CHAPTER V

An Extended ACTI Example

This chapter provides an extension of the running Partial_Map example introduced in Chapter III, to illustrate some of the more advanced aspects of ACTI. It demonstrates the use of interpretation mappings between abstract and concrete instances, and explains the semantics of subsystem-level program statements such as generic instantiations in terms of the ACTI model. This is done by examining the effect that the execution of a single RESOLVE facility instantiation statement has on a basis environment modeling the state of a program. After describing the instantiation example and the initial basis environment, this chapter shows how to instantiate an abstract template, instantiate a concrete template, instantiate an interpretation mapping template, combine the three instances, and embed the result back in the original basis environment. A good understanding of RESOLVE will be helpful in fully appreciating the example presented in chapter.

5.1 Facility Instantiation in RESOLVE

Chapter III introduced a Partial_Map ADT, which is modeled as a set of ordered pairs of D_Items and R_Items. Each ordered pair in a Partial_Map object represents one key-to-value association stored in that map, and the map as a whole is simply a collection of such associations for some group of unique keys (D_Items). This abstract notion of a set of key-value associations can be used for many purposes where database-like behavior is required, such as in symbol tables, dictionaries, or associative memories.

Chapter III also introduced the idea of a “communally bounded” version of an ADT. The Communal_PMap_Template concept presented in Figures 22 and 23 describes a type Partial_Map that is communally bounded—it has an upper limit on the aggregate number of associations stored at any one time in all Partial_Map objects. Here, we further examine the Communal_PMap_Template concept, a RESOLVE realization for this concept, and how the two are instantiated to form a facility, or instance module, that can then be used in a program.
facility Int_Int_Pair_Facility is
    Record2_Template (Integer, Integer)
    realized by
    Standard

facility Int_Int_Stack_Facility is
    Stack_Template (Int_Int_Pair_Facility.Record2)
    realized by
    List

facility Int_Multiset is
    Communal_PMap_Template (Integer, Integer, 1024)
    realized by Unordered_Stack (Int_Int_Pair_Facility,
    Int_Int_Stack_Facility,
    Int_Multiset_Facility,
    Are_Equal)

Figure 30: Instantiating Communal_PMap_Template in RESOLVE

Figure 30 shows the RESOLVE syntax for instantiating Communal_PMap_Template to create a facility Int_Multiset. Suppose this facility is being created by a programmer implementing another abstraction, say a multiset of integers. Such a multiset could be conveniently represented by just storing the distinct elements of the multiset, and then storing an occurrence count for each of those distinct elements. A partial map could be used for this purpose.

Box (a) of Figure 30 shows Communal_PMap_Template being instantiated with the program type Integer as its D_Item (the kind of elements in the multiset), Integer as its R_Item (the occurrence count for the corresponding element), and 1024 as the max_total_size limit. Further, box (a) of Figure 30 shows that the Unordered_Stack realization of Communal_PMap_Template is being selected to provide the implementation for Int_Multiset facility. This realization, adapted directly from an unbounded implementation presented by Bucci et al. [2, pp. 42–44], was chosen for simplicity and understandability. It represents each Partial_Map object as a stack of ordered pairs, each containing one D_Item and one R_Item. The stack is not maintained in any particular order, hence the name Unordered_Stack.
5.1. FACILITY INSTANTIATION IN RESOLVE

There are two additional instantiation statements present in Figure 30, one for Int_Int_Pair_Facility and one for Int_Int_Stack_Facility. These statements are there to declare instances of other components that will be used by the implementation of Int_Multiset, and they are thus provided as parameters to Unordered_Stack as part of the instantiation in box (a). Int_Int_Pair_Facility defines a type Record2 that is an ordered pair of two integers. Int_Int_Stack_Facility defines a type Stack that is a stack of Record2 ordered pairs. It is clear how these two pieces can be used to construct the Unordered_Stack realization of the Communal_PMap_Template concept, and the source code for this implementation will be presented in Section 5.5.

In ACTI terms, Communal_PMap_Template is an abstract template, Unordered_Stack is a concrete template, and the three facilities Int_Int_Pair_Facility, Int_Int_Stack_Facility, and Int_Multiset, are all concrete instances. Further, implicitly there is an interpretation mapping that describes why the behavior of Unordered_Stack, written in terms of Record2 and Stack objects, conforms with the more abstract behavior specified in Communal_PMap_Template, written in terms of sets of ordered pairs. In this chapter, we look at how the single RESOLVE statement in box (a) of Figure 30 can be interpreted in ACTI terms.

facility Int_Multiset is
   Communal_PMap_Template (   
      Standard_Integer_Facility.Integer, 
      Standard_Integer_Facility.Integer, 
      1024)  

realized by
   Unordered_Stack (   
      Standard_Integer_Facility.Integer, 
      Standard_Integer_Facility.Integer, 
      1024, 
      Int_Int_Pair_Facility, 
      Int_Int_Stack_Facility, 
      Standard_Integer_Facility.Are_Equal)  

interpreted by
   UStack_To_CMap (   
      Standard_Integer_Facility.Integer, 
      Standard_Integer_Facility.Integer)  

Figure 31: A More Detailed View of the Instantiation
CHAPTER V. AN EXTENDED ACTI EXAMPLE

To set the stage for this, it is important to note that in RESOLVE, all of the parametric context to `Communal_PMap_Template` is available inside `Unordered_Stack`. Figure 31 makes explicit the fact that the realization is parameterized by the concept’s parameters. This is done by repeating the concept’s parameters in the arguments to the realization. Figure 31 also makes explicit the interpretation mapping that describes the relationship between the results of the abstract template and the concrete template. Here, this interpretation mapping is called `UStack_To_CMap`, and it will be discussed in detail in Section 5.6.

Making these details visible (where they are implicit in RESOLVE as currently defined) allows us to view the semantics of the single RESOLVE-like statement in Figure 31 in ACTI terms. The instantiation involves five distinct steps:

1. Instantiating `Communal_PMap_Template` (an abstract template) to produce an abstract instance.

2. Instantiating `Unordered_Stack` (a concrete template) to produce a concrete instance.

3. Instantiating `UStack_To_CMap` (an interpretation mapping template) to produce an interpretation mapping.

4. Combining the results of steps (1), (2), and (3) to form a new facility, which is a concrete instance.

5. Associating this newly created concrete instance with the name `Int_Multiset` in our model of the program’s state.

After laying the groundwork for discussing this example in ACTI terms, we explore each of these steps in greater detail.

5.2 Notation

Throughout Chapter III an informal graphical representation of subsystems was used, as typified by Figure 26 (p. 53). Given the formal definition of the ACTI mathematical spaces in Chapter IV, it is now possible to give such figures a more rigorous meaning. Figure 32 summarizes the formal definition of an ACTI abstract instance.

The outermost box in Figure 32 represents an entire abstract instance, and all other module-like structures—abstract instances, concrete instances, and specification adornment environments—within that abstract instance are also depicted using boxes. The outermost abstract instance is actually a 12-tuple; the individual components
of the tuple are presented vertically and identified by their names, as defined in Section 4.6. All other tuple objects will be represented similarly. The name-to-value mappings within the abstract instance in Figure 32 are represented using the notation introduced in Section 4.1.2, laid out vertically. Interpretation mappings are shown simply as upward-pointing double-arrows, and will be fully described in separate figures when required. Similarly, templates are graphically represented as functions (arrows) from a parameter to a result.

To simplify subsystem diagrams of this form, we will adopt the convention that only non-$\perp$ components of tuple objects need be shown. Thus, any tuple fields that are omitted are understood to have the $\perp$ value from their corresponding domain.
The only exceptions to this rule are invariants, boolean predicates that when omitted are understood to be universally true.

5.3 A Complete Computational Environment

Intuitively, the instantiation of Communal_PMap_Template presented in Figure 30 declares a new program unit, the facility Int_Multiset, that can be used in later program statements. In order to meaningfully discuss the formalization of this intuition, one needs a formal analog of the “state” of a program. This sets the groundwork for presenting the effect of the Int_Multiset instantiation statement, in terms of how that statement alters the state that the program was in before the instantiation was executed.

In ACTI, the natural denotation for a program state is a Basis Environment. Figure 33 shows a basis environment BE that represents the state of a program before the execution of the Int_Multiset instantiation. Figure 33 shows all of the components of BE, but using the conventions presented in Section 5.2, we can abbreviate BE by eliding ⊥-valued components. Figure 34 shows BE, omitting all of the components with “uninteresting” values.

The basis environment BE records the assert status (section 4.1.6) of the program state, and an environment (concrete instance) E that associates identifiers with values. BE includes four nested concrete instances, each representing a particular RESOLVE facility. Two of the four are standard facilities defining the programming types Boolean and Integer. The remaining two are the result of the first two instantiations shown in Figure 30. BE also associates values with Communal_PMap_Template, Unordered_Stack, and UStack_To_CMap, each of which will be discussed in a separate section. Finally, BE has a size_limit specification adornment variable with the integer value 1024.

To fully understand the signature (i.e., type) of size_limit, a brief explanation of the mathematical models for the traditional scalar types in RESOLVE is useful\(^1\). In RESOLVE, the reserved words boolean and integer are used to refer to the usual mathematical (not programming) notions, which are built-in to the RESOLVE specification notation for convenience. The corresponding programming types are not built-in to the language, but are instead defined in facilities, just like user-defined programming types. As a result, the program identifiers “boolean” and “integer” are overloaded, each representing both a mathematical type used for model-based specifications, and a programming type used to construct implementations. Because

\(^1\)See [66] for a more detailed description.
5.3. A COMPLETE COMPUTATIONAL ENVIRONMENT

\[ AS = NL \]

\[ CTXT = \perp \]

\[ SATY = \begin{cases} \text{math-integer} & \mapsto \frac{MD}{\text{CNSTR}} = Z \text{ TRUE} \\ \text{size_limit} & \mapsto \frac{VSIG}{\text{math-integer}} \text{ VAL} = 1024 \end{cases} \]

\[ SAE = \]

\[ SAV = \]

\[ TE = \perp \]
\[ VE = \perp \]
\[ OE = \perp \]
\[ EI = \text{true} \]
\[ AIE = \perp \]

\[ BE = \]

\[ E = \]

\[ CIE = \begin{cases} \text{Standard Boolean Facility} & \mapsto \square \\ \text{Standard Integer Facility} & \mapsto \square \\ \text{Int Int Pair Facility} & \mapsto \square \\ \text{Int Int Stack Facility} & \mapsto \square \end{cases} \]

\[ IME = \perp \]

\[ ATE = \{ \text{Communal PMap Template} \mapsto (\square \rightarrow \square) \} \]

\[ CTE = \{ \text{Unordered Stack} \mapsto (\square \rightarrow \square) \} \]

\[ IMTE = \{ \text{UStack To CPMap} \mapsto (\square \rightarrow \uparrow) \} \]

Figure 33: An Initial Basis Environment \( BE \)
of the ubiquitous nature of these scalar programming types, a RESOLVE programming
environment will contain a “standard” implementation for each of them.

By convention, the RESOLVE literature uses boolean or integer (in bold face)
as a reserved word to refer to the corresponding mathematical type, and Boolean or
Integer (with initial capitalization) to refer to the programming type. Because all
ACTI identifiers are assumed to be unique, throughout this example “math-boolean”
(or “math-integer”) will be used to refer to the mathematical type denoted by the
corresponding RESOLVE reserved word, and “prog-Boolean” (or “prog-Integer”)

![Diagram](image-url)
5.3. A COMPLETE COMPUTATIONAL ENVIRONMENT

\[
\begin{align*}
AS &= NL \\
\text{SAE} &= \begin{cases}
SATY = \{ \text{math-integer} \mapsto [MD = Z] \\
CNSTR = \text{true} \\
SAVE = \{ \text{size_limit} \mapsto [VSIG = \text{math-integer}] \\
VAL = 1024 \}
\end{cases}
\end{align*}
\]

\[
BE = \\
\begin{align*}
E &= CIE = \\
\begin{cases}
\text{Standard.Boolean.Facility} \mapsto \square \\
\text{Standard.Integer.Facility} \mapsto \square \\
\text{Int.Int.Pair.Facility} \mapsto \square \\
\text{Int.Int.Stack.Facility} \mapsto \square \\
\text{Int.Multiset} \mapsto \square
\end{cases}
\end{align*}
\]

\[
ATE = \{ \text{Communal.PMap.Template} \mapsto (\square \rightarrow \square) \\
CTE = \{ \text{Unordered.Stack} \mapsto (\square \rightarrow \square) \\
IMTE = \{ \text{UStack.To.CMap} \mapsto (\square \rightarrow \uparrow) \}
\]

Figure 35: BE After Instantiating Communal.PMap.Template

will be used to refer to the programming type. Further, for the purposes of the figures in this chapter, we will present both the mathematical and programming types as being defined within the corresponding standard facility. Of course, defining the denotational semantics of RESOLVE within the ACTI model would naturally involve completely defining an initial basis environment capturing all of the types, operations,
facilities, and templates that are built-in to the language or provided as part of the standard programming environment. That is beyond the scope of this dissertation, and not essential to understanding the example.

The **Int_Multiset** instantiation statement presented in Figure 30 has the simple effect of adding one more name-to-value association to the **BE.E.CIE**—one that associates **Int_Multiset** with a context-bound concrete instance representing the facility produced by the instantiation statement. After executing the **Int_Multiset** instantiation, the resulting program state **BE** is the one depicted in Figure 35. This is the high-level view of the semantics of this single program statement. We will now proceed to describe the five smaller steps that are combined to achieve this effect.

### 5.4 Instantiating **Communal_PMap_Template**

As described in Section 5.1, the first step in the **Int_Multiset** instantiation is applying the abstract template **Communal_PMap_Template**. **Communal_PMap_Template** was first introduced in Chapter III, Figures 22 and 23.

As indicated in Chapter III, it is possible to generalize the **PARTIAL_FUNCTION** adornment definitions and move them into a separate program unit so that they can be reused in many specifications. Figures 36 through 38 present a slight variation of **Communal_PMap_Template** where this has been done. Here, the specification adornment definition of the type **PARTIAL_FUNCTION**, as well as several operations on partial functions, have been moved into a parameterized RESOLVE mathematics module [28]. This mathematics module is called **PARTIAL_FUNCTION THEORY TEMPLATE**, and is presented in Section 5.4.1. As shown in Figure 36, this mathematics module is imported and instantiated by **Communal_PMap_Template**. Examining the meaning of **PARTIAL_FUNCTION THEORY TEMPLATE**, and how it might be instantiated, will help in the understanding of **Communal_PMap_Template**.

#### 5.4.1 Instantiating **PARTIAL_FUNCTION THEORY TEMPLATE**

**PARTIAL_FUNCTION THEORY TEMPLATE** is a RESOLVE parameterized mathematics module. Mathematics modules only contain mathematical definitions used in the specification of program types and operations. Because of this, it is natural to use a **Specification Adornment Environment** as the denotation for a mathematics module. Similarly, a **Specification Adornment Template** is the natural denotation for a parameterized mathematics module. Figures 39 and 40 present the RESOLVE definition of **PARTIAL_FUNCTION THEORY TEMPLATE**.
5.4. INSTANTIATING COMMUNAL_PMAP_TEMPLATE

concept Communal_PMap_Template

case

global context

facility Standard_Boolean_Facility
facility Standard_Integer_Facility

mathematics PARTIAL_FUNCTION_THEORY_TEMPLATE

parametric context

  type D_Item
  type R_Item

  constant max_total_size : integer
    restriction
    max_total_size > 0

local context

  math facility PARTIAL_FUNCTION_THEORY is
  PARTIAL_FUNCTION_THEORY_TEMPLATE ( math[D_Item], math[R_Item] )

  state variables
  total_size : integer
    constraint
    0 ≤ total_size ≤ max_total_size
    initialization
    ensures total_size = 0

interface

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

Figure 36: The Communal_PMap_Template Concept
operation Define (  
    alters m : Partial.Map  
    consumes d : D.Item  
    consumes r : R.Item  
  )

referenced state variables  
    alters total.size  
    requires not DEFINED_IN (m, d) and  
        total.size < max.total.size  
    ensures m = #m union {(#d, #r)} and  
        total.size = #total.size + 1

operation Undefine (  
    alters m : Partial.Map  
    preserves d : D.Item  
    produces d_copy : D.Item  
    produces r : R.Item  
  )

referenced state variables  
    alters total.size  
    requires DEFINED_IN (m, d)  
    ensures (d, r) is in #m and  
        m = #m - {(#d, #r)} and  
        d_copy = d and  
        total.size = #total.size - 1

operation Undefine.Any.One (  
    alters m : Partial.Map  
    produces d : D.Item  
    produces r : R.Item  
  )

referenced state variables  
    alters total.size  
    requires m ≠ empty.set  
    ensures (d, r) is in #m and  
        m = #m - {(#d, r)} and  
        total.size = #total.size - 1

Figure 37: The Communal_PMap_Template Concept (continued)
5.4. INSTANTIATING COMMUNAL_PMAP_TEMPLATE

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|

dend Communal_PMap_Template

Figure 38: The Communal_PMap_Template Concept (continued)

mathematics PARTIAL_FUNCTION_THEORY_TEMPLATE

collection  
parametric  context

math type DOMAIN_ITEM

math type RANGE_ITEM

type
collection  
exemplar  m

constraint  
for all d : DOMAIN_ITEM, r1, r2 : RANGE_ITEM  
where ((d, r1) is in m and (d, r2) is in m)  
(r1 = r2)

Figure 39: The PARTIAL_FUNCTION_THEORY_TEMPLATE Mathematics
math operation EMPTY_PARTIAL_FUNCTION : PARTIAL_FUNCTION
    explicit definition
    empty_set

math operation DEFINED_IN ( 
    m : PARTIAL_FUNCTION
    d : DOMAIN_ITEM 
) : boolean
    explicit definition
    there exists r : RANGE_ITEM ((d, r) is in m) (b)

math operation DIFFER_ONLY_AT ( 
    m1 : PARTIAL_FUNCTION
    m2 : PARTIAL_FUNCTION
    d_items : set of DOMAIN_ITEM 
) : boolean
    explicit definition
    for all dr_pair : ( 
        d : DOMAIN_ITEM
        r : RANGE_ITEM
    ) where (dr_pair.d is not in d_items) 
        (dr_pair is in m1 iff dr_pair is in m2) (c)

end PARTIAL_FUNCTION_THEORY_TEMPLATE

Figure 40: The PARTIAL_FUNCTION_THEORY_TEMPLATE Mathematics (continued)

A specification adornment template is a function from specification adornment environments to specification adornment environments, as shown in Figure 41. Thus, PARTIAL_FUNCTION_THEORYTEMPLATE takes a single specification adornment environment as its parameter, and produces a single specification adornment environment as its result. As indicated by the parametric context in Figure 39, PARTIAL_FUNCTION_THEORY_TEMPLATE expects two types (or, in ACTI terms, a specification adornment environment providing two types) as its parameter, and there are no restrictions on what abstract models can be used for these types. In ACTI terms, we can say that the specification adornment template denoted by PARTIAL_FUNCTION_THEORY_TEMPL-
5.4. INSTANTIATING COMMUNAL_PMAP_TEMPLATE

<table>
<thead>
<tr>
<th>Parameter (A Specification Adornment Instance)</th>
<th>PARTIAL_FUNCTION_THEORY_TEMPLATE</th>
<th>Result (A Specification Adornment Template)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(A Specification Adornment Template)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 41: PARTIAL_FUNCTION_THEORY_TEMPLATE is a Specification Adornment Template

PLATE will accept as its parameter any specification adornment environment matching the one in Figure 42. Effectively, the domain predicate (e.g., SATS.SAIDP) for this template is true for any specification adornment environment matching Figure 42, and false for others.

Similarly, we can characterize both the effect predicate (e.g., SATS.SAIEP) and the specification adornment instance function (SAISAIF) associated with the specification adornment template PARTIAL_FUNCTION_THEORY_TEMPLATE using another figure to show the structure of the result produced by the template. Figure 43 shows the specification adornment instance that results from applying this template to a parameter matching Figure 42. This result depends on the specific Type Model values associated with the identifiers DOMAIN_ITEM and RANGE_ITEM in the parameter. If the parameter maps DOMAIN_ITEM to the pair ⟨MD₁, CNSTR₁⟩, then in Figure 43 we use DI-Model to refer to the space CNSTR₁(MD₁). RI-Model is defined similarly.

The type PARTIAL_FUNCTION in the resulting specification adornment instance has a type model that is the power set of pairs of domain and range items (e.g., \( P(\text{DI-Model} \times \text{RI-Model}) \)). The constraint over this mathematical domain is defined in box (a) of Figure 39. The resulting specification adornment instance also defines two other types, here named Anonymous1 and Anonymous2. These types are used as the types of mathematical operations defined in the mathematics module, but are not specifically given program identifiers in the RESOLVE description of the template.

\[
SATY = \begin{cases} 
\text{DOMAIN_ITEM} & \mapsto \text{<Any model, say DI-Model>} \\
\text{RANGE_ITEM} & \mapsto \text{<Any model, say RI-Model>}
\end{cases}
\]

Figure 42: PARTIAL_FUNCTION_THEORY_TEMPLATE’s Parameter
\[
SATY = \begin{cases}
\text{PARTIAL\_FUNCTION} & \mapsto \begin{cases}
MD &= P(\text{DI\_Model} \times \text{RI\_Model}) \\
\text{CNSTR} &= \text{<Constraint defined in Fig. 39.\(a\)>}
\end{cases} \\
\text{Anonymous1} & \mapsto \begin{cases}
MD &= \text{PARTIAL\_FUNCTION} \times \text{DI\_Model} \rightarrow B \\
\text{CNSTR} &= \text{true}
\end{cases} \\
\text{Anonymous2} & \mapsto \begin{cases}
MD &= \text{PARTIAL\_FUNCTION} \times \text{PARTIAL\_FUNCTION} \times P(\text{DI\_Model}) \rightarrow B \\
\text{CNSTR} &= \text{true}
\end{cases}
\end{cases}
\]

\[
SAVE = \begin{cases}
\text{EMPTY\_PARTIAL\_FUNCTION} & \mapsto \begin{cases}
VSIG &= \text{PARTIAL\_FUNCTION} \\
VAL &= \{\}
\end{cases} \\
\text{DEFINED\_IN} & \mapsto \begin{cases}
VSIG &= \text{Anonymous1} \\
VAL &= \text{<Function defined in Fig. 39.\(b\)>}
\end{cases} \\
\text{DIFFER\_ONLY\_AT} & \mapsto \begin{cases}
VSIG &= \text{Anonymous2} \\
VAL &= \text{<Function defined in Fig. 40.\(c\)>}
\end{cases}
\end{cases}
\]

Figure 43: PARTIAL\_FUNCTION\_THEORY\_TEMPLATE's Resulting Specification Adornment Instance
5.4. INSTANTIATING COMMUNAL_PMap_Template

It is interesting to note that the math operations defined in the RESOLVE templates are interpreted as variables. Specification adornment environments do not contain “operations” per se, since a mathematical operation can be represented as a variable whose type is a space of functions.

5.4.2 Using the Communal_PMap_Template Abstract Template

The Communal_PMap_Template defined in Figures 36 through 38 uses the PARTIAL_FUNCTION_THEORY_TEMPLATE specification adornment template. Communal_PMap_Template is a RESOLVE parameterized concept, most naturally denoted in ACTI by an abstract template. Figure 44 shows Communal_PMap_Template as an abstract template, which takes a single abstract instance as a parameter and produces a single abstract instance as its result.

The parametric context section of Communal_PMap_Template, shown in Figure 36, explicitly describes the requirements placed on legitimate parameters to the template. Figure 45 graphically depicts the structure of any abstract instance that can be used as a parameter to Communal_PMap_Template. Each of the three distinct parameters from the template’s parametric context is explicitly represented. This figure characterizes the domain predicate (ATS.AIDP) of the abstract template denoted by Communal_PMap_Template.

When Communal_PMap_Template is applied to a parameter of the form given in Figure 45, it produces a resulting abstract instance with the structure shown in Figures 46 and 47. The CTXT component of the resulting abstract instance defines all of the external dependencies that the resulting instance will have. Before the resulting instance can be embedded in some other environment, such as BE, external entities will have to be mapped onto this context interface using an interpretation mapping.

The SAE component of the resulting abstract instance shows the specification adornment definitions it provides. These definitions include two integer-valued vari-

![Diagram](image-url)

Figure 44: Communal_PMap_Template is an Abstract Template
CHAPTER V. AN EXTENDED ACTI EXAMPLE

\[
\begin{array}{|c|c|c|}
\hline
\text{CTX} & \text{SAE} & \text{SATY} = \begin{cases} 
\text{math-integer} & \text{MD} = \mathbb{Z} \\
\text{CNSTR} & \text{true} 
\end{cases} \\
\hline
\text{SAE} & \text{SAVE} = \begin{cases} 
\text{max_total_size} & \text{VSIG} = \text{math-integer} \\
\text{VAL} & <\text{Any value} > 0 \\
\text{SAEI} & \text{max_total_size} > 0 
\end{cases} \\
\hline
\text{TE} & \begin{cases} 
\text{D}_\text{Item} & <\text{Any model, say DI-Model}> \\
\text{R}_\text{Item} & <\text{Any model, say RI-Model}> 
\end{cases} \\
\hline
\end{array}
\]

Figure 45: Communal_PMap_Template's Parameter

ables, \text{max_total_size} and \text{total_size}. \text{Max_total_size} records the value provided in the parameter to the template, while \text{total_size} is a specification variable (as opposed to a program variable) described in the \text{state variables} section of the Communal_PMap_Template concept in Figure 36. The \text{SAE} component of the resulting instance also associates the name \text{PARTIAL_FUNCTION THEORY} with the result of applying \text{PARTIAL_FUNCTION THEORY TEMPLATE} on the given parameters.

The \text{TE} component of the resulting abstract instance contains the single program type that it provides, \text{Partial_Map}. As shown in Figure 46, the type model associated with this identifier is the power set of pairs of domain and range items. Figure 47 shows the \text{OES} component of the resulting abstract instance, which contains the operation models for all of the operations the instance provides. Each operation is associated with a parameter profile, domain predicate, and effect predicate derived directly from the corresponding behavioral specification in the Communal_PMap_Template concept. Note that Initialize, Finalize, and Swap operations are also provided here, even though they are not explicitly described in the RESOLVE concept specification. In RESOLVE, these three operations must be provided for every type [9]. The behavior of the Initialize and Finalize operations are described as part of the corresponding type's declaration. Swap always has the same behavior: it exchanges the abstract values of its two operands.

To instantiate Communal_PMap_Template, we need only provide it with a parameter matching the structure of Figure 45. Unfortunately, in the basis environment \text{BE}, there is no such parameter. But the instantiation statement in Figure 31 explains exactly which entity in \text{BE} is to be substituted for each parameter to Communal_PMap_
### SAE

\[
SAE = \begin{cases} 
    SATY = \begin{cases} 
        \text{math-integer} & \rightarrow [MD = Z, \text{CNSTR} = \text{true}] \\
    \end{cases} \\
    SATE = \begin{cases} 
        \text{PARTIAL\_FUNCTION\_THEORY\_TEMPLATE} & \rightarrow <\text{Figs. 41 43}> \\
    \end{cases} 
\end{cases}
\]

### CTXT

\[
CTXT = \begin{cases} 
    TE = \begin{cases} 
        \text{D Item} & \rightarrow \text{DI-Model} \\
        \text{R Item} & \rightarrow \text{RI-Model} \\
    \end{cases} \\
    CIE = \begin{cases} 
        \text{Std Bool Fac} & \rightarrow TE = \begin{cases} 
            \text{prog-Boolean} & \rightarrow [MD = B, \text{CNSTR} = \text{true}] \\
        \end{cases} \\
        \text{Std Int Fac} & \rightarrow TE = \begin{cases} 
            \text{prog-Integer} & \rightarrow [MD = Z, \text{CNSTR} = \text{true}] \\
        \end{cases} \\
    \end{cases} 
\end{cases}
\]

### SAVE

\[
SAVE = \begin{cases} 
    \text{max total size} & \rightarrow [VSIG = \text{math-integer}, \text{VAL} = \text{Value from parameter}] \\
    \text{total size} & \rightarrow [VSIG = \text{math-integer}, \text{VAL} = 0] \\
    \end{cases}
\]

### SAEI

\[
SAEI = 0 \leq \text{total size} \leq \text{max total size}
\]

### SAIE

\[
SAIE = \begin{cases} 
    \text{FUNCTION\_THEORY} & \rightarrow \text{PARTIAL\_FUNCTION\_THEORY\_TEMPLATE(DI-Model, RI-Model)} \\
    \end{cases}
\]

### TE

\[
TE = \begin{cases} 
    \text{Partial Map} & \rightarrow [MD = \mathcal{P}(\text{DI-Model} \times \text{RI-Model}), \text{CNSTR} = \langle \text{Constraint defined in Fig. 39.(a)} \rangle] \\
    \end{cases}
\]

Figure 46: Communal_PMap_Template’s Resulting Abstract Instance
\[
\begin{align*}
\text{Define} & \quad \mapsto \quad [PP = \langle \text{Partial Map, D Item, R Item} \rangle] \\
\text{Undefine} & \quad \mapsto \quad [PP = \langle \text{Partial Map, D Item, D Item, R Item} \rangle] \\
\text{Undefine \_Any\_One} & \quad \mapsto \quad [PP = \langle \text{Partial Map, D Item, R Item} \rangle] \\
\text{Is\_Defined} & \quad \mapsto \quad [DP = \text{true}] \\
\text{Size} & \quad \mapsto \quad [PP = \langle \text{Partial Map, prog\_Integer} \rangle] \\
\text{Initialize} & \quad \mapsto \quad [PP = \langle \text{Partial Map} \rangle] \\
\text{Finalize} & \quad \mapsto \quad \ldots \\
\text{Swap} & \quad \mapsto \quad \ldots 
\end{align*}
\]

Figure 47: Communal\_PMap\_Template’s Resulting Abstract Instance (continued)
Template. What that parameter substitution information describes is a way to interpret (a subset of) BE as a valid parameter to Communal_PMap_Template. This information can be rephrased as an interpretation mapping.

\[
\begin{align*}
CPMT-Parm-IM & \equiv \\
DP &= \{ (AI_s, AI_G) : AI_s = BE \text{ and } AI_G = CPMap-Params \} \\
TYmap &= \{ \\
  D_{Item} &\mapsto \begin{cases} 
  N = \text{prog-Integer} \\
  R = \{(d_1, d_2) : d_1 = d_2\} \\
  C = \text{true} 
\end{cases} \\
  R_{Item} &\mapsto \begin{cases} 
  N = \text{prog-Integer} \\
  R = \{(r_1, r_2) : r_1 = r_2\} \\
  C = \text{true} 
\end{cases} \\
MCORR &= \text{SAVariableModel(size_limit, } AI_s).VAL = \text{SAVariableModel(max_total_size, } AI_G).VAL \\
MCONV &= 0 \leq \text{SAVariableModel(size_limit, } AI_s).VAL
\end{align*}
\]

Figure 48: An Interpretation Mapping for Communal_PMap_Template’s Parameter

Figure 48 shows the interpretation mapping implicitly described by the arguments to Communal_PMap_Template in Figure 31. This interpretation mapping explains specifically how to interpret BE.E as having the form shown in Figure 45. Let CPMT-Params represent an abstract instance exactly like Figure 45, but with its DI-Model and RI-Model both \( (Z, \text{true}) \).

The interpretation mapping CPMT-Parm-IM has a domain predicate (DP) that is only true for BE and CPMap-Params. In Figure 48, conventional set notation is used to describe the domain predicate, with the understanding that the predicate is true for all argument tuples in the set, and false for others. The TE component of the mapping describes the type-to-type interpretations, including a correspondence relation (R) and convention (C) for each one. In this case, prog-Integer is mapped to both D_Item and R_Item, using an identity correspondence. The MCORR predicate describes the relationship between the variable values in the two abstract instances, and the MCONV represents an invariant that must be true of the subset of BE involved in the mapping.
CHAPTER V. AN EXTENDED ACTI EXAMPLE

Using the ↓ operator described in Section 4.13.3, it is now possible to reshape BE so that it is in an appropriate form, and then apply the template:

\[
CPMap-Instance \equiv \text{Communal\_PMap\_Template(abstract(BE \downarrow \text{CPMT-Parm-IM CPMT-Params}))} \tag{5.1}
\]

The result is \textit{CPMap-Instance}, an abstract instance of the form shown in Figures 46 and 47.

5.5 Instantiating Unordered\_Stack

The second step in the \texttt{Int\_Multiset} instantiation is applying the concrete template denoted by \texttt{Unordered\_Stack}, a RESOLVE realization of the \texttt{Communal\_PMap\_Template}. This concrete template takes a single concrete instance as a parameter and produces a single concrete instance as its result.

The \texttt{Unordered\_Stack} realization presented here is derived directly from the unbounded version described by Bucci et al. [2, pp. 42-44]. Figure 49 contains the realization header for \texttt{Unordered\_Stack}. Every RESOLVE realization has a realization header, which contains all of the information a client programmer needs in order to use the realization. This includes describing the \textbf{parametric context} of the realization, which captures the essence of the single parameter to the concrete instance.

Figure 50 graphically depicts the structure of any concrete instance that can be used as a parameter to \texttt{Unordered\_Stack}. Note that in RESOLVE, realizations automatically include the parameters to their corresponding concepts, so all of the \texttt{Communal\_PMap\_Template} parameters have been explicitly duplicated in Figure 50. In addition to the three parameters from \texttt{Communal\_PMap\_Template}'s parametric context, Figure 50 includes the three additional parameters from the realization header: an \texttt{Are\_Equal} operation, a \texttt{D\_R\_Pair\_Facility}, and a \texttt{Stack\_Facility}.

The \texttt{Are\_Equal} operation is shown in the operation environment of the concrete instance in Figure 50. The operation meaning associated with \texttt{Are\_Equal} includes the operation model defined by the specification in the realization header, along with the procedure relation and status function computed by the actual operation substituted for this parameter.

\texttt{D\_R\_Pair\_Facility} is shown as a context-bound concrete instance in the \texttt{CIE} component of the parameter concrete instance. This context-bound concrete instance is shown in Figure 51. The concrete instance component (\texttt{CI}) of \texttt{D\_R\_Pair\_Facility} provides a type \texttt{Record2}, and the appropriate operations for that type, as defined by the RESOLVE concept \texttt{Record2\_Template} used to specify this parameter in the
5.5. **INSTANTIATING UNORDERED_STACK**

realization header Unordered\_Stack for Communal\_PMap\_Template

context

global context
concept Record2\_Template
concept Stack\_Template

parametric context

facility D\_R\_Pair\_Facility is
Record2\_Template (D\_Item, R\_Item)

facility Stack\_Facility is
Stack\_Template (Record2)

operation Are\_Equal (
    preserves d1 : D\_Item
    preserves d2 : D\_Item
) : Boolean
    ensures Are\_Equal iff d1 = d2

end Unordered\_Stack

Figure 49: The Unordered\_Stack Realization Header

realization header. Further, this facility depends on two types defined outside of it: Item1 and Item2, the types of the two fields in each Record2 object. The interpretation mapping component (IM) of D\_R\_Pair\_Facility describes how the context of its CI component is fulfilled by the remainder of the parameter concrete instance shown in Figure 50. As shown in Figure 51, the type D\_Item from the parameter concrete instance is bound to the Item1 type in D\_R\_Pair\_Facility’s context, and R\_Item is bound to Item2. Thus, an object of the Record2 type provided by D\_R\_Pair\_Facility contains a D\_Item and R\_Item pair.

Figure 52 shows the Stack\_Facility context-bound concrete instance. The concrete instance component (CI) of Stack\_Facility provides the types and operations described by the corresponding RESOLVE concept used to specify this parameter in
\[ CTXT = \begin{array}{|c|c|c|}
\hline
SAE & SATY = \begin{cases} 
\text{math-integer} \rightarrow \text{MD} = \mathbb{Z} \\
\text{CNSTR} = \text{true} 
\end{cases} \\
\hline
\text{SAVE} & \begin{cases} 
\text{max_total_size} \rightarrow \text{VSIG} \rightarrow \text{VAL} = \text{<Any value > 0> } \\
\text{EI} = \text{max_total_size > 0} 
\end{cases} \\
\hline
TE & \begin{cases} 
\text{D_Item} \mapsto \text{<Any model, say DI-Model>} \\
\text{R_Item} \mapsto \text{<Any model, say RI-Model>} \\
\text{OMOD} = \begin{cases} 
\text{PP} = \langle \text{D_Item, D_Item, prog-Boolean} \rangle \\
\text{DP} = \text{true} 
\end{cases} \\
\text{OR} = \langle \text{<See postcondition in Fig. 49> \rangle} \\
\text{PR} = \ldots \\
\text{SF} = \ldots 
\end{cases} \\
\hline
OE & \begin{cases} 
\text{Are_Equal} \mapsto \text{EP} = \langle \text{<See postcondition in Fig. 49> \rangle} \\
\text{D_R_Pair_Facility} \mapsto \text{<See Fig. 51> \rangle} \\
\text{Stack_Facility} \mapsto \text{<See Fig. 52> \rangle} 
\end{cases} \\
\hline
\end{array} \]

Figure 50: Unordered Stack’s Parameter

the realization header. The interpretation mapping component binds the context of the CI component, just as before.

Figures 53 through 57 give the complete RESOLVE code for the Unordered Stack realization body. The main difference between this realization and the one presented by Bucci et al. is that here, an internal global state variable communal_length (Figure 55) is used to track the cumulative size of all Partial_Map objects combined. The code for the various operations provided by this realization show how this state variable is updated.

In addition to having an internal state variable, the Unordered Stack realization also includes two internal operations, Pop_Until_Passed and Combine. These operations are used locally within the realization, but in RESOLVE they are not exported outside the realization. Also, note that here no implementations for Initialize, Finalize, or Swap on Partial_Map objects are provided. This is because a RESOLVE compiler could automatically use the corresponding operations on the Stack type being used as the Partial_Map representation, and in this case the appropriate behavior would be exhibited. If this did not provide the appropriate behavior, the
5.5. INSTANTIATING UNORDERED STACK

\[
\begin{align*}
CTXT &= \{ \text{Item1} \mapsto \text{DI-Model}, \text{Item2} \mapsto \text{RI-Model} \} \\
TE &= \{ \text{Record2} \mapsto \text{Item1} \times \text{Item2}, \\
&\quad \text{Initialize} \mapsto \ldots, \\
&\quad \text{Finalize} \mapsto \ldots \\
OE &= \{ \text{Swap} \mapsto \ldots, \\
&\quad \text{Swap} \_ \text{Field1} \mapsto \ldots, \\
&\quad \text{Swap} \_ \text{Field2} \mapsto \ldots \\
CI &= \text{D.R.} \_ \text{Pair} \_ \text{Facility} \mapsto \text{IM} \\
DP &= \text{<See text description>} \\
TY\text{map} &= \begin{cases} \\
N &\mapsto \text{D} \_ \text{Item} \\
N &\mapsto \text{R} \_ \text{Item} \\
&\quad \text{Item1} \mapsto \text{R} = \{(d_1,d_2) : d_1 = d_2 \} \\
&\quad \text{Item2} \mapsto \text{R} = \{(r_1,r_2) : r_1 = r_2 \} \\
&\quad \text{C} = \text{true} \\
\end{cases}
\end{align*}
\]

Figure 51: The D.R.Pair.Facility Context-Bound Concrete Instance

\[
\begin{align*}
CTXT &= \{ \text{Item} \mapsto \text{DI-Model} \times \text{RI-Model} \} \\
TE &= \{ \text{Stack} \mapsto \text{Item}^* \\
&\quad \text{Initialize} \mapsto \ldots, \\
&\quad \text{Finalize} \mapsto \ldots \\
OE &= \{ \text{Swap} \mapsto \ldots, \\
&\quad \text{Push} \mapsto \ldots, \\
&\quad \text{Pop} \mapsto \ldots \\
&\quad \text{Size} \mapsto \ldots \\
CI &= \text{Stack} \_ \text{Facility} \mapsto \text{IM} \\
DP &= \text{<See text description>} \\
TY\text{map} &= \begin{cases} \\
N &\mapsto \text{Record2} \\
&\quad \text{Item} \mapsto \text{R} = \{(dr_1,dr_2) : dr_1 = dr_2 \} \\
&\quad \text{C} = \text{true} \\
\end{cases}
\end{align*}
\]

Figure 52: The Stack.Facility Context-Bound Concrete Instance
realization body
Unordered Stack for Communal_PMap_Template

calendar context

  global context
mathematics STRING_OCCURS_COUNT_MACHINERY
mathematics STRING.Reverse.MACHINERY
mathematics STRING_ELEMENTS_MACHINERY

  local context
renaming type Record2 as D_R_Pair
  with subcomponent field1 as domain_value
  with subcomponent field2 as range_value

math facility STRING_OCCURS_COUNT_FACILITY is
  STRING_OCCURS_COUNT_MACHINERY (math[D_R_Pair])

math facility STRING.Reverse.FACILITY is
  STRING.Reverse.MACHINERY (math[D_R_Pair])

math facility STRING_ELEMENTS.FACILITY is
  STRING_ELEMENTS_MACHINERY (math[D_R_Pair])

math operation CONTAINS (  
  s : string of math[D_R_Pair]  
  d : math[D_Item]  
) : boolean
explicit definition
  there exists r : math[R_Item]  
    (OCCURS_COUNT (s, (d,r)) > 0)

math operation CONTAINS_AS_FIRST_ENTRY (  
  s : string of math[D_R_Pair]  
  d : math[D_Item]  
) : boolean
explicit definition
  there exists r: math[R_Item],  
    t : string of math[D_R_Pair]  
    (s = <(d,r)> * t)

Figure 53: The Unordered Stack Realization Body
operation Pop_Until_Passed (  
  alters s1 : Stack  
  alters s2 : Stack  
  preserves d : D_Item  
)
  requires s2 = empty_string  
  ensures ELEMENTS (s1 * s2) = ELEMENTS (#s1) and  
    (if s2 = empty_string  
    then not CONTAINS (s1, d)  
    else CONTAINS_AS_FIRST_ENTRY (s2, d))  
context  
  variables  
    found : Boolean  
    p : D_R_Pair  
begin  
  loop maintaining  
    not CONTAINS (s2, d) and  
    REVERSE (s2) * s1 = REVERSE (#s2) * #s1  
    if Length (s1) = 0 then  
      s1 := s2  
      exit  
    end if  
    Pop (s1, p)  
    found := Are_Equal (p.domain_value, d)  
    Push (s2, p)  
    if found then  
      exit  
    end if  
  end loop  
end Pop_Until_Passed  
operation Combine (  
  alters s1 : Stack  
  consumes s2 : Stack  
)
  ensures s1 = REVERSE (#s2) * #s1  

Figure 54: The Unordered_Stack Realization Body (continued)
context
  variables
    p : D.R_Pair
begin
  loop maintaining
    REVERSE (s2) * s1 = REVERSE (#s2) * #s1
  while Length (s2) > 0  do
    Pop (s2, p)
    Push (s1, p)
  end loop
end Combine

state variables
  communal_length : Integer
convention
  0 ≤ communal_length ≤ max_total_size
convention
  total_size = communal_length

interface

type Partial_Map is represented by Stack
convention
  for all d : math[D_Item], r : math[R_Item]
    (OCCURS_COUNT (m.rep, (d,r)) ≤ 1)
convention
  m = ELEMENTS (m.rep)

operation Define (  
  alters m : Partial_Map  
  consumes d : D_Item  
  consumes r : R_Item  
)
begin
  Push (m.rep, (d, r))
  communal_length := communal_length + 1
end Define

Figure 55: The Unordered Stack Realization Body (continued)
5.5. *INSTANTIATING UNORDERED_STACK*

```plaintext
operation Undefine (  
alters m : Partial_Map  
preserves d : D_Item  
produces d_copy : D_Item  
produces r : R_Item  
)
context  
variables  
catalyst : Stack
begin  
Pop_Until_Passed (m.rep, catalyst, d)  
Pop (catalyst, (d_copy, r))  
Combine (m.rep, catalyst)  
communal_length := communal_length - 1
end Undefine

operation Undefine_Any_One (  
alters m : Partial_Map  
produces d : D_Item  
produces r : R_Item  
)
begin  
Pop (m.rep, (d, r))  
communal_length := communal_length - 1
end Undefine_Any_One

operation IsDefined (  
preserves m : Partial_Map  
preserves d : D_Item  
): Boolean  
context  
variables  
catalyst : Stack  
defined : Boolean
begin  
Pop_Until_Passed (m.rep, catalyst, d)
```

Figure 56: The Unordered_Stack Realization Body (continued)
if Length (catalyst) > 0 then
    Combine (m.rep, catalyst)
    defined := true
end if
return defined
end Is_Defined

operation Size (preserves m : Partial_Map) : Integer
begin
    return Length (m.rep)
end Size

end Unordered_Stack

Figure 57: The Unordered_Stack Realization Body (continued)

A programmer writing this realization would be obliged to provide an implementation herself.

Just as the realization header characterizes the acceptable parameters to the Unordered_Stack concrete template, the realization body characterizes the resulting concrete instances. Figure 58 presents an outline of the structure of a resulting concrete instance. Many of the details are elided here for simplicity.

As with the result of Communal_PMap_Template, the CTXT component describes all of the external dependencies of the resulting concrete instance. These dependencies can be taken almost directly from the global context sections of the realization body and the concept, and from the parameters to Unordered_Stack. The concept’s global context must be included here because in RESOLVE, the realization implicitly imports everything the corresponding concept does. Thus, the CTXT for Unordered_Stack’s resulting instance will depend on D_Item, R_Item, max_total_size, Standard_Bool-ean_Facility, Standard_Integer_Facility, D_R_Pair_Facility, Stack_Facility, and assorted parameterized mathematics modules. Similarly, the SAE component of the resulting concrete instance will contain all of the mathematics definitions introduced in the local context section of the Unordered_Stack realization body.

The type, variable, and operation environments in the resulting concrete instance are where the differences from Communal_PMap_Template are most noticeable. In Figure 58 we see that the type Partial_Map in the realization is modeled as a string of
5.5. **INSTANTIATING UNORDERED_STACK**

\[
\begin{align*}
CTXT &= 
\begin{cases}
\text{\rule{1cm}{1cm}}
\end{cases} \\
SAE &= 
\begin{cases}
\text{\rule{1cm}{1cm}}
\end{cases} \\
TE &= \left\{ \begin{array}{c}
\text{Partial\_Map} \mapsto \left[ 
\begin{array}{l}
MD = \text{Item}^* \\
CNSTR = \text{true}
\end{array} \right] \\
\text{communal\_length} \mapsto \left[ 
\begin{array}{l}
VSIG = \text{prog\_Integer} \\
VAL = 0
\end{array} \right]
\end{array} \right. \\
VE &= \left\{ \begin{array}{l}
\text{Pop\_Until\_Passed} \mapsto \left[ 
\begin{array}{l}
OMOD = \text{PP = \langle Stack, Stack, D\_Item \rangle} \\
DP = \langle See\ precondition\ in\ code \rangle \\
EP = \langle See\ postcondition\ in\ code \rangle \\
PR = \ldots \\
SF = \ldots
\end{array} \right]
\end{array} \right.
\end{align*}
\]

\[
\begin{align*}
OE &= \left\{ \begin{array}{l}
\text{Combine} \mapsto \ldots \\
\text{Define} \mapsto \ldots \\
\text{Undefine} \mapsto \ldots \\
\text{Undefine\_Any\_One} \mapsto \ldots \\
\text{Is\_Defined} \mapsto \ldots \\
\text{Size} \mapsto \ldots \\
\text{Initialize} \mapsto \ldots \\
\text{Finalize} \mapsto \ldots \\
\text{Swap} \mapsto \ldots
\end{array} \right.
\end{align*}
\]

Figure 58: Unordered\_Stack's Resulting Concrete Instance

**Items** (e.g., the space “Item*”), which is the mathematical model for a stack exported by the Stack\_Facility parameter. Further, all operations in the realization have behavioral specifications and implementations in terms of this type model. We also see that communal\_length is an integer-valued program variable, and it is exposed along with all of the “internal” operations.

In RESOLVE, no implementation-level behavioral specifications (in terms of Stack objects) are given for the operations on Partial\_Map. Instead, these operations are obligated to conform to the more abstract behavioral descriptions in the corresponding concept. In ACT1, however, concrete instances and abstract instances are fully independent, meaning that the Unordered\_Stack concrete instance might be used to
US-Param-IM ≡

\[
\begin{align*}
DP & = \{(AI_S, AI_G) : AI_S = BE \text{ and } AI_G = US-Params\} \\
T_Ymap & = \\
\text{D, Item} & \mapsto R = \{(d_1,d_2) : d_1 = d_2\} \\
\text{C} & = \text{true} \\
N & = \text{prog-Integer} \\
\text{R, Item} & \mapsto R = \{(r_1,r_2) : r_1 = r_2\} \\
\text{C} & = \text{true} \\
N & = \text{Int.Int.Record} \\
\text{Record2} & \mapsto R = \{(r_1,r_2) : r_1 = r_2\} \\
\text{C} & = \text{true} \\
N & = \text{Int.Int.Stack} \\
\text{Stack} & \mapsto R = \{(s_1,s_2) : s_1 = s_2\} \\
\text{C} & = \text{true} \\
\text{Swap.Field1} & \mapsto \ldots \\
\text{Swap.Field2} & \mapsto \ldots \\
\text{Push} & \mapsto \ldots \\
\end{align*}
\]

\[
\begin{align*}
M_{CORR} & = \text{SAVariableModel(size_limit, } AI_S).VAL = \text{SAVariableModel(max.total.size, } AI_G).VAL \\
M_{CONV} & = 0 \leq \text{SAVariableModel(size_limit, } AI_S).VAL
\end{align*}
\]

Figure 59: An Interpretation Mapping for Unordered Stack’s Parameter

supply behavior conforming with any number of different abstract instances. As a result, we cannot depend on any one abstract instance as the target when defining the type and operation models used in the concrete instance. We know that, in theory, there is a minimal precondition and maximal postcondition for the code of each operation in the Unordered Stack realization, expressible in terms of the Stack objects being manipulated. The RESOLVE verification rules do not require that these be made explicit, but nevertheless they are available for use in the denotation of Unordered Stack.

Also, because abstract and concrete instances are independent, there is no notion of “local” or “internal” resources in an ACTI concrete instance. An operation like Combine, or a variable like communal_length, is only “internal” because it is not exported. The question of what is exported can only be answered relative to an
interface. Thus, the resulting concrete instance produced by an \texttt{Unordered\_Stack}
instantiation in ACTI makes all of its variables and operations available. It is only
when that concrete instance is combined with a more restrictive abstract instance
that the notion of “internal” or “internally hidden” comes into play.

Figure 59 shows the interpretation mapping implicitly described by the arguments
to \texttt{Unordered\_Stack} in Figure 31. This interpretation mapping explains specifically
how to interpret \texttt{BE} as having the form shown in Figure 50. Let \texttt{US-Params}
represent a concrete instance exactly like Figure 50, but with its \texttt{DI-Model} and \texttt{RI-Model}
both \( \langle Z, \text{true} \rangle \).

The interpretation mapping \texttt{US-Parm-IM} has a domain predicate (\texttt{DP}) that is
only true for \texttt{BE} and \texttt{US-Params}. The remainder of the interpretation mapping is
exactly analogous to the one used in instantiating \texttt{Communal\_PMap\_Template}, shown
in Figure 48. Using the \( \downarrow \) operator, one can now reshape \texttt{BE} so that it is in an
appropriate form, and then apply the template:

\[
\text{UStack-Instance} \equiv \\
\text{Unordered\_Stack(} \\
\quad \text{BE} \downarrow \text{US-Parm-IM} \quad \text{US-Params})
\]

The result is \texttt{UStack-Instance}, a concrete instance of the form outlined in Figure 58.

5.6 \textbf{Instantiating UStack\_To\_CMap}

The third step in the \texttt{Int\_Multiset} instantiation is applying the implementation
mapping template \texttt{UStack\_To\_CMap}. In the current version of RESOLVE, no separate
construct exists to represent such a parameterized mapping, or give it a name\(^2\). However, RESOLVE does require programmers to provide all of the information necessary
to construct such a interpretation, and it is all recorded in the realization body.

RESOLVE has explicit syntactic slots for recording type-specific correspondence
and convention assertions, as well as subsystem-level correspondence and convention
assertions. RESOLVE also implicitly restricts type and operation interpretation mappings by requiring the programmer to use the same program identifier for a type or
operation in both a concept and a realization when the two are connected. For the
\texttt{Unordered\_Stack} realization body in Figures 53 through 57, those bits and pieces can be pulled out and repackaged into a RESOLVE-like description of an interpretation
mapping template. The result is shown in Figures 60 and 61.

\(^2\)One has been proposed as a result of this research; Figures 60 and 61 show an example.
interpretation UStack_To_CPMap

custom
  global context
  concept Communal_PMap_Template
  realization Unordered_Stack
  mathematics STRING_OCCURS_COUNT_MACHINERY
  mathematics STRING_ELEMENTS_MACHINERY

parametric context

type D_Item
  type R_Item
  constant max_total_size : integer
    restriction
      max_total_size > 0

local context
  math facility STRING_OCCURS_COUNT_FACILITY is
    STRING_OCCURS_COUNT_MACHINERY (math[D_Item], math[R_Item])
  math facility STRING_ELEMENTS_FACILITY is
    STRING_ELEMENTS_MACHINERY (math[D_Item], math[R_Item])

mapping
  from R is Unordered_Stack (D_Item, R_Item, max_total_size, *, *, *)
  to C is Communal_PMap_Template (D_Item, R_Item, max_total_size)
  concept convention
    0 ≤ communal_length ≤ max_total_size
  concept correspondence
    C.total_size = R.communal_length

Figure 60: The UStack_To_CPMap Interpretation Mapping
5.6. INSTANTIATING USTACK_TO_CPMAP

\[\text{types}\]
\[\text{R.Partial\_Map to C.Partial\_Map}\]
\[\text{exemplar from m\_rep to m}\]
\[\text{convention}\]
\[\text{for all d : math[D\_Item], r : math[R\_Item]}\]
\[\text{(OCCURS\_COUNT (m\_rep, (d,r)) \leq 1)}\]
\[\text{correspondence}\]
\[\text{m = ELEMENTS(m\_rep)}\]

\[\text{operations}\]
\[\text{R.\_Define to C.\_Define}\]
\[\text{R.Undefine to C.Undefine}\]
\[\text{R.Undefine\_Any\_One to C.Undefine\_Any\_One}\]
\[\text{R.Is\_Defined to C.Is\_Defined}\]
\[\text{R.Size to C.Size}\]
\[\text{R.Initialize to C.Initialize}\]
\[\text{R.Finalize to C.Finalize}\]
\[\text{R.Swap to C.Swap}\]

\[\text{end UStack\_To\_CPMap}\]

Figure 61: The UStack\_To\_CPMap Interpretation Mapping (continued)

UStack\_To\_CPMap is a parameterized interpretation mapping. It has the same parametric context as Communal\_PMap\_Template, and for any RESOLVE instantiation of Communal\_PMap\_Template realized by Unordered\_Stack, UStack\_To\_CPMap will generate the corresponding interpretation mapping. In the mapping section of UStack\_To\_CPMap, the from and to clauses give local names to the corresponding instances of Communal\_PMap\_Template and Unordered\_Stack for use in defining the mapping. Here they are called \textit{R} (for realization) and \textit{C} (for concept), respectively. Immediately below the from and to clauses, the correspondence and convention for the subsystem-level state can be specified. These syntactic slots are the syntactic analog of the \textit{MCORR} and \textit{MCORD} components of an ACTI interpretation mapping. UStack\_To\_CPMap then describes the type interpretation mapping and the operation interpretation mapping, both of which naturally follow from the Unordered\_Stack realization body.
Once this information has been presented separately from the realization body itself, it should be clear how UStack_To_CMap can be interpreted in ACTI. It is an interpretation mapping template, which takes a single abstract instance as a parameter and produces a single interpretation mapping as a result. The parametric context of UStack_To_CMap is the same as that for Communal_PMap_Template, so the structure of its parameter is characterized by Figure 45. The resulting interpretation mapping is shown in Figure 62.

\[
\begin{align*}
DP &= \{(AI_s, AI_G) : AI_s = R \text{ and } AI_G = C\} \\
TYmap &= \{ \begin{align*}
N &= \text{R.Partial}\_\text{Map} \\
R &= \{ (m, m._\text{rep}) : m = \text{ELEMENTS}(m._\text{rep}) \} \\
C &= \{ m._\text{rep} : \forall d \in \text{DI-Model}, r \in \text{RI-Model} \\
&\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \ quad
5.7 Combining the Instances and Binding Context

Now that Communal_PMap_Template, Unordered_Stack, and UStack_To_CPM have all been instantiated, they can be combined to form a new concrete instance:

\[
\text{Int\_Multiset\_Instance} \equiv \text{abstract}(\text{UStack\_Instance} \downarrow \text{US2CPM\_Instance} \text{CPMap\_Instance})
\]

(5.4)

This new instance has the implementation-level procedure relations and status functions defined in the Unordered_Stack realization, but the abstract behavioral descriptions it contains are those from Communal_PMap_Template. Further, the \( \downarrow \) operator has removed all of the features of the concrete instance produced by the realization that were not described in the abstract instance produced by the concept. Thus, Combine, Pop_Until_Passed, and communal_length are not present in \text{Int\_Multiset\_Instance}. This is the way RESOLVE's hidden, internal resources can be modeled in ACTI.

Now all that remains is to convert \text{Int\_Multiset\_Instance} into a context-bound concrete instance so that it can be embedded in \text{BE.CIE}, as described in Section 4.13.6. This is done by providing an interpretation mapping that explains how \text{BE.E} can be interpreted as \text{Int\_Multiset\_Instance.CTXT}. Together, the interpretation mapping and \text{Int\_Multiset\_Instance} form a context-bound concrete instance that can be embedded as a new entry in \text{BE.CIE}. This produces the basis environment \text{BE}', which is the final program state achieved after the execution of the entire \text{Int\_Multiset} instantiation.

5.8 Chapter Summary

In this chapter, an ACTI interpretation of the semantics of a RESOLVE instantiation was provided. This example helps to show some of the more advanced features of ACTI, including the use of interpretation mappings, the \( \downarrow \) operator, and the \text{abstract} operator. Further, this example highlights the fact that in ACTI, concrete instances (implementations) are completely independent from abstract instances (specifications), and the relationships between instances are independent from the instances involved. This means that in addition to allowing multiple realizations for one concept, ACTI allows multiple concepts for the same realization, and even multiple interpretation mappings for a single concept-realization pair.

Also, it is clear that the key ideas in ACTI are only the core primitives needed to understand the semantics of subsystems. In a real programming language, the statements provided for writing software may have semantic denotations that are
complex compositions of a series of more primitive steps describable in the terms presented in Chapter IV.