Specware 4.1 Tutorial

Specware 4.1 Tutorial

Table of Contents

Chapter 1. Specware Concepts

1.1. Overview

Specware is designed with the idea that large and complex problems can be specified by combining small and simple specifications, and that problem specifications can be refined into a working system by the controlled stepwise introduction of implementation design decisions, in such a way that the refined specifications and ultimately the working code is a provably correct refinement of the original problem specification.

Specware allows you to express requirements as formal specifications without regard to the ultimate implementation or target language. Specifications can describe the desired functionality of a program independently of such implementation concerns as architecture, algorithms, data structures, and efficiency. This makes it possible to focus on the correctness, which is crucial to the reliability of large software systems.

Often, there are many possible solutions to a single problem, all valid, with different advantages and drawbacks. It is sometimes very difficult to make a decision given all the different trade-offs. Using Specware, the analysis of the problem can be kept separate from the implementation process, and implementation choices can be introduced piecemeal, making it easier to backtrack or explore alternatives.

Specware allows you to articulate software requirements, make implementation choices, and generate provably correct code in a formally verifiable manner. The progression of specifications forms a record of the system design and development that is invaluable for system maintenance. You can later adapt the specifications to new or changed requirements or make different implementation decisions at any level of the development while reusing what has not changed, and generate provably correct new code by the same process.

1.2. Specware Development Process

1.2.1. Building Specifications

The first step in building an application in Specware is to describe the problem domain in abstract form. You use the Metaslang language to build specifications (specs) that describe the abstract concepts of the problem domain. Specs contain types, which denote collections of values, operations, which denote functions on values, and axioms, which describe the required properties of the operations in logical formulas.

To design specifications, you may combine top-down and bottom-up approaches. To begin, you break down the problem domain into small, manageable parts that you can understand more easily in complete detail. You create specifications for each part. This conceptual decomposition process allows you to isolate and describe each detail of functionality.

You then extend and compose the smaller specifications to form a larger, more complex specification.

You create morphisms to describe how specifications are related. A morphism is a formal mapping between two specifications that describes exactly how one is translated or extended into the other. Morphisms can describe “part-of” as well as “is-a” relationships.

You can extend a specification by importing it and adding new concepts. When you extend a specification, you can also translate the names of the concepts to a different terminology.

As you extend the specification, you provide more and more information about the structure and details of the types and operations, by including axioms and theorems to describe them.

You compose simple specifications to form more complex specifications using substitutions (e.g., to instantiate parameterized specifications) and colimit. Colimit “glues” specs together along shared concepts. Morphisms describe exactly how concepts are shared.

1.2.2. Refining Specifications

When you have described the abstract concepts in your problem domain, you begin the refinement process. Refinement maps a problem specification set into a solution specification set. Refinements replace functionality constraints with algorithms and abstract types with implementation data structures. You refine a specification by mapping it into the implementation domain. You describe how the concepts of the problem domain are implemented in terms of computational concepts. Computational concepts, such as numbers (e.g., integers) and collections (e.g., lists) are already specified in the Specware library. In the process of refinement, you build a bridge from your abstract specifications to these concrete, implementation-specific specifications.

Morphisms are also used to describe how each specification is to be mapped into a desired target specification. The source spec is a more abstract description of concepts, while the target spec describes those concepts more concretely. Often, the target spec is obtained by extending some other spec that contains concepts in term of which the concepts of the source spec are expressed. For example, if you have an abstract spec for a house, you can refine it by importing a spec for materials (wood, etc.) and expressing how the house is built in terms of those materials.

A morphism maps each type or operation of the source spec to a type or operation of the target spec. In certain cases, it may be useful to map a type or operation to a new type or operation that is not present in the target spec but that is defined in terms of those in the target spec. Interpretations are used for this purpose.

An interpretation contains three specs: a source spec (the domain), a target spec (the codomain), and a mediator. The mediator spec extends the target, so there is an inclusion morphism from the target to the mediator. A morphism from the source spec to the mediator expresses how the source spec maps to the extension of the target spec.

A morphisms can be viewed as a particular kind of interpretation where the mediator coincides with the target. Indeed, the refinement relation between a more abstract spec and a more concrete one can be expressed by a morphism but also by an interpretation.

1.2.3. Extending Refinements

You build up a refinement in the same way you build up a specification, in an iterative, stepwise progression. You build basic morphisms (or interpretations) from more abstract specs to more concrete specs, then compose the morphisms to extend the refinement. There are two ways to compose morphisms.

Sequential composition (`A=>B + B=>C = A=>C`) deepens a refinement. For example, if you are specifying a house, you might use sequential composition of morphisms to create an increasingly detailed description of a single room.

Parallel composition (`A=>A' + B=>B' = (A,B) => (A',B')`) widens, or increases the complexity of a refinement. For example, in the specification of a house, when you have several morphisms that refine different rooms, you could combine them in parallel to create a description of a multi-room house. You could do this before or after providing the additional detail for each one. You use the colimit method for this kind of composition, as you do to compose specifications.

Specware contains a library of specs that fully describe various mathematical and computational concepts that you need to produce an implementation of an abstract specification. The goal of refinement is to refine your abstract specs in terms of these library specs. This process guarantees that Specware can generate code to implement your specifications in its target language, Lisp.

1.2.4. Generating Code

Once the abstract spec is refined to a “constructive” spec, Specware can generate code in the target language. The generated code contains a function for each of the operations you have described in the abstract specification. After you examine and test it, you can embed it in an application that uses the functions.

1.3. Specification Components

A specification (spec) consists of some types, some operations (ops), and some axioms about the types and ops.

1.3.1. Types, Operations, Axioms

A type is a syntactic entity that denotes a set of values. In its simplest form, a type is a symbol. For example, a spec can contain a type `Nat` that denotes the set of natural numbers, i.e., 0, 1, 2, ...

Type symbols can be combined by means of some pre-defined constructs to build more complex types. One such construct is “`->`”: if `A` and `B` are types, `A -> B` is also a type. The set denoted by `A -> B` is the set of all total functions from the set denoted by `A` to the set denoted by `B`.

An op is a syntactic symbol accompanied by a type. An op denotes an element of the set denoted by its type. For example, a spec can contain an op `zero` of type `Nat`, which denotes the natural number 0. It can also contain an op `succ` of type `Nat -> Nat`, which denotes the function that returns the successor of any given natural number.

From the type `Nat` and the ops `zero` and `succ` alone, it does not follow that they denote the natural numbers, 0, and successor function. They could denote the set of the days of the week, Wednesday, and the identity function, respectively. The intended meaning can be enforced by means of axioms.

An axiom is a term of type `Boolean`. `Boolean` is a type automatically present (built-in) in every spec, which denotes the set of boolean truth values (“true” and “false”). Terms are built from typed variables (i.e., symbols accompanied by types), ops, and some pre-defined constructs. These include universal and existential quantifiers (“for all” and “exists”), logical connectives (“and”, “or”, “implies”, “iff”, “not”), and equality.

Here is an example of an axiom to constrain successor to never return 0:

```
fa(x) ~(succ x = zero)
```

In the axiom, “`=`” denotes equality, “`~`” boolean negation, and “`fa`” universal quantification. Note that this axiom rules out the possibility that `succ` is the identity function. Additional axioms can be added to constrain the spec to capture exactly the natural numbers (essentially, the rest of Peano’s axioms).

1.3.2. Models

In the above description, the notion of a type or op “denoting” a set or a function corresponds to the notion of model of a spec. A model of a spec is an assignment of a set to each type and of an element to each op from the set assigned to the type of the op, such that all the axioms of the spec are satisfied.

In the example spec sketched above, a model consists of a set *`N`* assigned to `Nat`, an element *`z`* in *`N`* assigned to `zero`, and a function *`s`* from *`N`* to *`N`* (i.e., an element *`s`* of the set denoted by `Nat -> Nat`) assigned to `succ`. In the absence of axioms, the model where *`N`* consists of the days of the week, *`z`* Wednesday, and *`s`* identity, is a valid model. But with the axiom shown above, since *`s(z) = z`*, this cannot be a model. With the rest of Peano’s axioms, *`N`*, *`z`*, and *`s`* are constrained to be isomorphic to natural numbers, 0, and successor. (No matter how many axioms are added to the spec, it is not possible to pin down *`N`* to be exactly the set of natural numbers. Things can be pinned down only up to isomorphism. But this is fine because isomorphic sets are totally equivalent.)

A spec may have no models. This happens when the spec contains incompatible axioms. This situation is often subtle and difficult to detect, and it is always a symptom of human errors in the specification process. Whether a spec has models or not is an undecidable problem. However, by following certain practices and disciplines in the development of specs, this situation can be avoided.

1.3.3. Polymorphism

Types can be polymorphic. In its simplest form, a polymorphic type is a symbol plus one or more “parameter types”. While a monomorphic (i.e., non-polymorphic) type denotes a set, a polymorphic type denotes a function that returns a set given sets for all its parameters. For example, a spec can contain a type `List a`, where `a` is the type parameter, which denotes the set of (finite) lists over `a`: more precisely, it denotes a function that, given a set *S* for `a`, returns the set of all lists of elements of *S*. If *S* is the set of natural numbers, it returns the set of all lists of natural numbers; if *S* is the set of the days of the week, it returns the set of all lists of days of the week.

A polymorphic type can be instantiated by replacing its parameters with other types. The latter can be polymorphic or monomorphic: if at least one is polymorphic, the instantiated type is polymorphic; otherwise, it is monomorphic. For example, `List a` can be instantiated to the monomorphic `List Nat` or to the polymorphic `List(List a)`.

Correspondingly, ops can be polymorphic. An op is polymorphic when its type is a polymorphic type. While a monomorphic op denotes an element of the set denoted by the type of the op, a polymorphic op denotes a function that, given a set for each parameter type of the polymorphic type of the op, returns an element of the set obtained by applying to such parameter sets the function denoted by the type of the op. For example, a spec can contain an op `nil` of type `List a`, that denotes the empty list, for each set assigned to parameter `a`.

1.3.4. Morphisms

A morphism is a mapping from a source spec to a target spec. More precisely, it consists of two functions: one maps each type symbol of the source to a type symbol of the target, and the other maps each op symbol of the source to an op symbol of the target. The mapping must be type-consistent: if `f` of type `T` in the source spec is mapped to `g` of type `U` in the target spec, then `T` must be mapped to `U`. This mapping of types and ops can be lifted to terms, and thus to the axioms of the source spec. A morphism must be such that each axiom of the source spec maps to a theorem in the target spec: in other words, the translation of the axiom (according to the mapping expressed by the morphism) must be provable from the axioms in the target spec.

So, a morphism expresses that all the properties (i.e., the axioms) of the source spec are satisfied by the target spec. This is why refinement is expressed by means of morphisms: the source spec contains more abstract concepts; the target spec contains more concrete concepts, but all the properties of the abstract concepts must be satisfied by the concrete ones.

At the level of models, a morphism `m` induces a function that maps models of the target spec to models of the source spec (the function goes in the opposite direction of the morphism). The function operates as follows: given a model of the target spec, the corresponding model of the source spec is constructed as follows. The set assigned to a type `T` of the source spec is the set assigned to `m(T)` by the model of the target spec (or, if `T` is polymorphic, replace "set" with "set-valued function over sets"); the element assigned to an op `f` of type `T` of the source spec is the element (of the set assigned to `T`) assigned to `m(f)` by the model of the target spec (or, if `f` has a polymorphic type, replace "element" with "element-valued function over sets").

In other words, the morphism induces a reduction of the models of the target spec to models of the source spec. A model of the target spec can be reduced to a model of the source spec. This shows how a morphism can express an "is-a" relationship.

For example, if a spec imports another spec, possibly adding types, ops, and axioms, there is an inclusion morphism from the imported spec to the importing spec. Since all the types, ops, and axioms are mapped to themselves, the fact that axioms are mapped to theorems is immediate.

As another, less trivial example, consider a spec for natural numbers that also includes an op `plus` and an op `times`, both of type `Nat * Nat -> Nat`. (The construct "*" builds the cartesian product of two types: in a model, `A * B` denotes the cartesian product of the set denoted by `A` and the set denoted by `B`.) The spec also contains axioms that define `plus` and `times` to be addition and multiplication. Now, consider another spec consisting of a type `X`, an op `f` of type `X * X -> X`, and an axiom stating that `f` is commutative:

```
fa(x,y) f(x,y) = f(y,x)
```

There is a morphism from the latter spec to the former that maps `X` to `Nat` and `f` to `plus`: since addition is commutative, the commutativity axiom can be proved as a theorem in the spec for natural numbers. Note that there is also another morphism that maps `X` to `Nat` and `f` to `times`.

1.3.5. Diagrams and Colimits

A diagram is a graph whose nodes are labeled by specs and whose edges are labeled by morphisms. The morphism labeling an edge must be such that its source is the spec labeling the source node of the edge, and its target is the spec labeling the target node of the edge.

The colimit operation produces a spec from a diagram. The resulting spec is the gluing of the specs of the diagram, with the sharing expressed by the morphisms of the diagram.

In order to understand how the colimit operation works, consider first the simple case of a diagram without edges (and morphisms). This is called a discrete diagram. The colimit operation produces a spec whose types, ops, and axioms are the disjoint union of the types, ops, and axioms of the specs in the diagram. In other words, the specs are all glued together without any sharing.

Now, consider a diagram with some edges, labeled by morphisms. The colimit operation produces a spec containing all the types, ops, and axioms of the specs in the diagram, but all the types or ops that are linked, directly or indirectly, through the morphisms, are identified (i.e., they are the same type or op).

Consider, for example, a diagram with three nodes, `a`, `b`, and `c`, and two edges, one from `a` to `b` and the other from `a` to `c`. Node `a` is labeled by a spec consisting of a type `X`, node `b` by a spec consisting of two types `Y` and `Z`, and node `c` by a spec consisting of a type `W`. The morphism labeling the edge from `a` to `b` maps `X` to `Y`, and the one labeling the edge from `a` to `c` maps `X` to `W`. The colimit contains all types `X`, `Y`, `Z`, and `W`, but `X`, `Y`, and `W` are identified. So, the colimit effectively contains two types, one that can be referred to as `X`, `Y`, or `W`, and the other that can be only referred to as `Z`. For diagrams of this shape, with three nodes and two edges forming a wedge, the colimit operation is also called a pushout.

1.3.6. Substitutions

Given a spec `S` and a morphism `M`, it is possible (under certain conditions) to substitute the domain of `M` with its codomain inside `S`. Another way to say the same thing is that it is possible to “apply” the morphism `M` to the spec `S`.

Let `A` and `B` be the domain and codomain specs of `M`. The substitution operation is possible if and only if `A` is a sub-spec of `S`, in the sense that all the types, ops, and axioms of `A` are also in `S`. This is the case when `S` is constructed by importing and extending, directly or indirectly, `A`.

If that condition is satisfied, the result of the substitution is the spec `S'` that consists of all the types, ops, and axioms of `B` plus all the types, ops, and axioms of `S` that are not in `A`; the latter must all be translated according to the name mapping of `M`.

For example, suppose that:

1. `A` consists of a type `X`;
2. `S` consists of two types `X` and `Y` and an op `f` of type `X -> Y`;

3. `B` consists of a type `X'` and an op `c` of type `X'`;
4. `M` maps type `X` in `A` to type `X'` in `B`.

The result `S'` of the substitution consists of types `X'` and `Y`, an op `f` of type `X' -> Y`, and an op `c` of type `X'`. In other words, `A` is replaced with `B` inside `S` and the remaining portion of `S` is renamed accordingly.

1.3.7. Interpretations

A morphism maps a type or op of the source spec to a type or op of the target spec. In certain cases, it may be useful to map the type or op to a new type or op that is not present in the target spec but that can be defined in terms of those present in the target spec. This is captured by the concept of interpretation.

An interpretation is a morphism from a spec to a definitional extension of another spec. A definitional extension is an extension of a spec that only introduces new types and ops with axioms that define them in terms of those present in the spec that is being extended.

More precisely, an interpretation contains three specs: a source spec (the domain), a target spec (the codomain), and a mediator spec. The mediator is a definitional extension of the target spec, and there is an inclusion morphism from the target spec to the mediator. There is a morphism from the source spec to the mediator.

Consider, as an example, a spec for natural numbers without any `plus` op, but just with `zero` and `succ`, and with Peano's axioms. Consider the spec (also used as an example above) consisting of a type `X`, and op `f`, and a commutativity axiom about `f`. There is no morphism from the latter spec to the one for natural numbers. But there is an interpretation, where the mediator extends the spec for natural numbers with an op `plus` for addition, which can be inductively defined by the following two axioms:

```
fa(x) plus(x,zero) = x
fa(x,y) plus(x,succ y) = succ(plus(x,y))
```

A morphism can be viewed as a particular case of an interpretation, where the mediator is the same spec as the target, and the inclusion morphism from the target to the mediator is the identity morphism.

Diagrams of specs and morphisms can be generalized to diagrams of specs and interpretations: nodes are labeled by specs and edges by interpretations. The colimit operation works on these diagrams as well. The types and ops in the resulting spec include not only those from the specs labeling the nodes, but also those from the mediators of the interpretations.

Chapter 2. Example

This chapter presents a step-by-step example of a small application developed with Specware. Despite its small size, the example illustrates the general methodology to develop software with Specware. Furthermore, the description of the example includes concrete explanations to run the example through Specware, thus serving as a usage tutorial. The code for this example may be found in the `Examples\Matching\` (Windows) or `Examples/Matching/` (Linux) folder in your installation directory (e.g., `C:\Program Files\Specware\` or `/usr/local/specware`).

2.1. The Problem

The problem solved by this simple application is the following. A message is given, consisting of a sequence of characters, some of which are "obscured", i.e., it may look something like:

```
**V*ALN**EC*E*S
```

Then, a list of words is given, where a word is a sequence of characters, e.g.:

```
CERAMIC
CHESS
DECREE
FOOTMAN
INLET
MOLOCH
OCELOT
PROFUSE
RESIDE
REVEAL
SECRET
SODIUM
SPECIES
VESTIGE
WALNUT
YOGURT
```

Even though the above list is alphabetically sorted, the list of words can be given in any order.

The application must find which words of the list may occur somewhere in the message, where an obscured character in the message may match any character in a word, while a non-obscured character must be matched exactly. The application must also show the offset in the message at which the possible occurrence is found; if there is more than one offset, the smallest one should be returned. So, the output for the example message and word list above is the following:

```
10 CHESS
 8 DECREE
 0 REVEAL
 8 SECRET
 7 SPECIES
 3 WALNUT
```

Again, even though the list of words and numbers is alphabetically ordered, the application can produce it in any order.

2.2. Specification

2.2.1. Construction

We specify the application in a bottom-up fashion. We build specs for the concepts involved in the application, starting with the simplest ones up to the spec for the application.

Metaslang provides strings of characters among the built-in types, ops, and axioms. We could define messages and words as strings satisfying certain properties, e.g., that the character `*` can appear in messages but not in words, etc.

However, it is generally better to specify things as abstractly as possible. It is clear that the problem as stated above does not depend on obscured characters being represented as `*` and on non-obscured characters being uppercase letters only, or letters and numbers, or other. The abstract notion is that there are characters that form words, and messages are made of those characters but some characters may be obscured.

So, we start with a spec for symbols (i.e., characters):

```
Symbols = spec
  type Symbol
endspec
```

This is a very simple spec: it just consists of one type. This is perfectly adequate because the application treats symbols atomically: it only compares them for equality (which is built-in in Metaslang for every type). Any further constraint on symbols would just make the spec unnecessarily less general.

The text above is Metaslang syntax: it not only introduces a spec consisting of the type `Symbol`, but also assigns a name to it, `Symbols`. This spec can thus be referred to by its name, as shown shortly. The exact way in which the text above is supplied to Specware is explained later.

Now that we have the concept of symbols, we can introduce the concept of word as a sequence of symbols. Metaslang provides lists, built-in. The polymorphic type `List a` is used to define words:

```
Words = spec
  import Symbols
  type Word = List Symbol
endspec
```

The name of the spec is `Words`. The spec imports `Symbols` defined above, and extends it with a new type `Word`, defined to be `List Symbol`. This type is obtained as an instantiation of `List a` by replacing the type parameter `a` with `Symbol`.

A message is a sequence of symbols some of which may be obscured. This can be specified by lists whose elements are either symbols or a special extra value that stands for an "obscured symbol" (the "`*`" in the problem description given earlier).

Metaslang provides, built-in, a polymorphic type `Option a` that adds an extra element to a type. More precisely, it is defined as `type Option a = | Some a | None`. The type `Option a` is defined as a coproduct of type `a` tagged by `Some` and the singleton type consisting of the constant `None`.

So, we can define messages as follows:

```
Messages = spec
  import Symbols
  type Message = List (Option Symbol)
endspec
```

At this point, we can define the notion of symbol matching: a symbol must be matched by the exact same symbol, while an obscured symbol may be matched by any symbol. This is captured by the following spec:

```
SymbolMatching = spec
  import Symbols
```

```
  op symb_matches? : Symbol * Option Symbol -> Boolean
  def symb_matches?(s,os) = case os of Some s1 -> s = s1
                                     | None    -> true
endspec
```

The spec imports `Symbols` and extends it with an op `symb_matches?` that returns a boolean from a pair whose first component is a symbol and the second component is a possibly obscured symbol. This op can be viewed as a binary predicate. The op is defined by pattern matching on the second argument, in a straightforward way. Note that `s = s1` is a term of type `Boolean`.

The definition is constructive in the sense that code can be eventually generated directly, without any need to refine it. This is one of those cases where the simplest and most abstract definition of an op happens to be directly executable.

Having the notion of symbol matching, now we define the notion of word matching, i.e., when a word matches a message. We could define an op (predicate) of type `Word * Message -> Boolean`. However, since the application involves offsets for matching words, it is better to declare the op to have type `Word * Message * Nat -> Boolean`: the predicate is true if the given word matches the given message at the given position. Here is the spec:

```
WordMatching = spec

  import Words
  import Messages
  import SymbolMatching

  op word_matches_at? : Word * Message * Nat -> Boolean
  def word_matches_at?(wrd,msg,pos) =
      pos + length wrd <= length msg &&
      (fa(i:Nat) i < length wrd =>
         symb_matches?(nth(wrd,i), nth(msg,pos+i)))

endspec
```

First, the spec imports the specs for words, messages, and symbol matching. Then it introduces the op `word_matches_at?`, with the type explained above. Its definition says that the predicate holds on a triple `(wrd,msg,pos)` if and only if two conditions are satisfied:

1. `msg` is long enough to possibly contain the whole `wrd` at position `pos`;

2. every symbol of `wrd` at position `i` matches the corresponding, possibly obscured symbol in `msg` at position `pos + i`.

Note the use of `symb_matches?` previously defined. The ops `length` and `nth` are built-in, polymorphic ops over lists: the former returns the length of a list, while the latter returns the n-th element of a list (n is numbered starting from 0).

Unlike `symb_matches?` above, the definition of `word_matches_at?` is not executable. This means that it must be eventually refined. An executable definition of this op involves some kind of loop through the message: so, it would not be as simple and abstract as it is now. The general rule is that, at the specification level, things should be expressed as simply, clearly, and declaratively as possible.

Having all the above concepts in hand, we are ready to define how a message and a list of words are processed by the application to produce a list of matches. As explained in the problem description above, a match is not only a word, but also the (least) position in the message at which the match occurs. So, it is appropriate to define the concept of a match, as a pair consisting of a word and a position (i.e., a natural number):

```
Matches = spec
  import Words
  type Match = {word : Word, position : Nat}
endspec
```

The type `Match` is defined to be a record with two components, named `word` and `position`. A record is like a cartesian product, but the components have user-chosen names.

Finally, the spec for the whole application is the following:

```
FindMatches = spec

  import WordMatching
  import Matches

  op find_matches : Message * List Word -> List Match
  axiom match_finding is
        fa(msg,wrds,mtch)
          member(mtch,find_matches(msg,wrds)) <=>
          member(mtch.word,wrds) &&
          word_matches_at?(mtch.word,msg,mtch.position) &&
          (fa(pos) word_matches_at?(mtch.word,msg,pos) =>
                   pos >= mtch.position)

endspec
```

The spec imports `WordMatching` and `Matches`, and declares an op `find_matches` that, given a message and a list of words, returns a list of matches. This op captures the processing performed by the application. The axiom (whose name is `match_finding`) states the required properties. The built-in polymorphic op `member` is a predicate that says whether an element belongs to a list or not. So, the required property for `find_matches` is the following: given a message `msg` and a list of words `wrds`, a match `mtch` belongs to the result of `find_words` if and only if:

1. the word of the match is in `wrds`, i.e., the word must have been given as input;
2. the word can be matched with `msg` at the position indicated by the match;
3. the position of the match is the least position where the word matches.

Note how the specification of this word matching application is simple and abstract. No commitments have been made to particular data structures or algorithms. These commitments are made during the refinement process. Note also that the op `find_matches` is not completely defined by the axiom: the order of the resulting matches is not specified (the axiom only says what the members of the resulting list are, but not their order). The axiom does not prohibit duplicate elements in the output list; a suitable conjunct could be added to enforce uniqueness, if desired. This under-specification is consistent with the informal problem description given above. Of course, it is possible to change the axiom to require a particular order (e.g., the same as in the input list), if desired.

2.2.2. Processing by Specware

How do we actually enter the above text into Specware and how do we have Specware process it?

The current version of Specware works more or less like a compiler: it processes one or more files producing results. The files define specs, morphisms, etc. The results include error messages if something is wrong (e.g., type errors in specs), and possibly the generation of other files (e.g., containing code generated by Specware).

The files processed by Specware are text files with extension `.sw` (the “s” and “w” come from the first and fifth letter of “Specware”). A `.sw` file contains a sequence of definitions of specs, morphisms, etc.

For the word matching application constructed above, it is sensible to put all the specs above inside a file `MatchingSpecs.sw`:

```
Symbols = spec
```

```
  type Symbols
endspec

...

FindMatches = spec
 ...
endspec
```

Unlike a traditional compiler, the interaction with Specware is within a shell. When Specware is started, the shell is available for interaction with Specware (Specware is part of the Lisp image). In the shell, the user can move to any desired directory of the file system by means of the `cd` command, followed by the name of the directory, e.g., `cd c:\mydir` (Windows) and `cd ~/mydir` (Linux). Usually, the `.sw` files that form an application are put inside a directory, and from the shell the user moves to that directory.

In order to have Specware process a spec (or morphism, etc.) contained in a `.sw` file in the current directory, the user provides the following command in the shell:

```
proc <filename>#<specname>
```

The `<filename>` portion of the argument string is a place holder for the file name (without the `.sw` extension); the `<specname>` portion is a place holder for the spec (or morphism, etc.) name as it appears inside the file.

The effect of the above command is to have Specware process the indicated spec (or morphism, etc.) in the indicated file, recursively processing other specs, morphisms, etc. that are referenced by the indicated spec.

To have Specware process the spec of the word matching application, the command is:

```
proc MatchingSpecs#FindMatches
```

This has the effect of processing the spec named `FindMatches` in file `MatchingSpecs.sw`. Since this spec imports specs `WordMatching` and `Matches`, these are processed first, and so are the specs imported by them, recursively. Thus, all the specs in `MatchingSpecs.sw` are processed. `FindMatches` is the top-level spec in the file.

Specware finds the specs `WordMatching` and `Matches`, imported by `FindMatches`, because they are contained in the same file. As it will be explained shortly, it is possible to refer from one file to specs defined in different files.

2.3. Refinement

2.3.1. Construction

We now refine the application specified above in order to obtain a running program that implements the specified functionality. We do that by defining the specs and morphisms below inside a file `MatchingRefinements.sw`, in the same directory as `MatchingSpecs.sw`.

In order to obtain an executable program, we need to choose a concrete representation for the symbols composing words and messages. For example, we can choose uppercase characters:

```
Symbols = spec
  type Symbol = (Char | isUpperCase)
endspec
```

The built-in type `Char` is the type for characters. The built-in op `isUpperCase` is a predicate on characters that says whether a character is an uppercase letter or not. The subtype construct “`|`” is used to define the type `Symbol` as a subtype of `Char`.

Note that the above spec has the same name (`Symbols`) as its corresponding abstract spec. This is allowed and it is a feature of Specware: `.sw` files define separate name spaces. The file `MatchingSpecs.sw` creates a name space, and the file `MatchingRefinements.sw` creates a separate name space. The full name of the spec `Symbols` in `MatchingSpecs.sw` is `MatchingSpecs#Symbols`, while the full name of the spec `Symbols` in `MatchingRefinements.sw` is `MatchingRefinements#Symbols`. Indeed, when Specware is invoked to process a spec (or morphism, etc.), the full name is supplied, as in `proc MatchingSpecs#FindMatches`.

The fact that spec `MatchingRefinements#Symbols` is a refinement of `MatchingSpecs#Symbols` is expressed by the following morphism:

```
Symbols_Ref = morphism MatchingSpecs#Symbols ->
                       MatchingRefinements#Symbols {}
```

This text defines a morphism called `Symbols_Ref`, with domain `MatchingSpecs#Symbols` and codomain `MatchingRefinements#Symbols` and where the type `Symbol` in `MatchingSpecs#Symbols` is mapped to the type `Symbol` in `MatchingRefinements#Symbols`.

The specs `Words`, `Messages`, and `SymbolMatching` (in `MatchingSpecs.sw`) need not be refined, because they constructively define their types and ops. But the op `word_matches_at?` in `WordMatching` needs to be refined. We do that by constructing a spec that imports the same specs imported by `WordMatching` and that defines op `word_matches_at?` in an executable way:

```
WordMatching0 = spec

  import MatchingSpecs#Words
  import MatchingSpecs#Messages
  import MatchingSpecs#SymbolMatching

  op word_matches_at? : Word * Message * Nat -> Boolean
  def word_matches_at?(wrd,msg,pos) =
      if pos + length wrd > length msg
      then false
      else word_matches_aux?
            (wrd, if pos = 0 then msg
                               else nthTail(msg, pos - 1))

  op word_matches_aux? :
     {(wrd,msg) : Word * Message | length wrd <= length msg}
     -> Boolean
  def word_matches_aux?(wrd,msg) =
      case wrd of Nil -> true
               | Cons(wsym,wrd1) ->
                 let Cons(msym,msg1) = msg in
                 if symb_matches?(wsym,msym)
                 then word_matches_aux?(wrd1,msg1)
                 else false

endspec
```

Since the imported specs are not in the file `MatchingRefinements.sw`, their full names are used after `import`.

The definition of `word_matches_at?` makes use of an auxiliary op `word_matches_aux?`, which takes as input a word and a message such that the length of the word is not greater than that of the message. This constraint is expressed as a subtype of the cartesian product `Word * Message`. Op `word_matches_aux?` returns a boolean if the word matches the message, at the start of the message. It is defined recursively, by pattern matching on the word. Note the use of `let` to decompose the msg into the initial symbol and the rest. This is always possible because of the subtype constraint on the word and the message. So, `word_matches_at?` simply calls word_matches_aux? with the word and the tail of the message obtained by eliminating the first pos symbols, by means of the built-in op nthTail over lists.

The fact that WordMatching0 is a refinement of MatchingSpecs#WordMatching is expressed by the following morphism:

```
WordMatching_Ref0 = morphism MatchingSpecs#WordMatching ->
                             WordMatching0 {}
```

The refinement for word matching can be composed with the refinement for symbols constructed earlier. This is achieved by means of Specware's substitution operator:

```
WordMatching = WordMatching0[Symbols_Ref]
```

The resulting spec is like WordMatching0, but in addition the type Symbol is defined to consist of uppercase characters.

The fact that MatchingRefinements#WordMatching is a refinement of MatchingSpecs#WordMatching is expressed by the following morphism:

```
WordMatching_Ref =
   morphism MatchingSpecs#WordMatching ->
            MatchingRefinements#WordMatching {}
```

Now we proceed to refine op find_matches. We do that in two steps, analogously to word matching above. First, we build a spec FindMatches0 that imports the same specs imported by MatchingSpecs#FindMatches and that defines op find_matches in an executable way:

```
FindMatches0 = spec

  import MatchingSpecs#WordMatching
  import MatchingSpecs#Matches

  op find_matches : Message * List Word -> List Match
  def find_matches(msg,wrds) =
      foldl (fn(wrd,mtchs) ->
               case find_matches_aux(msg,wrd,0)
                 of Some pos ->
                    Cons({word = wrd, position = pos},
                         mtchs)
                  | None -> mtchs)
            Nil
            wrds
```

```
   op find_matches_aux : Message * Word * Nat -> Option Nat
   def find_matches_aux(msg,wrd,pos) =
       if pos + length wrd > length msg
       then None
       else if word_matches_at?(wrd,msg,pos)
            then Some pos
            else find_matches_aux(msg, wrd, pos + 1)

 endspec
```

Op `find_matches` makes use of the auxiliary op `find_matches_aux`, which takes as input a message `msg`, a word `wrd`, and a position `pos`. It returns either a natural number (a position where the match starts) or `None` if there is no match. Op `find_matches_aux` first checks if `msg` is long enough to possibly contain a match for `wrd` starting at `pos`. If that is not the case, `None` is returned. Otherwise, `word_matches_at?` is called: if it returns `true`, then the position `pos` is returned (wrapped by `Some`). Otherwise, `find_matches_aux` is recursively called, incrementing the position by 1. So, when `find_matches_aux` is called with 0 as its third argument, it iterates through the message to find the first match, if any. The position of the first match is returned, otherwise `None`.

The op `find_matches` iterates through the words of the list constructing a list of matches. The iteration is performed by means of the built-in op `foldl` for lists. For each word, `find_matches_aux` is called, with 0 as its third argument. Then, a pattern matching on the result is done: if the result is a position, a match is added to the output list; otherwise, the list is left unmodified.

The following morphism expresses that `FindMatches0` is a refinement of `MatchingSpecs#FindMatches`:

```
FindMatches_Ref0 = morphism MatchingSpecs#FindMatches ->
                            FindMatches0 {}
```

This refinement for `MatchingSpecs#FindMatches` can be composed with the one for `MatchingSpecs#WordMatching` built earlier. The composition is analogous to the one for `MatchingSpecs#WordMatching`:

```
FindMatches = FindMatches0[WordMatching_Ref]
```

The resulting spec includes the refinement for op `find_matches` as well as the refinement for op `word_matches_at?` and for symbols.

The fact that `MatchingRefinements#FindMatches` is a refinement of `MatchingSpecs#FindMatches` is expressed by the following morphism:

```
FindMatches_Ref =
  morphism MatchingSpecs#FindMatches ->
           MatchingRefinements#FindMatches {}
```

All the specs and morphisms of the file `MatchingRefinements.sw` can be processed by means of the following command:

```
proc MatchingRefinements#FindMatches_Ref
```

2.3.2. Proof Obligations

In general, a morphism has proof obligations associated to it: all the axioms in the source spec must be mapped to theorems in the target spec. These proof obligations are expressed in the form of a spec obtained by extending the target spec of the morphism with the axioms of the source spec (translated along the morphism) as conjectures. The user can then attempt to prove these conjectures using theorem provers linked to Specware.

Also specs have proof obligations associated to them. Some are part of typechecking, e.g., if an expression of a type `T` is used as an argument to a function whose domain is a subtype `T|p` of `T`, the expression must satisfy `p`, in the context in which the application occurs. Others arise from definitions via `def`: the equation must be uniquely define the op. The proof obligations associated to a spec ensure that the spec satisfied certain expected well-formedness properties, which is useful to validate the spec.

We collect the proof obligations of the specs and morphisms defined above in a file `MatchingObligations.sw`, in the same directory as `MatchingSpecs.sw` and `MatchingRefinements.sw`. The file contains the following definitions:

```
%%% obligations from MatchingSpecs.sw:

SymbolMatching_Oblig =
  obligations MatchingSpecs#SymbolMatching

WordMatching_Oblig =
  obligations MatchingSpecs#WordMatching

%%% obligations from MatchingRefinements.sw:

WordMatching0_Oblig =
  obligations MatchingRefinements#WordMatching0
```

```
WordMatching_Ref0_Oblig =
  obligations MatchingRefinements#WordMatching_Ref0

FindMatches0_Oblig =
  obligations MatchingRefinements#FindMatches0

FindMatches_Ref0_Oblig =
  obligations MatchingRefinements#FindMatches_Ref0
```

`SymbolMatching_Oblig` is a spec that expresses the proof obligations associated to the spec `SymbolMatching` as conjectures. Analogously for `WordMatching_Oblig`, `WordMatching0_Oblig`, and `FindMatches0_Oblig`. `WordMatching_Ref0_Oblig` and `FindMatches_Ref0_Oblig` are specs that express the proof obligations of the associated morphisms as conjectures. Specware provides the capability to display (i.e., print) any spec, morphism, etc., via the `show` command. This command can be used to see the proof obligations associated to the specs and morphisms, by displaying the specs expressing such obligations: the conjectures are the obligations.

Note that not all specs and morphisms from `MatchingSpecs.sw` and `MatchingRefinements.sw` are listed in `MatchingObligations`. Those that are not listed have really no proof obligations associated to them. For example, inspection (via `show`) of `obligations MatchingSpecs#Symbols` would reveal no conjectures because the spec is very simple (it just declared an abstract type). As another example, the spec `MatchingRefinements#WordMatching` is built by substituting `MatchingSpecs#Symbols` with `MatchingRefinements#Symbols` in `WordMatching0`: thus, the validity of the morphism `WordMatching_Ref` (i.e., that all axioms in the source are theorems in the target) follows immediately from the validity of the morphism `WordMatching_Ref0`.

The proof obligations associated to the spec `SymbolMatching` can be displayed by using the following command:

```
show MatchingObligations#SymbolMatching_Oblig
```

The spec only contains one conjecture, arising from the typechecking of the `case` expression used in the definition of `symb_matches?`, namely that the branches cover all possible cases, which is very easy to prove.

The spec `MatchingSpecs#WordMatching` has three proof obligations, all arising from subtypes, e.g., that `pos + i < length msg` in the context that (i.e., under the assumption that) `pos + length wrd <= length msg` and `i < length wrd`.

The morphism `FindMatches_Ref0` contains the “main” proof obligation of the overall refinement, namely that the definitio of `find_matches` in `MatchingRefinements#FindMatches0` satisfies the axiom in `MatchingSpecs#FindMatches`.

Specware provides the capability to invoke external theorem provers in order to attempt proofs of conjectures. Concretely, this is carried out by creating proof objects. Each proof object is associated with a certain conjecture in a certain spec; it also indicates the prover to use and some directives on how to perform the proof. Processing of the proof object invokes the indicated prover with the given directives. Currently, the only available prover is Snark; more provers will be added in the future.

For example, the user can attempt to discharge the proof obligation for `MatchingSpecs#SymbolMatching` by writing the following command in a file named `MatchingProofs.sw`, located in the same directory as `MatchingObligations.sw`:

```
p1 = prove symb_matches? in
     MatchingObligations#SymbolMatching_Oblig
```

At the shell, the user can issue this command to attempt the proof:

```
proc MatchingProofs#p1
```

2.3.3. Alternatives

The refinement for the word matching application developed above is certainly not the only one possible.

For example, we could have refined symbols differently. We could have refined them to be all letters (uppercase or lowercase) and numbers, or to be natural numbers. It is worth noting that the refinement for symbols is encapsulated in spec `MatchingRefinements#Symbols`. If we want to change the refinement for symbols, we just need to change that spec, while the other specs remain unaltered.

As another example, we could have chosen a more efficient refinement for op `find_matches`, using some fast substring search algorithm. In particular, while we have refined op `word_matches_at?` first, and then composed its refinement with one for `find_matches`, we could have refined `find_matches` “directly”, without using `word_matches_at?`, so that it would have been unnecessary to refine `word_matches_at?`.

The latter example illustrates an important principle concerning refinement. In general, it is not necessary to refine all ops present in a spec. Only the ops that are meant to be used by an external program need to be refined to be executable, and in turn the ops that are used inside their executable definitions. Other ops serve only an auxiliary role in specifying abstractly the ops that are meant to be eventually refined.

Currently, Specware provides no support to indicate explicitly which ops are meant to be "exported" by a spec. In future versions of the system, some functionality like this may be included.

2.4. Code Generation and Testing

2.4.1. Code Generation

Now that our simple word matching application has been refined to be executable, we can generate code from it. This is concretely achieved by creating a (Lisp) program from a spec, indicating a file name where the code is deposited as a side effect. The following command is given in the shell:

```
gen-lisp <specname> <targetfilename>
```

The above command first processes the named spec and then generates Lisp code from it, writing the code into the indicated file. The file is created if it does not exist; if it exists, it is overwritten.

For instance, we can generate Lisp code for the word matching application by means of the following command:

```
gen-lisp MatchingRefinements#FindMatches find-matches
```

Note that if the `.lisp` suffix is omitted, Specware adds it to the file name.

After we generate the code, we can then try to run it by calling the Lisp function produced from op `find_matches`. Since messages and words are represented as lists of characters (plus `None` for messages), it would be slightly inconvenient to enter and read lists of characters. It would be better to enter strings with letters and `*`'s, as shown in the informal problem description at the beginning of this chapter.

In order to do that, we define translations between strings and messages and words, and we wrap `find_matches` in order to translate from and to strings. We do that inside a spec called `Test` in a file `MatchingTest.sw` in the same directory as the other `.sw` files:

```
Test = spec

  import MatchingRefinements#FindMatches

  op word_char? : Char -> Boolean
  def word_char? ch = isUpperCase ch

  op msg_char? : Char -> Boolean
  def msg_char? ch = isUpperCase ch or ch = #*

  type WordString = (String | all word_char?)

  type MessageString = (String | all msg_char?)

  op word2string : Word -> WordString
  def word2string wrd = implode wrd

  op string2word : WordString -> Word
  def string2word wstr = explode wstr

  op message2string : Message -> MessageString
  def message2string msg =
      implode(map (fn Some ch -> ch
                    | None -> #*)
                  msg)

  op string2message : MessageString -> Message
  def string2message mstr =
      map (fn ch -> if ch = #* then None else Some ch)
          (explode mstr)

  type MatchString = {word : WordString, position : Nat}

  op match2string : Match -> MatchString
  def match2string mtch =
      {word     = word2string mtch.word,
       position = mtch.position}

  op test_find_matches :
     MessageString * List WordString -> List MatchString
  def test_find_matches(mstr,wstrs) =
```

```
      map match2string
          (find_matches(string2message mstr,
                        map string2word wstrs))

endspec
```

Since the translation is not defined on all strings, we introduce two subtypes of the built-in type `String`: the type `WordString` consists of all strings whose characters are uppercase letters; the type `MessageString` consists of all strings whose characters are uppercase letters or `*`. The built-in op `all` over strings is used to define the subtypes, using the ops (predicates) `word_char?` and `msg_char?`.

The op `word2string` translates a word to a word string, by means of the built-in op `implode`. The op `string2word` performs the opposite translation, using the built-in op `explode`. The ops `message2string` and `string2message` translate between messages and string messages. Besides the use of `implode` and `explode`, they need to map `Some ch` from/to `ch` (where `ch` is an uppercase letter) and `None` from/to `*`.

Since `find_matches` returns a match, i.e., a pair consisting of a word and a position, we define a type `MatchString` consisting of a word string and a position, together with an op `match2string` that translates from a match to a string match.

Finally, we define an op `test_find_matches` to take a message string and a list of word strings as input, and to return a list of string matches as output. The message string and word strings are first translated to a message and list of words, then `find_matches` is called, and then the resulting matches are translated to string matches. Note that the op `message2string` is never used. In fact, it could have been omitted.

Now, instead of generating code from `MatchingRefinements#FindMatches`, we generate code from `MatchingTest#Test`:

```
gen-lisp MatchingTest#Test find-matches-test.lisp
```

2.4.2. Testing

In order to test the generated Lisp code, we need to load the generated Lisp file into a Lisp environment (provided with Specware). We do that by means of the following command (this and all the following commands must be given in the Lisp environment, not the Specware shell):

```
:ld find-matches-test
```

In Lisp, entities like functions, constants, etc. are defined inside packages. When Specware generates code, it puts it inside a package named `sw-user`. This can be seen from the package declaration at the beginning of file `find-matches-test.lisp` (note that Lisp is case-insensitive).

So, in order to call the functions defined in that file (after it has been loaded), it is necessary either to prepend the package name to them, or to move to that package and then call them without package qualification. To follow the first approach, we would write `(sw-user::test_find_matches-2 <arg1> <arg2>)` to call the function. (The "`-2`" appended to the function name is for the two-argument version of the Lisp function; see the Section on Arity and Currying Normalization in the User Manual. Alternatively, we can supply a single argument by consing `<arg1>` and `<arg2>`, thus: `(sw-user::test_find_matches (cons <arg1> <arg2>))`.)

To follow the second approach, we would first change package by means of the Lisp command `:pa sw-user`, and then we can just write `(test_find_matches-2 <arg1> <arg2>)` to call the function.

In order to test the program on the example input and output given at the beginning of the chapter, we proceed as follows. First, we define a variable containing the message:

```
(setq msg "**V*ALN**EC*E*S")
```

Then we define a variable containing the list of words:

```
(setq words '("CERAMIC" "CHESS" "DECREE" "FOOTMAN"
              "INLET" "MOLOCH" "OCELOT" "PROFUSE"
              "RESIDE" "REVEAL" "SECRET" "SODIUM"
              "SPECIES" "VESTIGE" "WALNUT" "YOGURT"))
```

Finally, we call (assuming to be in package `sw-user`):

```
(test_find_matches-2 msg words)
```

The following result is then displayed:

```
((3 . "WALNUT") (7 . "SPECIES") (8 . "SECRET")
 (0 . "REVEAL") (8 . "DECREE") (10 . "CHESS"))
```

The result is a list of pairs, each of which represents a match.