*8*

## Scoping and Linking

*Linking refers to the resolution of name-based references to the referenced symbols in parser-based languages. In projectional systems this is not necessary, since every reference is stored as a direct pointer to the target element. However, in both cases we have to define which elements are actually visible from a given reference site. This information serves as the basis for code completion and to check existing references for their validity. The set of visible elements for a given reference is called its scope.*

|  |  |
| --- | --- |
| As we discussed in the previous chapter, the abstract syntax in its simplest form is a tree. However, the information represented by the program is semantically almost always a graph; i.e. in addition to the tree’s containment hierarchy, it contains |  |
| non-containment cross-references1. The challenge thus is: how |  |
| to get from the "syntactic tree" to the "semantic graph" – or, how to establish the cross-links. There is a marked difference between the projectional and parser-based approach: |  |

* In parser-based systems, the cross-references have to be *resolved*, from the parsed text *after* the AST has been created. An IDE may provide the candidates in a code completion menu, but after selecting a target, the resulting textual representation of the reference must contain all the information to *re-resolve* the reference each time the program is parsed.
* In projectional editors in which every program element has a unique ID, a reference is represented as a pointer to that ID. Once a reference is established, it can always be re-resolved trivially based on the ID. The reference is established directly as the program is edited: the code completion menu shows candidate target elements for a reference in the code completion menu, and selection of one of them creates the reference2.

|  |  |
| --- | --- |
| Typically, a language’s structure definition specifies which concepts constitute valid target concepts for any given reference  (e.g., a **Function**, a **Variable** or a **State**), but this is usually not enough. Language-specific visibility rules determine which *instances* of these concepts are actually permitted as a |  |
| reference target3. The collection of model elements which are |  |
| valid targets of a particular semantic cross-reference is called the *scope* of that cross-reference. Typically, the scope of a particular cross-reference not only depends on the target concept of the cross-reference, but also on its surroundings, e.g. the namespace within which the element lives, the location inside the larger structure of the site of the cross-reference or something that’s essentially non-structural in nature.  A scope, the collection of valid targets for a reference, has two uses. First, it can be used to populate the code completion menu in the IDE if the user presses **Ctrl-Space** at the reference site. Second, independent of the IDE, the scope is used for checking the validity of an existing reference: if the reference target is not among the elements in the scope, the reference is |  |
| invalid. |  |

Scopes can be hierarchical, in which case they are organized as a stack of collections – confusingly, these collections are often called scopes themselves. During resolution of a crossreference, the lowest or *innermost* collection is searched first. If the reference cannot be resolved to match any of its elements, the parent of the innermost collection is queried, and so forth.

The hierarchy often mimics the structure of the language itself: e.g., the innermost scope of a reference consists of all the elements present in the immediately-surrounding "block", while the outermost scope is the *global* scope. This provides a mechanism to disambiguate target elements having the same reference syntax (usually the target element’s name) by always choosing the element from the innermost scope. This is often called *shadowing*, because the inner elements overshadow the (more) outer elements.

### 8.1 Scoping in Spoofax

In the previous chapter we described how to specify a grammar for a subset of the Mobl language. This chapter shows how to specify name resolution for this language by means of declarative name binding rules. Spoofax’ name binding rules are based on five concepts: namespaces, definitions, references, scopes and imports. We will introduce each of these concepts separately, going from simple to more complicated examples.

#### 8.1.1 Namespaces

To understand naming in Spoofax, the notion of a *namespace* is essential. In Spoofax, a namespace is a collection of names and is not necessarily connected to a specific language concept4.

Different concepts can contribute names to a single namespace. For example, in Java, classes and interfaces contribute to the same namespace. Namespaces are declared in the **namespace** section of a language definition. For Mobl, we have separate namespaces for modules, entities, properties, functions and local variables.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **namespaces** Module Entity Property Function Variable   |  |  | | --- | --- | | *8.1.2 Definitions and References*  Once we have defined namespaces, we can define name bindings with rules of the form **pattern : clause\***, where **pattern** |  | | is a term pattern5, and **clause\*** is a list of name binding declarations about the language construct that matches with **pattern**. For example, the following rules declare definition sites for module and entity names. The patterns in these rules match module and entity declarations, binding variables **m** and **e** to module and entity names respectively. These variables are then |  | | used in the clauses on the right-hand sides6. |  | |

Module(m, \_): **defines non**-**unique** Module m

Entity(e, \_): **defines unique** Entity e

As an example, let us reconsider the example module from the previous chapter:

|  |
| --- |
| **module** shopping  **entity** Item {  name : String  ...  } |

The parser turns this into an abstract syntax tree, represented as a term:

Use sites which refer to definition sites of names can be declared similarly. For example, the following rule declares use sites of entity names:

Type(t): **refers to** Entity t

|  |
| --- |
| Module(  "shopping", [ Entity(  "Item",  [Property("name", EntityType("String")), ...] ) ]  ]) |

|  |
| --- |
| The patterns in name binding rules match subterms of this term, indicating definition and use sites. The whole term is a definition site of the module name **shopping**. The first name binding rule specifies this binding. Its pattern matches the term and binds **m** to **"shopping"**. Similarly, the subterm **Entity ("Item", ...)** is a definition site of the entity name **Item**. The pattern of the second name binding rule matches this term and binds **e** to **"Item"**.  While entity declarations are unique definition sites, module declarations are non-unique definition sites. That is, multiple module declarations can share the same name. This allows Mobl users to spread the content of a module over several files, similar to Java packages. Namespaces are by default unique, so the **unique** keyword is only optional and can be omitted. For example, the following rules declare unique definition sites for property and variable names:  Property(p, \_): **defines** Property p Param(p, \_) : **defines** Variable p  Declare(v, \_) : **defines** Variable v  Note how Spoofax distinguishes the name of a namespace from the sort and the constructor of a program element: in the last rule above, the sort of the program element is **Statement**, its constructor is **Declare**, and it lives in the **Variable** namespace. By distinguishing these three things, it becomes easy to add or exclude program elements from a namespace[[1]](#footnote-1). |

Use sites might refer to different names from different namespaces. For example, a variable might refer either to a **Variable** or a **Property**. In Spoofax, this can be specified by exclusive resolution options:

|  |
| --- |
| Var(x):  **refers to** Variable x **otherwise refers to** Property x |

also of the syntactic sort **Statement**, but do not live in any namespace. On the other hand, function parameters also live in the **Variable** namespace, even though (in contrast to variable declarations) they do not belong to the syntactic sort **Statement**.

The **otherwise** keyword signals ordered alternatives: only if the reference cannot be resolved to a variable will Spoofax try to resolve it to a property. As a consequence, variable declarations shadow property definitions. If this is not intended, constraints can be defined to report corresponding errors. We will discuss constraints in Section 9.3 in the next chapter.

#### 8.1.3 Scoping

*Simple Scopes* In Spoofax, *Scopes* restrict the visibility of definition sites8. For example, an entity declaration scopes

property declarations that are not visible from outside the entity.

**entity** Customer {

name : String // Customer.name

}

**entity** Product {

name : String // Product.name

}

In this example, both **name** properties live in the **Property** namespace, but we can still distinguish them: if **name** is referenced in a function inside **Customer**, then it references the one in **Customer**, not the one in **Product**.

Scopes can be nested and name resolution typically looks for definition sites from inner to outer scopes. In Mobl, modules scope entities, entities scope properties and functions, and functions scope local variables. This can be specified in Spoofax in terms of **scopes** clauses:

Module(m, \_): **defines** Module m **scopes** Entity

Entity(e, \_): **defines** Entity e **scopes** Property, Function Function(f, \_): **defines** Function f **scopes** Variable

|  |  |
| --- | --- |
| As these examples illustrate, scopes are often also definition sites. However, this is not a requirement. For example, a block |  |
| statement9 has no name, but scopes variables: |  |

Block(\_): **scopes** Variable

*Definition Sites with Limited Scope* So far we have seen examples in which definitions are visible in their enclosing scope: entities are visible in the enclosing module, properties and functions are visible in the enclosing entity, and parameters are visible in the enclosing function. However, this does not hold for variables declared inside a function. Their visibility is limited to statements *after* the declaration. Thus, we need to restrict the visibility in the name binding rule for **Declare** to the *subsequent scope*:

Declare(v, \_): **defines** Variable v **in subsequent scope**

Similarly, the iterator variable in a **for** loop is only visible in its condition, the update, and the loop’s body, but not in the initializing expression. This can be declared as follows:

|  |
| --- |
| For(v, t, init, cond, update, body):  **defines** Variable v **in** cond, update, body |

*Scoped References* Typically, use sites refer to names which are declared in its surrounding scopes. But a use site might also refer to definition sites which reside outside its scope. For example, a property name in a property access expression might refer to a property in another entity:

|  |
| --- |
| **entity** Customer {  name : String  }  **entity** Order { customer : Customer **function** getCustomerName(): String { **return** customer.name;  }  } |

Here, **name** in **customer.name** refers to the property in entity **Customer**. The following name binding rule is a first attempt to specify this:

PropAccess(exp, p): **refers to** Property p **in** Entity e

But this rule does not specify which entity **e** is the right one.

Interaction with the type system10 is required in this case:

|  |
| --- |
| PropAccess(exp, p):  **refers to** Property p **in** Entity e **where** exp **has type** EntityType(e) |

Section 10.5.

This rule essentially says: give me a property with the name **p** in entity **e**, where **e** is the type of the current expression **exp**.

*Imports* Many languages offer import facilities to include definitions from another scope into the current scope. For example, a Mobl module can import other modules, making entities from the imported modules available in the importing module:

**module** order **import** banking **entity** Customer {

|  |
| --- |
| name : String  account: BankAccount } |

Here, **BankAccount** is not declared in the scope of module **order**. However, module **banking** declares an entity **BankAccount**, which is imported into module **order**. The type of property **account** should refer to this entity. This can be specified by the following name binding rule:

Import(m): **imports** Entity **from** Module m

This rule has two effects. First, **m** is interpreted as a name referring to a module. Second, every entity declared in this module becomes visible in the current scope.

#### 8.1.4 References in Terms

Spoofax uses terms to represent abstract syntax. This enables many interesting features, for example generic tree traversals. But in contrast to object structures as used in MPS and Xtext, terms lack a native concept to represent cross-references. There are two approaches to handle cross-references when working with terms or similar tree structures. First, we can maintain a temporary environment with required information about defined elements during a transformation. This information can then be accessed at use sites. Second, we can maintain similar information in a global environment, which can be shared by various transformations.

Spoofax follows the second approach and stores all definitions and references in an in-memory data structure called the index11. By collecting all this summary information about files

11 Spoofax also uses the index to store

in a project together, it ensures fast access to global information (in particular, to-be-referenced names). The index is updated automatically when Spoofax model files change (or are deleted) and is persisted as Eclipse exits. All entries in the index have a URI which uniquely identifies the element across a project. These URIs are the basis for name resolution, and, by default, are constructed automatically, based on the name binding rules. As an example, consider the following entity:

|  |
| --- |
| **module** storage  **entity** Store {  name : String  address : Address  } |

Following the name binding rules discussed so far, there are two scope levels in this fragment: one at the module level and

metadata about definitions, such as type information, as we show in the next chapter.

one at the entity level. We can assign names to these scopes (**storage** and **Store**) by using the naming rules for modules and entities. By creating a hierarchy of these names, Spoofax creates URIs: the URI for **Store** is **Entity://storage.Store**, and the one for **name** is **Property://storage.Store.name**. URIs are represented internally as lists of terms that start with the namespace, followed by a reverse hierarchy of the path names12.

For the **name** property of the **Store** entity in the **storage** module, this would be:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| [Property(), "name", "Store", "storage"]   |  |  | | --- | --- | | Spoofax annotates each definition and reference with a URI to connect names with information stored in the index. References are annotated with the same URI as their definition. This way, information about the definition site is also available at the reference. We can inspect URIs in Spoofax’ analyzed syntax view. This view shows the abstract syntax with all URIs |  | | as annotations13. Consider the following example with both |  | |

sentation makes it easier to efficiently store and manipulate URIs in memory: every tail of such a list can share the

named and anonymous blocks:

|  |
| --- |
| **module** banking  **entity** BankAccount {  name : String number : Num  **function** toString() : String {  { // anonymous block **var** result = name + number.toString(); **return** result;  }  }  } |

The analyzed abstract syntax for this example is the following:

|  |
| --- |
| Module( "banking"{[Module(),"banking"]},  [ Entity(  "BankAccount"{[Entity(),"BankAccount","banking"]},  [ Property(  "name"{[Property(),"name","BankAccount","banking"]},  StringType()  ),  Property( "number"{[Property(),"number","BankAccount","banking"]},  NumType()  ),  Function( "toString"{[Function(),"toString","BankAccount","banking"]},  [],  StringType(),  Block([  Declare("result"{[Var(),"result",Anon(125),Anon(124),  "toString","BankAccount","banking"]},  Add(  Var("name"{[Property(),"name","BankAccount","banking"]}),  MethodCall(..., "toString"{[Unresolved(Function()),  "toString", "BankAccount", "banking"]}) ) |

*lyzed Syntax (selection)* in the *Transform* menu of the Spoofax editor. Spoofax will open a new editor which updates automatically when the content of the original editor changes.

|  |
| --- |
| ),  Return(  Var("result"{[Var(),"result",Anon(125),Anon(124),  "toString","BankAccount","banking"]})  )  ])  )  ]  )  ]  ) |

Any references that cannot be resolved are annotated with a special **Unresolved** constructor. For example, a variable named **nonexistent** could be represented as:

Var("nonexistent"{[Unresolved(Var()),"non\-existent",...]})

This makes it easy to recognize any unresolved references in constraints14: we can simply pattern-match against the **Unre-**

**solved** term.

### 8.2 Scoping in Xtext

|  |  |
| --- | --- |
| Xtext provides Java APIs for implementing all aspects of lan- |  |
| guages except the grammar15. Language developers typically |  |
| provide Java classes that implement aspect-specific interfaces and then contribute those to Xtext using dependency injec- |  |
| tion16. For most language aspects, Xtext comes with various |  |
| default implementations developers can build on. A lot of functionality is provided "out of the box" with minimal config- |  |

uration, but it’s easy to swap specific parts by binding another or a custom class through Guice.

|  |
| --- |
| **public interface** IScopeProvider {  IScope getScope(EObject context, EReference reference);  } |

#### 8.2.1 Simple, Local Scopes

To implement scopes, language developers have to contribute a class that implements the **IScopeProvider** interface. It has one method called **getScope** that returns an **IScope** for a given reference. An **IScope** is basically a collection of candidate reference targets, together with the textual representation by which these may be referenced from the current reference site (the same target may be referenced by different text strings from different program locations). The **getScope** method has two arguments: the first one, **context**, is the current program element for which a reference should be scoped; the second one, **reference**, identifies the reference for which the scope that needs to be calculated[[2]](#footnote-2).

To make the scoping implementation easier, Xtext provides *declarative scope providers* through the **AbstractDeclarativeScopeProvider** base class: instead of having to inspect the **reference** and **context** object manually to decide how to compute the scope, the language implementor can express this information via the name of the method (using a naming convention). Two different naming conventions are available:

|  |
| --- |
| // <X>, <R>: scoping the <R> reference of the <X> concept **public** IScope scope\_<X>\_<R>(<X> ctx, EReference ref );  // <X>: the language concept we are looking for as a reference target // <Y>: the concept from under which we try to look for the reference **public** IScope scope\_<X>(<Y> ctx, EReference ref); |

Let’s assume we want to scope the **targetState** reference of the **ChangeStateStatement**. Its definition in the grammar looks like this:

|  |
| --- |
| ChangeStateStatement:  "state" targetState=[State]; |

We can use the following two alternative methods:

|  |
| --- |
| **public** IScope scope\_ChangeStateStatement\_targetState (ChangeStateStatement ctx, EReference ref ) { ... }  **public** IScope scope\_State(ChangeStateStatement ctx, EReference ref) { ...  } |

The first alternative is specific for the **targetState** reference of the **ChangeStateStatement**. It is invoked by the declarative scope provider only for that particular reference. The second alternative is more generic. It is invoked whenever we are trying to reference a **State** (or any subconcept of **State**) from any reference of a **ChangeStateStatement** and *all its descendants* in the AST. So we could write an even more general alternative, which scopes the visible **State**s from anywhere in a **CoolingProgram**, independent of the actual reference18.

**public** IScope scope\_State(CoolingProgram ctx, EReference ref) {

...

}

The implementation of the scopes is simple, and relatively similar in all three cases. We write Java code that crawls up the containment hierarchy until we arrive at a **CoolingProgram** (in the last alternative, we already get the **CoolingProgram** as an argument, so we don’t need to move up the tree), and then construct an **IScope** that contains the **State**s defined in that **CoolingProgram**. Here is a possible implementation:

(**scope\_ChangeStateStatement\_targetState** in the example) might not be called in all the places you expect it to be called. This can be remedied either by changing the syntax (often not possible or not desired), or by using the more general variants of the scoping function **scope\_State( CoolingProgram ctx, ... )**.

|  |
| --- |
| **public** IScope scope\_ChangeStateStatement\_targetState  (ChangeStateStatement ctx, EReference ref ) {  CoolingProgram owningProgram =  Utils.ancestor( ctx, CoolingProgram.**class** ); **return** Scopes.scopeFor(owningProgram.getStates()); } |

The **Scopes** class provides a couple of helper methods to create **IScope** objects from collections of elements. The simple **scopeFor** method will use the **name** of the target element as the text by which it will be referenced19. So if a state is called

**normalCooling**, then we’d have to write **state normalCooling** in a **ChangeStateStatement** in order to change to that state. The text **normalCooling** acts as the reference – pressing **Ctrl-F3** on that program element will go to the referenced state.

#### 8.2.2 Nested Scopes

The approach to scoping shown above is suitable for simple cases, such as the **targetState** reference shown above. However, in languages with nested blocks a different approach is recommended. Here is an example of a program expressed in a language with nested blocks:

|  |  |
| --- | --- |
| **var int** x; **var int** g;  **function** add( **int** x, **int** y ) { **int** sum = x + y; | // 1 |
| **return** sum;  }  **function** addAll( **int** es ... ) { **int** sum = 0; **foreach**( e **in** es ) { sum += e; | // 2 |
| }  x = sum;  } | // 3 |

At the program location marked as 1, the local variable **sum**, the arguments **x** and **y** and the global variables **x** and **g** are visible, although the global variable **x** is shadowed by the argument of the same name. At 2, we can see **x**, **g**, **sum** and **es**, but also the iterator variable **e**. At 3, **x** refers to the global, since it is not shadowed by a parameter or local variable of the same name. In general, some program elements introduce blocks (often statement lists surrounded by curly braces). A block can declare new symbols. References from within these blocks can see the symbols defined in that block, as well as all ancestor blocks. Symbols in inner blocks typically hide symbols with the same name in outer blocks[[3]](#footnote-3).

Xtext’s scopes support this scenario. **IScopes** can reference outer scopes. If a symbol is not found in any given scope, that scope delegates to its outer scope (if it has one) and asks it for a symbol of the respective name. Since inner scopes are searched first, this implements shadowing as expected.

Also, scopes are not just collections of elements. Instead, they are maps between a string and an element21. The string

is used as the reference text. By default, the string is the same as the target element’s **name** property. So if a variable is called **x**, it can be referenced by the string **x**. However, this reference string can be changed as part of the scope definition. This can

be used to make shadowed variables visible under a different name, such as **outer.x** if it is referenced from location 1. The following is pseudo-code that implements this behavior:

|  |
| --- |
| // recursive method to build nested scopes **private** IScope collect( StatementList ctx ) { IScope outer = **null if** ( ctx is within another StatementList parent ) { outer = collect(parent)  }  IScope scope = **new** Scope( outer ) **for**( all symbols s in ctx ) { scope.put( s.name, s ) **if** ( outer.hasSymbolNamed( s.name ) ) { scope.put( "outer."+s.name, outer.getSymbolByName( s.name ) )  } }  **return** scope  }  // entry method, according to naming convention  // in declarative scope provider **public** IScope scope\_Symbol( StatementList ctx ) { **return** collect( ctx ) } |

#### 8.2.3 Global Scopes

There is one more aspect of scoping that needs to be discussed. Programs can be separated into several files and references can cross file boundaries. That is, an element in file **A** can reference an element in file **B**. In earlier versions of Xtext file **A** had to explicitly import file **B** to make the elements in **B** available as reference targets[[4]](#footnote-4). Since Xtext 1.0 both of these problems are solved using the emphindex[[5]](#footnote-5). The index is a data structure that stores (**String**,**IEObjectDescription**)-pairs. The first argument is the qualified name of the object, and the second one, the **IEObjectDescription**, contains information about a model element, including a URI (a kind of global pointer that also includes the file in which the element is stored) as well as arbitrary additional data provided by the language implementation. By default, all references are checked against this name in the index, not against the actual object. If the actual object has to be resolved, the URI stored in the index is used. Only then is the respective file loaded24. The index is updated

|  |
| --- |
| AtomicLevel **returns** Expression:  ...  ({SymbolRef} symbol=[SymbolDeclaration|QID]); |

|  |  |
| --- | --- |
| whenever a file is changed25. This way, if an element is moved to a different file while keeping its qualified name (which is based on the logical program structure) constant, the reference remains valid. Only the URI in the index is updated.  There are two ways to customize what gets stored in the index, and how. The **IQualifiedNameProvider** returns a qualified name for each program element. If it returns **null**, the element is not stored in the index, which means it is not referenceable. The other way is the **IDefaultResourceDescriptionStrategy**, which allows language developers to build their own **IEObjectDescription** for program elements. This is important if custom user data has to be stored in the **IEObjectDescription** for later use during scoping.  The **IGlobalScopeProvider** is activated if a local scope returns **null** or no applicable methods can be found in the declarative scope provider class (or if they return **null**). By default, the **ImportNamespacesAwareGlobalScopeProvider** is config- |  |
| ured26, which provides the possibility of referencing model |  |
| elements outside the current file, either through their (fully) qualified name, or through their unqualified name if the re- |  |
| spective namespace is imported using an **import** statement27. |  |
| *Polymorphic References* In the cooling language, expressions also include references to entities such as configuration parameters, variables and hardware elements (compressors or fans defined in a different model). All of these referenceable elements extend **SymbolDeclaration**. This means that all of them can be referenced by the single **SymbolRef** construct. |  |
| The problem with this situation is that the reference itself does |  |
| not encode the kind of thing that is referenced28. This makes |  |
| writing code that processes the model cumbersome, since the target of a **SymbolRef** has to be taken into account when deciding how to treat (translate, validate) a symbol reference. A more natural design of the language would use different refer- |  |

ence constructs for the different referenceable elements. In this case, the reference itself is specific to the referenced element, making processing much easier[[6]](#footnote-6):

|  |
| --- |
| AtomicLevel **returns** Expression:  ...  ({VariableRef} var=[Variable]);  ({ParameterRef} param=[Parameter]);  ({HardwareBuildingBlockRef} hbb=[HardwareBuildingBlock]); |

However, this is not possible with Xtext, since the parser cannot distinguish the three cases syntactically. As we can see from the (invalid) grammar above, in all three cases the reference syntax itself is just an **ID**. Only during the linking phase could the system check which kind of element is actually referenced, but this is too late for the parser, which needs an unambiguous grammar. The grammar could be disambiguated by using a different syntax for each element:

|  |
| --- |
| AtomicLevel **returns** Expression:  ...  ({VariableRef} var=[Variable]);  ({ParameterRef} "%" param=[Parameter]);  ({HardwareBuildingBlockRef} "#" hbb=[HardwareBuildingBlock]); |

While this approach will technically work, it would lead to an awkward syntax and is hence typically not used. The only remaining alternative is to make all referenceable elements extend **SymbolDeclaration** and use a single reference concept, as shown above.

### 8.3 Scoping in MPS

Making references work in MPS requires several ingredients. First of all, as we have seen earlier, the reference is defined as part of the language structure. Next, an editor is defined that determines how the referenced element is rendered at the referencing site[[7]](#footnote-7). To determine which instances of the referenced

concept are allowed, a scoping function has to be implemented. This simply returns a list of all the elements that are considered valid targets for the reference, as well as an optional text string used to represent the respective element in the code completion menu.

As we explained above (Section 7.2), smart references are an important ingredient to make this work conveniently. They make sure that users can simply type the name (or whatever else is put into the code completion menu by the language developer) of the targeted element; once something is selected, the corresponding reference concept is instantiated, and the selected target is set.

*Simple Scopes* As an example, we begin with the scope definition for the target reference of the **Transition** concept. To recap, it is defined as:

|  |
| --- |
| **concept** Transition // ...  **references**:  State target 1 |

The scope itself is defined via the search scope constraint below. The system provides an anonymous **search scope** function that has a number of arguments that describe the context including the enclosing node and the referencing node. As the signature shows, the function has to return either an **ISearchScope** or simply a sequence of nodes of type **State**. The scope of the target state is the set of states of the state machine that (transitively) contains the transition. To implement this, the expression in the body of this function crawls up the containment hierarchy31 until it finds a **Statemachine** and then returns its

**states**32.

|  |
| --- |
| **link** {target} **referent set handler**:  <none> **search scope**:  (referenceNode, linkTarget, enclosingNode, ...)  ->join(ISearchScope | sequence<node<State>>) { enclosingNode.ancestor<Statemachine>.states;  } **validator**:  <default> **presentation** : <none> |

|  |  |
| --- | --- |
| In addition to the search scope, language developers can provide code that should be executed if a new reference target is set (**referent set handler**), additional validation (**validator**), as well as customized presentation in the code completion menu |  |
| (**presentation**)33. |  |
| *Nested Scopes* In a more complex, block-oriented language with nested scopes, a different implementation pattern is rec- |  |
| ommended34: |  |

* All program elements that contribute elements that can be referenced (such as blocks, functions or methods) implement an interface **IScopeProvider**.
* The interface provides **getVisibleElements(concept<> c)**, a method that returns all elements of type **c** that are available in that scope.
* The search scope function simply calls this method on the owning **IScopeProvider**, passing in the concept whose instances it wants to see (**State** in the above example).
* The implementation of the method recursively calls the method on its owning **IScopeProvider**, as long as there is one. It also removes elements that are shadowed from the result.

This approach is used in the mbeddr C language, for example for local variables, because those are affected by shadowing from blocks. Here is the code for the **variable** reference of the **LocalVariableReference** concept:

|  |
| --- |
| **link** {variable} **search scope**:  (referenceNode, linkTarget, enclosingNode, ... )  ->join(ISearchScope | sequence<node<LocalVariableDeclaration>>) {  // find the statement that contains the future local variable ref node<Statement> s = enclosingNode.ancestor<Statement, +>;  // find the first containing ILocalVariableScopeProvider which is // typically next next higher statement that owns a StatementList.  // An example would be a ForStatement or an IfStatement node<ILocalVarScopeProvider> scopeProvider =  enclosingNode.ancestor<ILocalVarScopeProvider, +>;  // In case we are not in a Statement or there  // is no ILocalVarScopeProvider,  // we **return** an empty list - no variables visible **if** (s == **null** || scopeProvider == **null**) { **return new** nlist<LocalVariableDeclaration>;  }  // we now retrieve the position of the current Statement in the  // context StatementList. This is important because we only want to // see those variables that are defined before the reference site **int** pos = s != scopeProvider ? s.index : LocalVarScope.NO\_POSITION;  // finally we query the scopeProvider **for** the visible local variables scopeProvider.getLocalVarScope(s, pos).getVisibleLocalVars();  } |

*Polymorphic References* We have explained above how references work in principle: they are real pointers to the referenced element, based on the target’s unique ID. In the section on Xtext we have seen how from a given location only one kind of reference for any given syntactic form can be implemented. Consider the following example, where we refer to a global variable **a** and an event parameter (**timestamp**) from within the guard condition expression:

|  |
| --- |
| **int** a; **int** b;  **statemachine** linefollower { **in event** initialized(**int** timestamp); **states** (**initial**=initializing) { **state** initializing {  **on** initialized [now() - timestamp > 1000 && a > 3] -> running |

|  |
| --- |
| } **state** running {  }  }  } |

Both references to local variables and to event parameters use the same syntactic form: a text string that represents the name of the respective target element. As we have discussed above, in Xtext, this is implemented with a single reference concept, typically called **SymbolReference**, that can reference to any kind of **Symbol**. **LocalVariableDeclaration** and **EventParameter** would both extend **Symbol**, and scopes would make sure both kinds are visible from within guard expressions35.

In MPS this is done differently. To solve the example above, one would create a **LocalVariableReference** and an **EventParameterReference**. The former references variables and the latter references event parameters. Both have an editor that renders the name of the referenced element, and each of them has *their own* scope definition36. The following is the respective code for the **EventParameterReference** expression:

|  |
| --- |
| **concept** EventParameterReference **extends** Expression  **link** {parameter} **search scope**:  (referenceNode, linkTarget, enclosingNode, ...)  ->join(ISearchScope | sequence<node<EventArg>>) { enclosingNode.ancestor<Transition, +>.trigger.event.args; } |

Entering the reference happens by typing the name of the referenced element (cf. the concept of smart references introduced

above). In the case in which there are a **LocalVariableDeclaration** and an **EventParameter** of the same name, the user has to make an explicit decision, at the time of entry (the name won’t bind, and the code completion menu requires a choice). It is important to understand that, although the names are similar, the tool still knows whether a particular reference refers to a **LocalVariableDeclaration** or to an **EventParameter**, because the reference is encoded using the ID of the target[[8]](#footnote-8).

1. [↑](#footnote-ref-1)
2. . [↑](#footnote-ref-2)
3. . [↑](#footnote-ref-3)
4. . [↑](#footnote-ref-4)
5. . [↑](#footnote-ref-5)
6. . [↑](#footnote-ref-6)
7. e [↑](#footnote-ref-7)
8. . [↑](#footnote-ref-8)