push, Pop, and Length — following the scenario outlined above. Figure 3 shows a more complex example, defining a type

¹ A concept also might export “constants” (although Stack_Template does not) as operations having no parameters.

called `Partial_Map` that resembles what other authors often call a “table” or “dictionary”. Its operations are `Define`, `Undefine`, `Undefine_Any_One`, `Is_Defined`, and `Size`.

A kernel concept may export more than one type. But in practice most concepts export just one because usually — although not always — it is possible and advantageous to de-couple even closely related types into separate components.

An operation specified along with a type in a kernel component is called a *primary* operation for that type, as its implementation has complete access to the representation of the exported type in any realization of the concept. But inevitably there are additional operations associated with a type, and these are specified in other conceptual modules. So a kernel component is the centerpiece of a group of concepts that are closely related to it.

- An *addition* — Figure 4 shows how to extend the basic functionality of `Stack_Template` by defining individual additional operations — `Top` and `Replica`.

An addition usually is parameterized by instances of other concepts to which it is closely related (e.g., `Stack_Template` in Figure 4). Note that an addition exports only the additional operation(s), not the operations for the related kernel concepts. Each of an addition’s realizations necessarily is *layered* on top of these underlying abstractions. An operation specified in an addition, therefore, is called a *secondary* operation for the types exported by the facilities by which the addition is parameterized.

- An *enhancement* — Figure 5 shows how to couple a kernel component with one of its additions, defining their combination `Replicable_Stack_Template` as a new component that extends each of the others. From the standpoint of its interface, this enhancement has the character of a kernel component, exporting a type and some operations. But the expression of this interface is relatively concise because it reuses the specifications of previously-defined components.

Realization of an enhancement might be accomplished in any of several ways. For example, you might layer `Replicable_Stack_Template` on top of instances of `Stack_Template` and `Stack_Replica_Capability` by using “call-through” or “delegation”. Or you might implement the entire interface from scratch. In `Replicable_Stack_Template`, the code for the `Replica` operation can be *directly implemented* with knowledge of the underlying `Stack` representation shared by `Push`, `Pop`, and `Length`, i.e., as though `Replica` were a primary operation for `Stacks`. Cleverly done, this approach can yield order-of-magnitude performance benefits: All the operations can be made to run in time that is independent of the length of the `Stacks` involved. A layered implementation leaves `Replica` with linear-time performance, at best.

There are other sorts of concepts, too, that we don't illustrate here. One important kind is a *specialization*. The main reason for specializing a concept is to simplify it in some way: understanding of behavior, number of parameters to be fixed for instantiation, etc. It also is possible that a specialized concept might admit more efficient realizations than a fully general one because the realization can take advantage of assumptions inherent in the specialized concept's more restrictive context.

In aggregate, these examples illustrate most of the features of the RESOLVE notation for conceptual modules, which we explain in more detail in the following sections.

Figure 2: A Kernel Component: Stack_Template

```

concept Stack_Template

  context

    global context

      facility Standard_Integer_Facility

    parametric context

      type Item

  interface

    type Stack is modeled by string of math[Item]
    exemplar      s
    initialization
      ensures      s = empty_string

    operation Push (
      alters      s: Stack
      consumes    x: Item
    )
    ensures      s = <#x> * #s

    operation Pop (
      alters      s: Stack
      produces    x: Item
    )
    requires      s /= empty_string
    ensures      #s = <x> * s

    operation Length (
      preserves s: Stack
    ): Integer
    ensures      Length = |s|

end Stack_Template

```

Figure 3: Another Kernel Component: Partial_Map_Template

```
concept Partial_Map_Template

context

  global context

    facility Standard_Boolean_Facility
    facility Standard_Integer_Facility

  parametric context

    type D_Item

    type R_Item

  local context

    math subtype PARTIAL_FUNCTION is set of (
      d: math[D_Item]
      r: math[R_Item]
    )
    exemplar      m
    constraint    for all d: math[D_Item],
                  r1, r2: math[R_Item]
                  where (d, r1) is in m and
                       (d, r2) is in m
                  (r1 = r2)

    math operation DEFINED_IN (
      m: PARTIAL_FUNCTION
      d: math[D_Item]
    ): boolean
    definition     DEFINED_IN (m, d) iff
                  there exists r: math[R_Item]
                  ((d, r) is in m)

interface

  type Partial_Map is modeled by PARTIAL_FUNCTION
  exemplar      m
  initialization
  ensures      m = empty_set

  operation Define (
    alters      m: Partial_Map
    consumes    d: D_Item
    consumes    r: R_Item
  )
  requires     not DEFINED_IN (m, d)
  ensures     m = #m union {(#d, #r)}
```

```

operation Undefine (
    alters      m: Partial_Map
    preserves  d: D_Item
    produces   d_copy: D_Item
    produces   r: R_Item
)
requires     DEFINED_IN (m, d)
ensures     (d, r) is in #m and
              m = #m - {(d, r)} and
              d_copy = d

operation Undefine_Any_One (
    alters      m: Partial_Map
    produces   d: D_Item
    produces   r: R_Item
)
requires     m /= empty_set
ensures     (d, r) is in #m and
              m = #m - {(d, r)}

operation Is_Defined (
    preserves  m: Partial_Map
    preserves  d: D_Item
): Boolean
ensures     Is_Defined iff DEFINED_IN (m, d)

operation Size (
    preserves  m: Partial_Map
): Integer
ensures     Size = |m|

end Partial_Map_Template

```

Figure 4: Two Additions to Stack_Template

```
concept Stack_Top_Capability
  context
    global context
      concept Stack_Template
    parametric context
      type Item
      facility Stack_Facility is Stack_Template (Item)
  interface
    operation Top (
      preserves s: Stack
    ): Item
    ensures there exists r: math[Stack]
      (s = <Top> * r)
end Stack_Top_Capability

concept Stack_Replica_Capability
  context
    global context
      concept Stack_Template
    parametric context
      type Item
      facility Stack_Facility_1 is Stack_Template (Item)
      facility Stack_Facility_2 is Stack_Template (Item)
  interface
    operation Replica (
      preserves s: Stack_Facility_1.Stack
    ): Stack_Facility_2.Stack
    ensures Replica = s
end Stack_Replica_Capability
```

Figure 5: An Enhancement of Stack_Template

```
concept Replicable_Stack_Template

  context

    global context

      concept Stack_Template
      concept Stack_Replica_Capability

    parametric context

      type Item

    local context

      facility Stack_Facility is Stack_Template (Item)

      facility Stack_Replica_Facility is Stack_Replica_Capability
        (Item, Stack_Facility, Stack_Facility)

  interface

    re-exports

      Stack_Facility
      Stack_Replica_Facility

end Replicable_Stack_Template
```

3. Notation for Conceptual Modules

We do not attempt to include precise syntax for, or discussion of, all of the RESOLVE specification language. We skip conventional aspects (e.g., identifier syntax) and those parts explained elsewhere (e.g., the assertion language [Heym 94a]), and concentrate on elements that have more interesting ramifications. Practically speaking, this means that many non-terminal symbols remain unelaborated in the syntax summaries and unexplained in the accompanying text. We trust this will cause no serious difficulties in understanding the major ideas.

Two conventions help to simplify and shorten the presentation:

- A non-terminal `<NT_list>` denotes one or more occurrences of non-terminal `<NT>`, where these occurrences are separated by commas.
- A non-terminal `<NT_sequence>` denotes one or more occurrences of `<NT>`, where these occurrences are separated *either* by white-space or by commas, depending on

personal preference. This case turns up when no ambiguity can arise from using just white-space, because of the syntax of the `<NT>` involved.

Now we can proceed with a discussion of conceptual modules, each of which defines an abstract component's context and interface:

```
<conceptual_module> ::=
  concept <conceptual_module_id>
    [<concept_context_section>]
    <concept_interface_section>
  end <conceptual_module_id>
```

RESOLVE conceptual modules deal with both *programming identifiers* and *mathematical identifiers*, the former denoting the programming things being specified — types and operations — and the latter denoting the mathematical things being used to model them [Heym 94a].² All identifiers of either kind that are used in a concept must be imported or declared in its context section, or must be declared in its interface section.

The interface section of a conceptual or realization module *explicitly exports* only programming identifiers, for use in writing executable statements. If a (concrete) client needs to refer to programming identifiers in code, it must instantiate a concept in order to get its explicit exports. But because these types and operations are formally specified using other programming identifiers which denote types, and mathematical identifiers, a conceptual module also *implicitly exports* all of the identifiers that are visible within it — but only for use in writing mathematical parts of (abstract or concrete) clients, not code.

3.1. Context Section

A conceptual module can have three different kinds of context: global, parametric, and local:

```
<concept_context_section> ::=
  context
    [<concept_global_context_section>]
    [<concept_parametric_context_section>]
    [<concept_local_context_section>]
```

One of the most important ways to bring identifiers into the context of a conceptual module C is to bring in a facility, i.e., an instance of some conceptual module C' .

² By convention, we use Mixed_Case for programming identifiers and UPPER_CASE for mathematical identifiers, with lower_case reserved for variables of both kinds.

Because C is an abstract client of C' and therefore contains no executable statements, it follows from the discussion above that when an instance of C' comes into the context of C it brings with it all identifiers that are visible within C' . Because of the way conceptual modules are organized, in C there is no need to use programming identifiers from C' that denote operations. But you can (and might need to) use both programming identifiers that denote types, and of course mathematical identifiers, from C' . Most of the examples in the figures illustrate this feature.

The scope of every identifier that is visible within a conceptual module is the entire module. This fact, combined with the transitive visibility of identifiers for writing mathematics, means that a concept often has visibility over two or more indistinguishable identifiers; and no interpretation takes precedence over others by virtue of ordinary block structure, for example. We handle such overloading by requiring qualified names if and only if an ambiguity arises that cannot be resolved by the following rules.

The first rule is that locally-declared names never are qualified. Consider Figure 4, where the parametric context of `Stack_Replica_Capability` includes two instances of `Stack_Template`. Visible within `Stack_Replica_Capability` are a local identifier “Item”, and the identifiers “Item” and “Stack” from both `Stack_Facility_1` and `Stack_Facility_2`. “Item” is not ambiguous, as it refers to the local identifier which is a formal type parameter. To talk about any other “Item”, e.g., `Stack_Facility_1.Item`, requires the qualified name. (However, there really is no need for this in the example because `Item` and `Stack_Facility_1.Item` and `Stack_Facility_2.Item` all refer to the same type, by the declarations of `Stack_Facility_1` and `Stack_Facility_2`.)

On the other hand, “Stack” is ambiguous in this conceptual module, and must be qualified wherever it appears. Indeed the purpose of `Stack_Replica_Capability` is to support copying of Stack values between representations. A client can create two instances of `Stack_Template` with the same `Item` type but with possibly different realizations, and then use these instances as the actuals corresponding to `Stack_Facility_1` and `Stack_Facility_2` when instantiating `Stack_Replica_Capability`.

The other rule for disambiguation involves using type information in the usual way: If two operations with the same name have the same type signature, then qualification is required; otherwise it is not.

3.1.1. Global Context

Global context introduces fixed coupling to other modules or facilities in a *RESOLVE library*:

```

<concept_global_context_section> ::=
  global context
    <concept_global_context_item_sequence>

<concept_global_context_item> ::=
  concept <conceptual_module_id> |
  facility <facility_id> |
  mathematics <math_module_id> |
  math facility <math_facility_id>

```

The standard RESOLVE library, and project- and user-specific extensions that might be linked to it, contain various modules and instances of them. A conceptual module may include, in its global context, conceptual modules and/or math modules [Heym 94a] and/or instances of either kind. We do not further discuss how a library is created or how it is maintained, except to say that there are no programming language constructs involved; standard library-maintenance procedures and utilities are involved.

The RESOLVE discipline suggests, as a rule for minimizing coupling and maximizing reusability, that a concept should be “fully parameterized”: Put everything you can in the parametric context. But sometimes a concept is, by design, coupled with another generic concept or with an instance of one. The global context section makes this possible.

The reason for coupling a conceptual module to another conceptual module (but not an instance of it) is to describe some instance of the latter in the parametric or local context of the former. For example, `Stack_Template` is part of the global context of `Stack_Replica_Capability` because of the need to declare the formal parameters `Stack_Facility_1` and `Stack_Facility_2` in the parametric context section. This illustrates another recommendation of the RESOLVE discipline: To specify an additional operation for a kernel concept, put it in a separate conceptual module and use global and parametric context to connect the new concept to the underlying one.

The facilities most commonly included in global context are those defining the scalar types `Boolean`, `Character`, and `Integer`. (There is also a built-in `Real`, but we do not discuss or use it here.) These types really are not built-in to RESOLVE in the usual sense, because we can specify them formally using ordinary conceptual modules and define a useful but austere dialect of RESOLVE without them. In practice, however, it is important to have syntactic sugar comparable to that in other imperative languages so programming with ordinary scalar types isn’t unbearably tedious. We therefore use a richer dialect of RESOLVE in which these types behave as they do in a typical imperative language; in which you may write expressions involving these types as in a typical imperative language; but in which you still must explicitly list any dependence on these types as though they were undistinguished. The library contains instances of standard realizations for the conceptual modules that define these special scalar types — called “`Standard_Boolean_Facility`”, “`Standard_Character_Facility`”, and “`Standard_Integer_Facility`” — and these belong in the global context section of any

conceptual module that needs them. Practically speaking, this need arises only from declaring an operation in the interface section (see Section 3.2.3) that has a formal parameter or result of such a type.

3.1.2. Parametric Context

The parametric context section lists the formal parameters of a conceptual module, along with restrictions on them, if any. There are four kinds of parameters to conceptual modules: (program) constants, (program) types, (program) facilities, and math operations:

```
<concept_parametric_context_section> ::=
  parametric context
  <concept_parameter_sequence>
  [<concept_parametric_context_restriction>]

<concept_parameter> ::=
  <concept_constant_parameter> |
  <concept_type_parameter> |
  <concept_facility_parameter> |
  <concept_math_operation_parameter>

<concept_constant_parameter> ::=
  constant <variable_id> : <type_id>

<concept_type_parameter> ::=
  type <type_id>

<concept_facility_parameter> ::=
  facility <facility_id> is <conceptual_module_id>
  [( <concept_facility_argument_list> )]

<concept_math_operation_parameter> ::=
  <math_operation_header_declaration>

<concept_parametric_context_restriction> ::=
  restriction <assertion>
```

A constant parameter introduces a local name for a particular value that must be supplied by a client. A typical use of this feature is to allow a client to choose a size limit, as in a bounded version (not shown) of `Stack_Template`.

A type parameter introduces a local name for the type that a client must provide to instantiate the module. For a type parameter, the client may supply any type.

Having constant and type parameters is the usual meaning of the phrase “generic component”. But there are two other important kinds of parameters. The first is a facility parameter, for which a client must provide an instance of the named conceptual module whose arguments match the parameters in the formal’s declaration. Figure 4 illustrates

how to “tie together” parameters of facilities using this mechanism. It restricts a client to providing, for `Stack_Facility_1` and `Stack_Facility_2`, instances of `Stack_Template` that define Stacks with the same Items.

A math operation parameter introduces a local name for some mathematical machinery to explain the effects of program operations. This declaration constrains the actual to have a particular signature. Math operations are discussed in more detail elsewhere [Heym 94a].

The optional `restriction` clause allows you to require certain properties of the actuals passed for constant or math operation parameters. We do not illustrate a restriction here, but a prototypical example involves a conceptual module that specifies sorting [Weide 94c]. In this case, a formal math operation parameter — a binary relation in form — defines the (mathematical) ordering to be used for sorting. The restriction makes sure that this relation is a total ordering. Each realization of this module then has an additional parameter [Part III] — a program operation — through which a client says how to compute this ordering.

3.1.3. Local Context

The local context section defines information needed to specify the module’s interface but which is not obtained from the library or from the client. It can introduce mathematical machinery, instances of other concepts, and/or module-level state information:

```

<concept_local_context_section> ::=
  local context
    [<concept_local_context_item_sequence>]
    [<concept_local_context_state_variables_section>]

<concept_local_context_item> ::=
  <math_subtype_declaration> |
  <math_operation_header_declaration> |
  <math_operation_definition_declaration> |
  <concept_facility_declaration>

<concept_facility_declaration> ::=
  facility <facility_id> is <conceptual_module_id>
    [( <concept_facility_argument_list> )]

<concept_local_context_state_variables_section> ::=
  state variables
    <concept_local_context_state_variable_sequence>
    [constraint <assertion>]
    [<concept_initialization_declaration>]
    [<concept_finalization_declaration>]

<concept_local_context_state_variable> ::=
  <variable_id> : <math_type_id>

```

The mathematical local context items are discussed elsewhere [Heym 94a], because math subtypes and math operations that might be expected to be reused should be defined in separate units called math modules. For an example of defining mathematics locally, too, see Figure 3. The basic idea is that the local context section introduces some names and definitions that unify and/or simplify the behavioral description of the exported functionality.

Some concepts are conveniently specified by referring to (i.e., reusing) other concepts. The thesis here is that if a reader already understands the reused component, then the new component probably is easier to comprehend than if it were specified from scratch. One application of this technique is to define enhancements, another to define specializations. In either case you first declare an instance of the reused concept by fixing all of its parameters, then use that facility to explain the interface of the new conceptual module. You may make the facility a parameter or you may declare it in the local context section. The latter often is preferred for an enhancement because it leaves the parametric context section as simple as possible and leaves open the possibility of direct implementation of the newly-defined interface, as discussed in Section 2.

RESOLVE permits only a safe version of this kind of “specification inheritance” [Muralidharan 90, Sitaraman 91, Edwards 93b]. There is no way to override the specification of any operation or to change or delete any operation from an interface; that would make it a different concept. The only allowable effect is to *re-export* a facility’s interface *in toto*, as explained in Section 3.2.

The other kind of information that may appear in the local context section is a list of module-level *state variables*. Each instance of a concept with state variables has its own set of them. The operations declared in the interface section may have side-effects on the values of the state variables. But state variables are not program variables. There are two ways to see this: Their declared types are mathematical types [Heym 94a], not programming types; and they are not exported to client programs (notice that they are not in the interface section) so there is no way to write code to manipulate them directly. State variables appear only in assertions in their role as mathematical models of state information that is shared among the operations specified in the interface section.

The state variables section includes three optional assertions. The first, a **constraint** clause, defines an invariant relationship among the state variables that holds at all times known to a client, i.e., before and after invocation of every operation exported by the component’s interface. This assertion helps explain properties of facility state, and introduces a proof obligation for the specifier to show — by the specification counterpart of data-type induction — that the constraint is an invariant property.

The optional initialization and finalization assertions define properties that hold for the state variables when the concept is instantiated, and when it ceases to exist. The former assertion is almost always present, as it tells a client what to expect before calling the first operation exported by an instance of this concept and provides the base case for the induction argument mentioned above. The latter assertion is rarely needed, as it can refer

only to state variables of other facilities which are part of the context, and this would imply serious coupling between modules that should be avoided in most cases. For details of initialization and finalization (of types, which is analogous), see Section 3.2.2.

Figure 6: A Poorly-Designed Component

```
concept Single_Stack
  context
    global context
      facility Standard_Integer_Facility
    parametric context
      type Item
    local context
      state variables
        s: string of math[Item]
      initialization
        ensures s = empty_string
  interface
    operation Push (
      consumes x: Item
    )
      referenced state variables
        alters s
      ensures s = <#x> * #s
    operation Pop (
      produces x: Item
    )
      referenced state variables
        alters s
      requires s /= empty_string
      ensures #s = <x> * s
    operation Length: Integer
      referenced state variables
        preserves s
      ensures Length = |s|
  end Single_Stack
```

Consider Figure 2 and compare it to Figure 6, which shows how you can specify stack-like behavior using state variables. The latter design is perfectly legal in RESOLVE, and

the proof system can deal with it, but the RESOLVE discipline counsels against it [Hollingsworth 92b]. There are many problems with Single_Stack: the need for one facility declaration per stack needed by a client; inconvenience in dealing with binary operations such as Replica that are handled easily by the design that exports a type; and a host of other annoyances. The final blow is that you can layer Single_Stack on top of Stack_Template, but not vice versa. Thus, there is no reason to recommend Single_Stack over Stack_Template as the design for a kernel concept.

It is easy to do without state variables for most concepts; it is not so easy for others. For example, because there is no built-in “pointer” type constructor in RESOLVE, the RESOLVE library includes a few concepts that encapsulate various kinds of safe manipulations of pointers [Hollingsworth 92a, Hollingsworth 92b]. These concepts have state variables which, in fact, gives them the power to exhibit pointer-like features. Similarly, dealing with I/O and persistence requires state variables. But if possible the use of state variables should be avoided in favor of exported types.

3.2. Interface Section

The interface section is the second major part of a conceptual module. It defines the programming identifiers that an instance of the module exports to a concrete client for use in writing executable code; but recall that the concept also implicitly exports all identifiers that are visible within it for purposes of writing mathematical statements in either abstract or concrete clients:

```

<concept_interface_section> ::=
  interface
    [re-exports
      <concept_re-exported_item_sequence>]
    [<concept_interface_item_sequence>]

<concept_re-exported_item> ::=
  <facility_id>
    [<concept_renaming_section>]

<concept_renaming_section> ::=
  renaming
    <concept_renaming_item_sequence>

<concept_renaming_item> ::=
  <type_id> as <type_id> |
  <operation_id> as <operation_id>

<concept_interface_item> ::=
  <type_declaration> |
  <operation_declaration>

```

The interface section for a typical kernel concept declares a type and primary operations. If you follow the RESOLVE discipline, these operations form a minimal basis for all intended manipulations of variables of the exported type. They do not, of course, include all operations anyone might ever want to do with that type; there is no way in advance to anticipate this. So a conceptual module can define additional operations for previously-defined types, building on other abstract components as illustrated in Figure 4.

3.2.1. Re-exported Interfaces

In order to support enhancements and specializations, RESOLVE lets you *re-export* from a concept C the entire interface of any instance of another concept C' that appears in C's global, parametric, or local context. Figure 5 illustrates this feature. We emphasize the re-exporting is used here only for specification. Although it is convenient to do so for a simple realization [Part III], there is no requirement that every implementation make a connection between the newly-defined interface and the re-exported interface.

When C re-exports an instance of C', it may rename any type or operation. The new name replaces the original one in C's interface, but in all other respects the interface of C' is simply passed through. In principle, renaming is occasionally necessary to prevent naming conflicts (e.g., two types with the same name, or two operations with the same name and same type signature). There is no way to use qualified names for this purpose because an entire interface is re-exported and no individual items may be qualified.

Re-exporting is more than mere notational shorthand. Because there is no way to modify a re-exported interface as it passes through into the new interface, in the scenario above the interface of C certainly includes as a subset the interface of C' (with possible renaming of some items). So, if a client program needs to supply as an actual parameter an instance of C' — or any other concept whose interface is re-exported by C — then it may instead provide an instance of C.

For example, suppose you want to instantiate `Stack_Top_Capability` in Figure 4. Apparently this requires an instance of `Stack_Template` as a parameter. However, you may pass an instance of `Replicable_Stack_Template` because its interface re-exports that of `Stack_Template`. On the other hand, if `Replicable_Stack_Template` were specified by copying-and-pasting from `Stack_Template` to define precisely the same interface behavior, then this substitution of instances would not be allowed by the language even though it should “work” in principle.

Another way of understanding re-exporting is to note that, while a facility parameter appears on the surface to be an extensionally-defined parameter [Edwards 93b], re-exporting makes it a bit more flexible. A facility parameter is not, however, intensionally-defined. Additional flexibility in this dimension will be added to RESOLVE in the near future [Edwards 94].

3.2.2. Types

When declaring a programming type (family), you give it a name and an associated mathematical model, as well as information about initial and final values:

```
<type_declaration> ::=
  type <type_id> is modeled by <math_type_expression>
  exemplar <variable_id>
  [<concept_initialization_declaration>]
  [<concept_finalization_declaration>]

<concept_initialization_declaration> ::=
  initialization
  [<state_variable_reference_declaration>]
  [requires <assertion>]
  [ensures <assertion>]

<concept_finalization_declaration> ::=
  finalization
  <state_variable_reference_declaration>
  [requires <assertion>]
  [ensures <assertion>]
```

Every program type has a mathematical model that you use to explain its behavior, as discussed in Section 1. The RESOLVE mathematical notation contains a few built-in mathematical types and type constructors that can be used to describe sophisticated mathematical models. You also can define new mathematical types for the rare occasion when that becomes necessary [Heym 94a].

In the RESOLVE discipline, a typical program type is parameterized by another one. When you need to mention the math type of the model of a programming type T , e.g., in declaring a parameterized type's mathematical model type, you call it `math[T]`. The notation emphasizes that it is the math model of T that is of interest, not T itself. (A “conversion” could be made implicitly, but we prefer the clarity offered by this construct as a way to highlight the distinction between programming and mathematical types.)

A programming type always has an *exemplar*, i.e., an identifier that stands for an arbitrary value of the type in assertions that may be attached to the type declaration. You must provide an exemplar even if these assertions are not present because this identifier also appears in other places, e.g., in the representation correspondence clause [Part III].

A RESOLVE operation never has a precondition like “ x has been initialized” because all variables are automatically initialized and finalized upon entry to and exit from their scope, respectively. To support this feature and to support primitive data movement, every program type has three operations that do not appear explicitly in the list of exported operations:

- The Initialize operation is invoked by a compiler-generated call at the beginning of the scope where its argument is declared. It gives that variable (in whose place the exemplar stands in the specification) an initial value satisfying the **initialization ensures** assertion of the type declaration; see Figures 2 and 3.
- The Finalize operation is invoked by a compiler-generated call at the end of the scope where its argument is declared. This operation is really a hook for the realization to release resources (e.g., memory) used by the type representation, but if there are also state variables in a concept it may have a visible effect, too. Only then is its effect specified.
- The swap operation (invoked in a client by using the infix “:=” operator) exchanges the values of its two arguments [Harms 91].

Typically, a type declaration includes a specification of initialization. A totally missing **initialization** clause means that its **requires** and **ensures** assertions are **true**, i.e., that a client knows nothing about the initial value of a variable of this type — except that it *has* a value of the type and therefore is *not* “uninitialized” or “undefined”. Ordinarily, a type declaration does not include a **finalization** clause, because there is no conceptual effect to the Finalize operation.

Initialization and finalization present some interesting problems when there are state variables. If either has a precondition involving a state variable — in the case of initialization this is the only thing a precondition could mention — or any side-effect on a state variable, then that variable must be listed explicitly in a state variable reference clause. The nature of any side-effect then must be specified in the **ensures** clause. We more fully discuss the declaration of references to state variables in the next section, for exported operations in general.

3.2.3. Operations

An operation specification consists of a header that defines its structural interface, an optional list of references to state variables, and assertions that define its input/output behavior:

```

<operation_declaration> ::=
  <operation_header_declaration>
  [<state_variable_reference_declaration>]
  [requires <assertion>]
  [ensures <assertion>]
```

An operation may be *procedural*, in which case it operates by changing the values of some of its arguments and/or the state variables it references, but does not return a value; or it may be *functional*, in which case it may not change any argument or state variable,

but operates by returning a value. The operation header is slightly different for these two cases:

```
<operation_header_declaration> ::=
  operation <operation_id>
  [( <operation_formal_parameter_sequence> )]
  [: <type_id>]

<operation_formal_parameter> ::=
  <abstract_mode> <variable_id> : <type_id>

<state_variable_reference_declaration> ::=
  referenced state variables
  <state_variable_reference_item_sequence>

<state_variable_reference_item> ::=
  <abstract_mode> <variable_id>

<abstract_mode> ::=
  alters |
  produces |
  consumes |
  preserves
```

You specify the effect of an operation using a **requires** clause (precondition) and an **ensures** clause (postcondition), which are assertions about the values of the mathematical models of the operation’s parameters and any referenced state variables. A variable stands for an abstract mathematical value — not a reference to it or the variable’s memory address or any other programming-specific notion. In a **requires** clause a variable stands for its value at the beginning of a call. In an **ensures** clause a variable stands for its value at the end of the call, while a variable with a prefixed “#” (pronounced “old”) stands for the value of that variable at the beginning of the call.

An operation’s specification is treated as a (partial) relation on the space of input and output values. The precondition tells where the relation is defined, and the postcondition defines it there. Operationally, the contract between client and implementer is as follows: If a client calls an operation in a state in which the **requires** clause holds when the actuals are substituted for the formals, then the implementer guarantees that the operation will return in a state in which the **ensures** clause holds with the same substitution. But if the **requires** clause does not hold when the call occurs, then the implementer makes no guarantees whatsoever.

Operation specifications are considerably simplified by using *abstract modes*:

- An **alters**-mode variable (either an operation parameter or a referenced state variable) is potentially changed by executing the operation. You can determine how by reading the **ensures** clause, which generally involves the old value as well as the new value of the variable.

- A **produces**-mode variable gets a new value that is defined by the **ensures** clause. But the **ensures** clause should not involve the variable's old value because it is irrelevant to the operation's effect.
- A **consumes**-mode variable gets a new value that is an initial value for its type, but its old value generally is relevant to the operation's effect. If x is a **consumes**-mode variable, then it is as if the clause "**and is_initial** (x)" were implicitly appended to the postcondition.
- A **preserves**-mode variable suffers no net change in value between the beginning of the operation and its return, although its value might be changed temporarily while the operation is executing. All parameters to, and state variables referenced by, a functional operation must have **preserves** mode. If x is a **preserves**-mode variable, then it is as if the clause "**and** $x = \#x$ " were implicitly appended to the postcondition.

It is important to recognize that abstract modes have nothing to do with how parameters are passed to operations. All parameters are passed *by swapping*, which (under certain conditions that are part of the realization language [Part III]) is operationally tantamount to call-by-reference in traditional languages [Harms 91]. Abstract modes are purely specification notations that aid understandability and shorten some specifications dramatically.

4. Conclusion

RESOLVE's specification language is rich and fairly complex — but (we claim) no more so than it has to be in order to express the mathematical models and functional behavior that must be captured in formal software specifications. The examples here illustrate most of the constructs for simple cases. We have designed significantly more involved conceptual modules that have convinced us of the need for all the mechanisms RESOLVE has now. They also have suggested many improvements that we are considering for the future.