*13*

## IDE Services

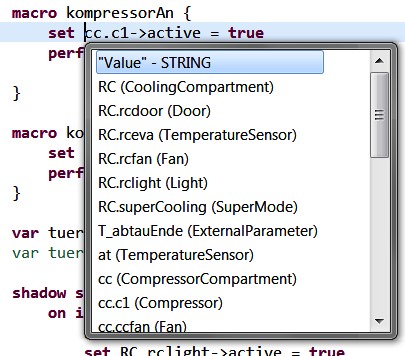
*In this chapter we discuss various services provided by the IDE. This includes code completion, syntax coloring, prettyprinting, go-to-definition and find references, refactoring, outline views, folding, diff and merge, tooltips and visualization. In contrast to the other chapters, this one is not organized by tool, but rather by IDE service. We then provide examples for each service with one or more of the tools. Note that debugging debugging is discussed in Chapter 15.*

In this chapter we illustrate typical services provided by the IDE that are *not* automatically derived from the language definition itself and for which additional configuration or programming is required. Note that we are not going to show every service with every example tool1.

### 13.1 Code Completion

Code completion is perhaps the most essential service provided by an IDE. We already saw that code completion is implicitly influenced by scopes: if you press **Ctrl-Space** at the location of a reference, the IDE will show you the valid targets of this reference (as defined in the scope) in the code completion menu. Selecting one establishes the reference.

*Customizing Code Completion for a Reference in Xtext* Consider the cooling DSL. Cooling programs can reference symbols. Symbols can be hardware building blocks, local variables or configuration parameters. It would be useful in the code completion menu to show what kind of symbol a particular symbol is (Fig. 13.1).



To customize code completion, you have to implement a method in the **ProposalProvider** for your language. The method name has to correspond to a rule/property whose code completion menu you want to customize2. In this example, we want to customize the **symbol** property of the **Atomic** expression:

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| The method takes various arguments; the first one, **model**, represents the program element for which the **symbol** property should be completed.   |  | | --- | | **public class** CoolingLanguageProposalProvider **extends** AbstractCoolingLanguageProposalProvider {  @Override  **public void** completeAtomic\_Symbol(EObject model, Assignment assignment,  ContentAssistContext context,  ICompletionProposalAcceptor acceptor) { ...  }  } | | Let us now look at the actual implementation of the method3. |  | | In line three we get the scope for the particular reference so we can iterate over all the elements and change their appearance in the code completion menu. To be able to get the scope, we need the **EReference** for the particular reference. The first two lines in this method are used to this end. |  |  |  | | --- | | CrossReference crossReference = ((CrossReference)assignment.getTerminal());  EReference ref = GrammarUtil.getReference(crossReference);  IScope scope = getScopeProvider().getScope(model, ref);  Iterable<IEObjectDescription> candidates = scope.getAllElements(); **for** (IEObjectDescription od: candidates) {  String ccText = od.getName()+" ("+od.getEClass().getName()+")"; String ccInsert = od.getName().toString(); acceptor.accept(createCompletionProposal(ccInsert, ccText, **null**, context)); } |   f. |

2 This also works for containment references or primitive properties and not just for references (which are affected by scopes).

Once we have the scope, we can iterate over all its contents

(i.e. the target elements)4. Inside the loop we then use the

name of the target object plus the name of the **EClass** to construct the string to be shown in the code completion menu (**ccText**)[[1]](#footnote-1). The last line then calls the **accept** method on the **ICompletionProposalAcceptor** to finally create a proposal. Note how we also pass in **ccInsert**, which is the text to be inserted into the program if the particular code completion menu item is selected.

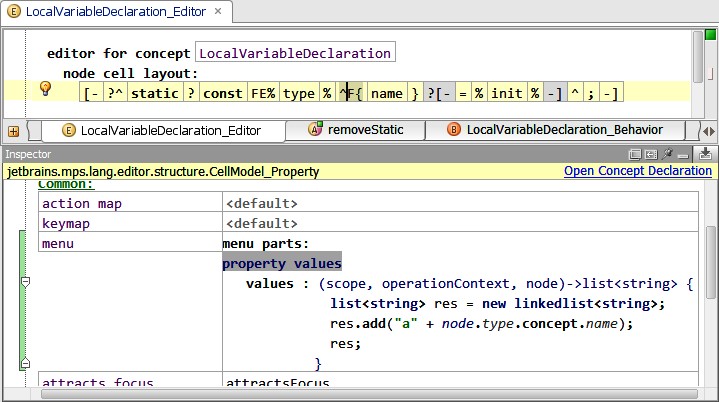
*An Example with MPS* The contents of the code completion menu for references can be customized in MPS as well. It is

instructive to look at this in addition to Xtext for two reasons. The first one is brevity. Consider the following code, where we customize the code completion menu for function calls:

|  |
| --- |
| **link** {function} ...  **presentation** :  (parameterNode, visible, smartReference, inEditor, ...)->**string** { parameterNode.signatureInfo();  } |

|  |  |
| --- | --- |
| we call a method that calculates a string that represents the complete signature of the function.  The second reason why this is interesting in MPS is that we don’t have to specify the text that should be inserted if an element is selected from the code completion menu: the reference is established based on the UUID of the target node, and the editor of the referencing node determines the presentation of | here. |
| this reference7. |  |
| *Code Completion for Simple Properties* In Xtext, code completion can be provided for any property of a rule, not just for references (i.e. also for children or for primitive properties such as strings or numbers). The mechanism to do that is the same as the one shown above. Instead of using the scope (only references have scopes) one could use a statically populated list of strings as the set of proposals, or one could query a database |  |
| to get a list of candidate values8. |  |
| In MPS, the mechanism is different. Since this is a pure editor customization and has nothing to do with scopes, this behavior is customized in the editor definition. Consider a  **LocalVariableDeclaration** (as in **int x = 0;**) where we want to customize the suggested name of the variable. So if you press **Ctrl-Space** in the name field of the variable, we want to suggest one or more reasonable names for the variable. Fig. 13.2 shows the necessary code.  An editor cell may have a cell menu (the menu you see when you press **Ctrl-Space**). It consists of several parts. Each part contributes a set of menu entries. In the example in Fig. 13.2, we add a cell menu part of type **property values**, in which we simply return a list of values (one, in the example; we use the name of the type of the local variable, prefixed by an **a**). |  |

To customize the contents of the code completion menu, you have to provide the expression that calculates the text in the **presentation** section of the scope provider6. In this example



*Editor Templates* Templates are more complex syntactic structures that can be selected from the code completion menu. For example, the code completion menu may contain an **if-thenelse** entry, which, if you select it, gets expanded into the following code in the program:

**if** ( expr ) { } **else** {

}

Xtext provides templates for this purpose. These can be defined either as part of the language, or by the user in the IDE.

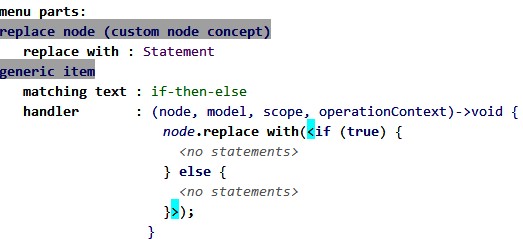
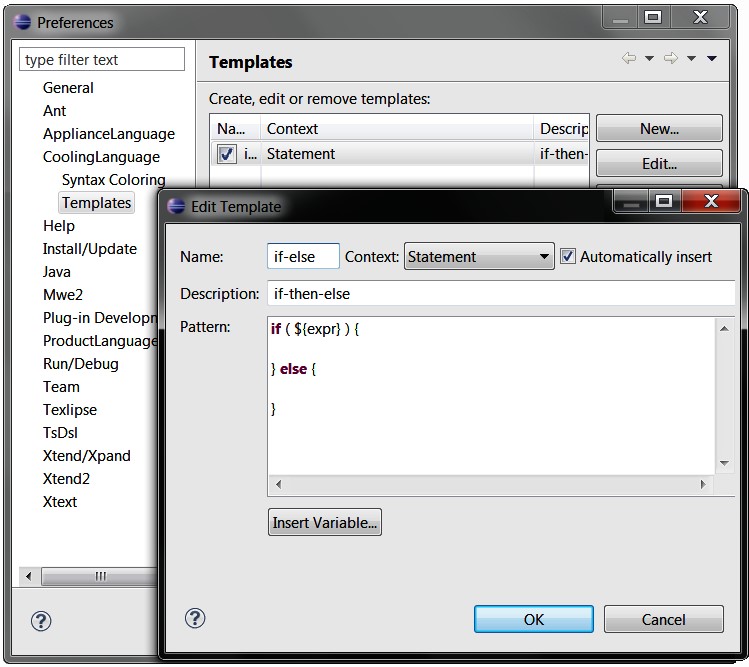
Fig. 13.3 shows the **if-then-else** example as defined in the IDE.

In MPS there are several ways to address this. One is simply an intention (explained in more detail in Section 7.7). It will not be activated via **Ctrl-Space**, but rather via **Alt-Enter**. In every other respect it is identical: the intention can insert arbitrary code into the program. Alternatively we can use a cell menu (already mentioned above). Fig. 13.4 shows the code for a cell menu that also creates the **if-then-else** structure illustrated above.

### 13.2 Syntax Coloring

There are two cases for syntax coloring: syntactic highlighting and semantic highlighting. Syntactic highlighting is used to color keywords, for example. These keywords are readily

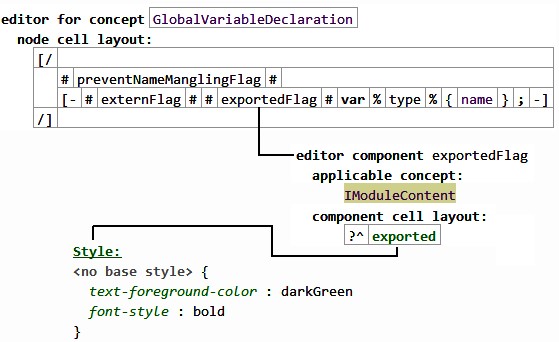
F



available from the grammar. No customization is necessary beyond configuring the actual color. Semantic coloring colors code fragments based on some query over the AST structure. For example, in a state machine, if a state is unreachable (no incoming transitions) the state may be colored in gray instead of black.

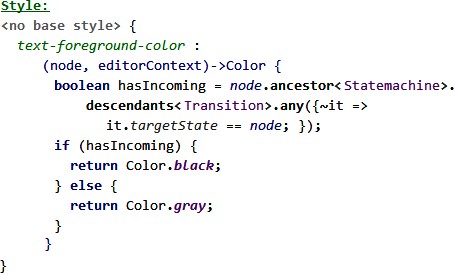
*An Example with MPS* Let us first look at syntax coloring in MPS, starting with purely syntactic highlighting. Fig. 13.5 shows a collage of several ingredients: at the top we see the editor for **GlobalVariableDeclaration**. **GlobalVariableDeclaration** implements the interface **IModuleContent**. **IModuleContent**s can be exported (which means they can be seen by

.



|  |  |
| --- | --- |
| modules importing the current one), so we define an editor component (a reusable editor fragment) for **IModuleContent** that renders the **exported** flag. This editor component is embedded into the editor of **GlobalVariableDeclaration**, and is also embedded into the editor of all other concepts that implement **IModuleCon- tent**. The editor component simply defines a keyword **exported** that is rendered in dark green and in bold font. This can be achieved by specifying the respective |  |
| style properties for the editor cell9. |  |
| Semantic highlighting works essentially the same way. Instead of using a constant (**darkGreen**) for the color, we embed a query expression. The code in Fig. 13.6 renders the **state** keyword of a **State** in a *Statemachine* gray if that particular state has no incoming transitions. |  |

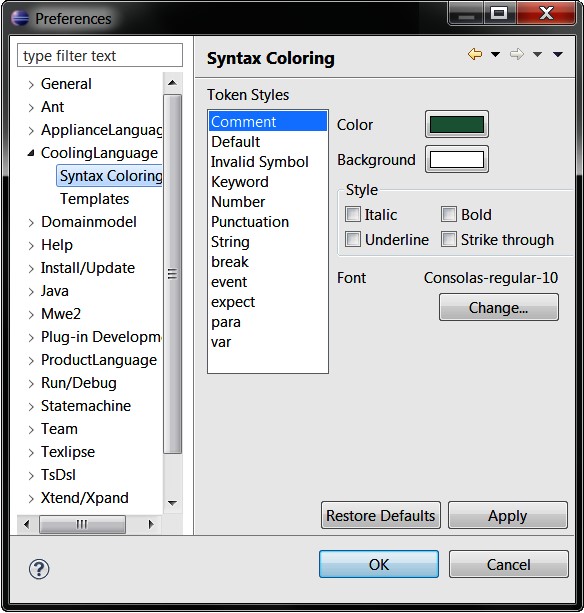
.



*An Example with Xtext* Xtext uses a two-phase approach. First, you have to define the styles you want to apply to parts of the text. This is done in the highlighting configuration of the particular language:

|  |
| --- |
| **public class** CLHighlightingConfiguration **extends** DefaultHighlightingConfiguration { **public static final** String VAR = "var";  @Override  **public void** configure(IHighlightingConfigurationAcceptor acceptor) { **super**.configure(acceptor); acceptor.acceptDefaultHighlighting(VAR, "variables", varTextStyle());  }  **private** TextStyle varTextStyle() {  TextStyle t = defaultTextStyle().copy(); t.setColor(**new** RGB(100,100,200));  t.setStyle(SWT.ITALIC | SWT.BOLD ); **return** t;  }  } |

The **varTextStyle** method creates a **TextStyle** object. The method **configure** then registers this style with the framework using a unique identifier (the constant **VAR**). The reason for registering it with the framework is that the styles can be changed by the user in the running application using the preferences dialog (Fig. 13.7).



We now have to associate the style with program syntax10. The semantic highlighting calculator for the target language is used to this end11. It requires the **provideHighlightingFor** method to be implemented. To highlight references to variables (not the variables themselves!) with the style defined above works the following way:

|  |
| --- |
| **public void** provideHighlightingFor(XtextResource resource,  IHighlightedPositionAcceptor acceptor) {  EObject root = resource.getContents().get(0);  TreeIterator<EObject> eAllContents = root.eAllContents(); **while** (eAllContents.hasNext()) {  EObject ref = (EObject) eAllContents.next(); **if** ( ref **instanceof** SymbolRef ) {  SymbolDeclaration sym = ((SymbolRef) o).getSymbol(); **if** ( sym **instanceof** Variable ) {  ICompositeNode n = NodeModelUtils.findActualNodeFor(ref); acceptor.addPosition(n.getOffset(), n.getLength(), CLHighlightingConfiguration.VAR);  }  }  }  } |

The method gets passed in an **XtextResource**, which represents a model file. From it we get the root element and iterate over all its contents. If we find a **SymbolRef**, we continue with coloring. Notice that in the cooling language we reference *any*

|  |  |
| --- | --- |
| variable, we now have to move from the abstract syntax tree (on which we have worked all the time so far) to the concrete syntax tree, so we can identify particular tokens that shall be | incoming transitions) as well. |
| colored13. We use a utility method to find the **ICompositeNode** |  |
| that represents the **SymbolRef** in the concrete syntax tree. Finally we use the **acceptor** to perform the actual highlighting using the position of the text string in the text. We pass in the |  |
| **VAR** style defined before14. |  |

symbol (variable, event, hardware element) with a **SymbolRef**, so we now have to check whether we reference a **Variable** or not12. If we have successfully identified a reference to a

*An Example with Spoofax* Spoofax supports syntax coloring on the lexical and the syntactic level. At the lexical level, tokens such as keywords, identifiers, or integers are colored. This is the most common use case of syntax coloring. At the syntactic level, we can color larger code fragments, for example to highlight embeddings. In Spoofax, syntax coloring is specified declaratively as part of the editor specification. For the lexical level, Spoofax predefines the token classes **keyword**, **identifier**, **string**, **number**, **var**, **operator** and **layout**. For each of these, we can specify a color (either by name or by RGB values) and optionally a font style (**bold**, **italic**, or both). Spoofax generates the following default specification:

|  |
| --- |
| **module** MoblLang-Colorer.generated **colorer** Default, **token**-**based highlighting**  keyword : 127 0 85 bold identifier : default string : blue number : darkgreen var : 255 0 100 italic operator : 0 0 128 layout : 63 127 95 italic  **colorer** System colors  darkgreen = 0 128 0 green = 0 255 0 darkblue = 0 0 128 blue = 0 0 255  ...  default = \_ |

The generated specification can be customized on the lexical level, but also extended on the syntactic level. These extensions are based on syntactic sorts and constructor names. For example, the following specification will color numeric types in declarations in dark green:

|  |
| --- |
| **module** DSLbook-Colorer **imports** DSLbook-Colorer.generated **colorer**  Type.NumType: darkgreen |

|  |
| --- |
| Here **Type** is a sort from the syntax definition, while **NumType** is the constructor for the integer type. There are other rules for **Type** in the Mobl grammar, for example for the string type. When we want other types also to be colored dark green, we can either add more rules to the colorer specification, or replace the current definition with **Type.\_**, where **\_** acts as a wildcard and all types will be colored dark green, independent of their constructor. Similarly, we can use a wildcard for sorts. For example, **\_.NumType** will include all nodes with a constructor **NumType**, independent of their syntactic sort.  In the current example, predefined types like **int** and entity types are all colored dark green, but only the predefined types will appear in bold face. This is because Spoofax combines specified colors and fonts. The rule on the syntactic level specifies only a color, but no font. Since the predefined types are keywords, they will get the font from the keyword specification, which is bold. In contrast, entity types are identifiers, which will get the default font from the identifier specification.  *13.3 Go-to-Definition and Find References*  Following a reference (go to definition, **Ctrl-Click**) as well as finding references to a given program element works automatically without any customization in any of the language workbenches. However, one might want to change the default behavior, for example because the underlying program element is not a reference at all (but you still want to go somewhere when **Ctrl-Click**ing on it).  *Customizing the Target with Xtext* Let us first look at how to change the target of the go-to-definition functionality. Strictly speaking, we don’t change go-to-definition at all. We just define a new hyperlinking functionality. Go-to-Definition is just the default hyperlinking behavior[[2]](#footnote-2). As a consequence: |

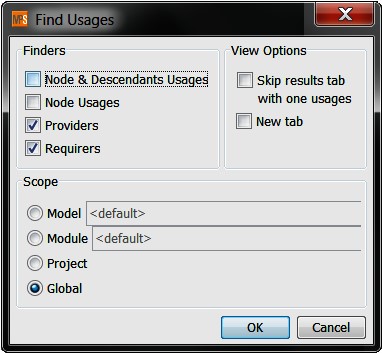
* You can define hyperlinking for elements that are *not* references in terms of the grammar (a hyperlink can be provided for any program element).
* You can have several hyperlinks for the same element. If you **Ctrl-Hover** on it, a little menu opens up and you can select the target you are interested in.

To add hyperlinks to a language concept, Xtext provides the **IHyperlinkHelper** interface, which can be implemented by language developers to customize hyperlinking behavior. It requires one method, **createHyperlinksTo**, to be implemented16.

A typical implementation looks as follows:

|  |
| --- |
| **public void** createHyperlinksTo(XtextResource from, Region region,  EObject to, IHyperlinkAcceptor acceptor) {  **if** ( to **instanceof** TheEConceptIAmInterestedIn ) { EObject target = // find the target of the hyperlink **super**.createHyperlinksTo(from, region, target, acceptor);  } **else** {  **super**.createHyperlinksTo(from, region, to, acceptor);  }  } |

.

 *Customized Finders in MPS* In many cases, there are different kinds of references to any given element. For example, for an **Interface** in the mbeddr C components extension, references to that interface can either be sub-interfaces (**ISomething extends IAnother**) or components, which can either *provide* an interface (so other components can call the interface’s operation), or they can *require* an interface, in which case the component itself calls operations defined by the interface. When finding references, we may want to distinguish between these different cases.

|  |  |
| --- | --- |
| MPS provides *finders* to achieve this.. |  |

|  |
| --- |
| **finder** findProviders **for concept** Interface **description**: Providers  find(node, scope)->**void** {  nlist<> refs = **execute** NodeUsages ( node , <same scope> ); **foreach** r **in** refs.select(it|it.isInstanceOf(ProvidedPort)) {  **add result** r.parent ;  } }  getCategory(node)->**string** { "Providers";  } |

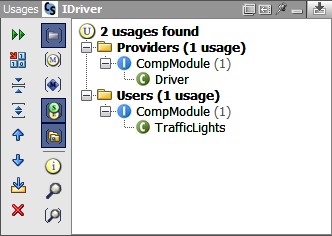
Implementing finders is simple, since, as usual, MPS provides a DSL for specifying them. The following code shows the implementation.

Figure 13.9: The result dialog of running Find Usages with our customized finders. Note the **Providers** and **Users** categories; these correspond to the strings returned from **getCategory** in the two finders.

We specify a name for the finder (**findProviders**) as well as the type to which it applies (references to which it will find: **Interface** in the example). We then have to implement the **find** method. Notice how in the first line of the implementation we delegate to an existing finder, **Node Usages**, which finds *all* references. We then check whether the referencing element is a **ProvidedPort**, and if so, we add the parent of the port, i.e. a **Component**, to the result17. Finally, **getCategory** re-

turns a string that is use to structure the result. Fig. 13.9 shows an example result.

*Customizing the Target with Spoofax* Spoofax provides a default hyperlinking mechanism from references to declarations. Alternative hyperlinking functionality can be implemented in rewrite rules. The names of these rules need to be specified in the editor specification. For example, the following specification tells Spoofax to use a custom rewrite rule to hyperlink **this** expressions to the surrounding class:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| references  reference Exp.This : resolve-this   |  |  |  |  |  | | --- | --- | --- | --- | --- | | On the left-hand side of the colon the **reference** rule specifies a syntactic sort and a constructor, for which the hyperlinking | | |  | | | should be customized18. On the right-hand side of the colon, | | |  | | | the rule names a rewrite rule which implements the hyperlinking: | | |  | | | resolve-this:  (link, position, ast, path, project-path) -> target **where**  Entity(t) := <type-of> link ; target := <index-lookup> t | | This rule determines the type of a *this* expression and links it | | |  | | to the declaration of this type19. | | |  | |

### 13.4 Pretty-Printing

Pretty-printing refers to the reverse activity from parsing[[3]](#footnote-3). A parser transforms a character sequence into an abstract syntax tree. A pretty printer (re-)creates the text string from the AST. As the term *pretty* printing suggests, the resulting text should be *pretty*, i.e. whitespace must be managed properly.

So when and where is a formatter useful? There is the obvious use case: users somehow mess up formatting, and they want to press **Ctrl-Shift-F** to clean it up. However, there is

more essential reason. If the AST is modified by a transformation, the updated text has to be rendered correctly. An AST is modified, for example, as part of a quick fix (see the next paragraph) or by a graphical editor that operates in parallel to a text editor on the same AST.

|  |  |
| --- | --- |
| of several different editors for a single concept. They may be fundamentally different (e.g., providing a textual and a graphical syntax for state machines) or just provide different "layouts" for a single notation (different positions of the opening curly brace, for example). More generally, there is no reason why a projectional editor may not provide a certain degree of freedom regarding layout. Users may be able to press **ENTER** to start a new line in a long expression, or press **TAB** to indent a |  |
| statement22. However, MPS does currently not support this. |  |
| *Pretty-Printing in Spoofax* Spoofax generates a languagespecific rewrite rule **pp-<LanguageName>-string** which rewrites an abstract syntax tree into a string according to a pretty-printer definition (expressed in the Box language). Spoofax generates a default pretty-printer definition from the syntax definition of a language. For example, Spoofax generates the following pretty-printer definition for Mobl: |  |

*Pretty-Printing in MPS* In MPS, pretty-printing is a nonissue. The editor always pretty-prints as part of the projection21. However, version 3.0 of MPS will support the definition

|  |
| --- |
| [  Module -- **KW**["module"] \_1 \_2,  Module.2:iter-star -- \_1,  Import -- **KW**["import"] \_1,  Entity -- **KW**["entity"] \_1 **KW**["{"] \_2 **KW**["}"],  Entity.2:iter-star -- \_1,  Property -- \_1 **KW**[":"] \_2,  Function -- **KW**["function"] \_1 **KW**["("] \_2 **KW**[")"]  **KW**[":"] \_3 **KW**["{"] \_4 **KW**["}"],  Function.2:iter-star-sep -- \_1 **KW**[","], Function.4:iter-star -- \_1,  Param -- \_1 **KW**[":"] \_2, EntType -- \_1,  NumType -- **KW**["int"],  BoolType -- **KW**["boolean"],  StringType -- **KW**["string"],  Declare -- **KW**["var"] \_1 **KW**["="] \_2 **KW**[";"],  Assign -- \_1 **KW**["="] \_2 **KW**[";"],  Return -- **KW**["return"] \_1 **KW**[";"],  Call -- \_1 **KW**["."] \_2 **KW**["("] \_3 **KW**[")"],  PropAccess -- \_1 **KW**["."] \_2,  Plus -- \_1 **KW**["+"] \_2, Mul -- \_1 **KW**["\*"] \_2,  Var -- \_1,  Int -- \_1  ] |

In the Box language, rules consist of constructors (i.e. AS elements or language concepts) on the left-hand side of a rule and a sequence of *boxes* and numbers on the right-hand side. The basic box construct is a simple string, representing a string in the output. Furthermore, two kinds of box operators can be applied to sub-boxes: layout operators specify the layout of sub-boxes in the surrounding box, and font operators specify which font should be used. In the example, all strings are embedded in **KW[...]** boxes. **KW** is a font operator, classifying the sub-boxes as keywords of the language23.

Numbers on the right-hand side can be used to combine boxes from the subtrees: a number *n* refers to the boxes from the *n*-th subtree. When the syntax definition contains nested constructs, additional rules are generated for pretty-printing the corresponding subtrees. On the left-hand side, these rules have *selectors*, which consist of a constructor, a number selecting a particular subtree, and the type of the nesting. The following table shows all nesting constructs in syntax definitions and their corresponding types in pretty-printing rules.

|  |  |
| --- | --- |
| **Construct** | **Selector Type** |
| optionals **S?** | **opt** |
| non-empty lists **S+** | **iter** |
| possibly empty lists **S\*** | **iter-star** |
| separated lists **S1 S2+** | **iter-sep** |
| possibly empty separated lists **S1 S2\*** | **iter-star-sep** |
| alternatives **S1 | S2** | **alt** |
| sequences **(S1 S2)** | **seq** |

Additionally, user-defined pretty-printing rules can be defined as well. Spoofax first applies the user-defined rules to turn an abstract syntax tree into a hybrid tree which is only partially pretty-printed. It then applies the default rules to pretty-print the remaining parts. For example, we could define our own pretty-printing rule for Mobl modules:

|  |  |
| --- | --- |
| Module | -- **V vs**=1 **is**=4 [ **H** [**KW**["module"] \_1] \_2] |

This rule lets Spoofax pretty-print the term **Module("shopping",**

**[Entity(...), Entity(...)])** as

**module** shopping **entity** ... **entity** ...

The **V** box operator places sub-boxes vertically. In the example, it places the entities underneath the **module shopping** line. The desired vertical separation between the sub-boxes can be specified by the spacing option **vs**. Its default value is **0**; that is, no blank lines are added between the boxes. In the example, a blank line is enforced by **vs=1**. For indenting boxes in a vertical combination, the spacing option **is** can be specified. All boxes except the first will be indented accordingly. In the example, the **module shopping** line is unindented, while the entities are indented by 4 spaces. The **H** box operator lays out sub-boxes horizontally. In the example, it is used to lay out the **module** keyword and its name in the same line. The desired horizontal separation between the sub-boxes can be specified by the spacing option **hs**. Its default value is **1**; that is, a single space is added between the boxes.

*Pretty-Printing in Xtext* In Xtext, the use of whitespace can be specified in a language’s **Formatter**. Formatters use a Java API to specify whitespace policies for a grammar. Consider an example from the cooling language. Assume we enter the following code:

|  |  |
| --- | --- |
| **state** Hello | : **entry** { **if true** { } } |

If we run the formatter (e.g., by pressing **Ctrl-Shift-F** in the IDE), we want the resulting text to be formatted like this:

|  |
| --- |
| **state** Hello:  **entry** { **if true** { }  } |

The following formatter code implements this.

|  |
| --- |
| **protected void** configureFormatting(FormattingConfig c) {  CoolingLanguageGrammarAccess f =  (CoolingLanguageGrammarAccess) getGrammarAccess();  c.setNoSpace().before(  f.getCustomStateAccess().getColonKeyword\_3());  c.setIndentationIncrement().after(  f.getCustomStateAccess().getColonKeyword\_3());  c.setLinewrap().before(  f.getCustomStateAccess().getEntryKeyword\_5\_0());  c.setLinewrap().after(  f.getCustomStateAccess().getLeftCurlyBracketKeyword\_5\_1());  c.setIndentationIncrement().after(  f.getCustomStateAccess().getLeftCurlyBracketKeyword\_5\_1()); c.setLinewrap().before(  f.getCustomStateAccess().getRightCurlyBracketKeyword\_5\_3());  c.setIndentationDecrement().before(  f.getCustomStateAccess().getRightCurlyBracketKeyword\_5\_3());  } |

In the first line we get the **CoolingLanguageGrammarAccess** object, an API to refer to the grammar of the language itself. This API is the basis for an internal Java DSL for expressing formatting rules. Let’s look at the first block of three lines. In the first line we express that there should be no space before the colon in the **CustomState** rule. Line two states that we want to have indentation after the colon. The third line specifies that the **entry** keyword should be on a new line. The next two blocks of two lines manage the indentation of the entry action code. In the first block we express a line wrap and incremented indentation after the opening curly brace. The second block expresses a wrap before the closing curly brace, as well as a decrement in the indentation level24.

|  |
| --- |
| **public static final** String VARIABLE\_LOWER\_CASE = "VARIABLE\_LOWER\_CASE";  @Check  **public void** checkVariable( Variable v ) { **if** ( !Character.isLowerCase( v.getName().charAt(0) ) ) { warning("Variable name should start with a lower case letter", al.getSymbolDeclaration\_Name(), VARIABLE\_LOWER\_CASE );  }  } |

|  |  |
| --- | --- |
|  |  |
| *13.5 Quick Fixes*  A quick fix is a semi-automatic fix for a constraint violation. It is semi-automatic in the sense that it is made available to the user in a menu, and after selecting the respective quick fix from the menu, the code that implements the quick fix rectifies the |  |
| problem that caused the constraint violation25. |  |
| *Quick Fixes in Xtext* Xtext supports quick fixes for constraint violations. Quick fixes can either be implemented on the concrete syntax (i.e. via text replacement) or on the abstract syntax (i.e. via a model modification and subsequent serialization). As an example, consider the following constraint defined in the cooling language’s **CoolingLanguageJavaValidator**: |  |

|  |  |
| --- | --- |
| Based on our discussion of constraint checks (Section 9.1), this code should be fairly self-explanatory. What is interesting is the third argument to the **warning** method: we pass in a constant to uniquely identify the problem. The quick fix will be tied to this constant. The following code is the quick fix, imple- |  |
| mented in the **CoolingLanguageQuickfixProvider**26. Notice |  |

how in the **@Fix** annotation we refer to the same constant that was used in the constraint check.

|  |
| --- |
| @Fix(CoolingLanguageJavaValidator.VARIABLE\_LOWER\_CASE) **public void** capitalizeName(**final** Issue issue,  IssueResolutionAcceptor acceptor) {  acceptor.accept(issue, "Decapitalize name",  "Decapitalize the name.", "upcase.png",  **new** IModification() { **public void** apply(IModificationContext context) **throws** BadLocationException {  IXtextDocument xtextDocument = context.getXtextDocument(); String firstLetter = xtextDocument.get(issue.getOffset(), 1); xtextDocument.replace( issue.getOffset(), 1, firstLetter.toLowerCase());  }  });  } |

Quick fix methods accept the **Issue** that caused the problem as well as an **IssueResolutionAcceptor** that is used to register the fixes so they can be shown in the quick fix menu. The core of the fix is the anonymous instance of **IModification** that, when executed after it has been selected by the user, fixes the problem. In our example, we grab the document that contains the problem and use a text replacement API to replace the first letter of the offending variable with its lower case version.

Working on the concrete syntax level is ok for simple problems like this one. More complex problems should be solved on the abstract syntax though27. For these cases, one can use

an instance of **ISemanticModification** instead:

|  |
| --- |
| @Fix(CoolingLanguageJavaValidator.VARIABLE\_LOWER\_CASE)  **public void** fixName(**final** Issue issue, IssueResolutionAcceptor acceptor) { acceptor.accept(issue, "Decapitalize name",  "Decapitalize the name", "upcase.png",  **new** ISemanticModification() { **public void** apply(EObject element, IModificationContext context) {  ((Variable) element).setName(  Strings.toFirstLower(issue.getData()[0]));  }  });  } |

|  |  |
| --- | --- |
| A quick fix using an **ISemanticModification** basically works the same way; however, inside the **apply** method we now use |  |
| the EMF Java API to fix the problem28. |  |
| *Quick Fixes in MPS* Quick fixes in MPS work essentially the same way as in Xtext. Of course there are only quick fixes that act on the abstract syntax – the concrete syntax is projected in any case. Here is a constraint that checks that the name of an element that implements **INameAllUpperCase** actually consists of only upper case letters: |  |

**checking rule** check\_INameAllUpperCase {

**applicable for concept** = INameAllUpperCase **as** a

The quick fix below upper-cases the name if necessary. The quick fix is associated with the constraint check by simply referencing the fix from the error message. Quick fixes are exe- 29

cuted by selecting them from the intentions menu (**Alt-Enter**)29.

|  |
| --- |
| **quick fix** fixAllUpperCase  **arguments**: node<IIdentifierNamedConcept> node  **description**(node)->**string** { "Fix name";  }  **execute**(node)->**void** { node.name = node.name.toUpperCase(); } |

|  |
| --- |
| **do** {  **if** (!(a.name.equals(a.name.toUpperCase()))) {  warning "name should be all upper case" -> a; }  }  } |

*Model Synchronization via Quick Fixes* A particularly interesting feature of MPS’ quick fixes is that they can be executed *automatically* in the editor. This can be used for synchronizing different parts of a model: a constraint check detects an inconsistency in the model, and the automatically executed quick fix resolves the inconsistency.

Here is an example where this makes sense. Consider the interfaces and components extension to C. An interface declares a couple of operations, each with their own unique signature. A component that **provides** the interface has to provide implementations for each of the operations, and the implementations must have the same signature as the operation it implements. A constraint checks the consistency between interfaces and implementing components. An automatically executed quick fix adds missing (empty) operation implementations and synchronizes their signatures with the signatures of the operations in the interface.

### 13.6 Refactoring

Refactoring addresses changing the program structure without changing its behavior. It is typically used to "clean up" the program structure after it has gotten messy over time. While DSLs

and their programs tend to be simpler than GPL programs, refactorings are still useful.

*Renaming in Xtext* One of the most essential refactorings is renaming a program element30. Xtext comes with rename

refactoring out of the box, every language supports rename refactoring automatically31. The only thing the user has to remember is to not just type a new name, but instead invoke the Rename refactoring, for example via **Ctrl-Alt-R**.

*Renaming in Spoofax* Like code generators, refactorings need to be specified in the editor specification and implemented with rewrite rules. For example, the following specification specifies a refactoring for renaming entities:

|  |
| --- |
|  |

The specification starts with a syntactic sort and a constructor, on which the refactoring should be available, followed by a label for the refactoring in the context menu, the implementing rewrite rule, and two options. In the example, Spoofax is instructed to use the current **cursor** position to determine the node on which the refactoring should be applied. The specification further defines a **shortcut** for the refactoring, which should be the same key binding as the one used in the JDT for renaming. Finally, it defines an interactive **input** dialog, with a label **"new name"** and an empty default input. The refactoring itself is implemented in a rewrite rule:

|  |
| --- |
| rename-entity:  (newname, Entity(name, elems), position, ast, path, project-path)  -> ([(ast, new-ast)], errors, [], [])  **with** new-ast := <topdown(try(rename-entity-local(|name, newname)))> ast;  [Entity(), oldname|path] := <index-uri> name; **if** <index-lookup> [Entity(), newname|path] **then** errors := [(name, $[Entity of name [newname] already exists.])]  **else** errors := []  **end**  rename-entity-local(|old-name, new-name):  Entity(old-name, elems) -> Entity(new-name, elems)  rename-entity-local(|old-name, new-name):  EntType(old-name) -> EntType(new-name) |

As we have seen already for other editor services, rewrite rules for refactorings have to adhere to a specific interface (i.e. signature). On the left-hand side, the example rule matches a tuple

consisting of the input from the refactoring dialog (**newname**), the node on which the refactoring is applied, its **position** in the abstract syntax tree, the tree itself (**ast**), and the paths of the current file and the project. On the right-hand side, it yields a tuple consisting of a list of changes in the abstract syntax tree and lists of fatal errors, normal errors and warnings.

For simplicity, the example rule changes the whole abstract syntax tree into a new one and provides only duplicate definition errors. To do so, the new abstract syntax tree is retrieved by traversing the old one in a **topdown** fashion, **try**ing to apply rewrite rules **rename-entity-local**32. These rules

|  |  |  |
| --- | --- | --- |
| The first rule rewrites entity declarations, while the second one rewrites types of the form **EntType(name)**, where **name** refers to an entity.  An error is detected if an entity with the new name already exists. Therefore, we match the annotated URI of the old name, change it to the new name, and look it up. If we find an entity, the renamed entity would clash with the one just found.  *Introduce Local Variable in MPS* A very typical refactoring for a procedural language such as C is to introduce a new local variable. Consider the following code:   |  | | --- | | int8 someFunction(int8 v) { int8 y = somethingElse(v \* FACTOR); **if** ( v \* FACTOR > 20 ) { **return** 1;  } **else** { **return** 0;  }  } |   As you can see, the first two lines contain the same expression (**v \* FACTOR**) twice. A nicer version of this code might look like this:   |  | | --- | | int8 someFunction(int8 v) {  int8 product = v \* FACTOR; int8 y = somethingElse(product); **if** ( product > 20 ) { **return** 1;  } **else** { **return** 0;  }  } |   The *Introduce Local Variable* refactoring performs this change. MPS provides a DSL for refactorings, based on which the implementation is about 20 lines of code. We’ll go through it in steps[[4]](#footnote-4). We start with the declaration of the refactoring itself. |

take the old and new entity name as parameters. They ensure that declarations and references to entities are renamed. succeeds, it will.

|  |
| --- |
| **refactoring** introduceLocalVariable ( "Introduce Local Variable" )  **keystroke**: <ctrl+alt>+<V> **target**: node<Expression> **allow multiple**: **false**  **isApplicableToNode**(node)->**boolean** { node.ancestor<Statement>.isNotNull;  } |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| find out about that by checking whether the **Expression** has a **Statement** among its ancestors in the tree. Next, we define a parameter for the refactoring:   |  | | --- | | **parameters**:  varName **chooser**: **type**: **string**  **title**: Name of the **new** Variable  **init**(refactoringContext)->**boolean** { **return ask for** varName; } | | The parameter represents the name of the newly introduced | | |  | | variable. In the refactoring’s **init** block we ask the user for this parameter35. We are now ready to implement the refactor- | | |  | | ing algorithm itself in the **refactor** block. We first declare two local variables that represent the expression on which we in- | | |  | |

The code above specifies the name of the refactoring (**introduceLocalVariable**), the label used in the refactoring menu, the keystroke to execute it directly (**Ctrl-Alt-V**) as well as the target, i.e. the language concept on which the refactoring can be executed. In our case, we want to refactor **Expression**s, but only if these expressions are used in a **Statement**34. We

voked the refactoring (we get it from the **refactoringContext**36) and the **Statement** under which this expression is located. Finally, we get the **index** of the **Statement**[[5]](#footnote-5).

node<Expression> targetExpr = **refactoringContext**.node; node<Statement> targetStmt = targetExpr.ancestor<Statement>; **int** index = targetStmt.index;

Next, we iterate over all **siblings** of the statement in which the expression lives. As we do that, we look for all expressions that are structurally similar to the one we’re executing the refactoring on (using **MatchingUtil.matchNodes**). We remember a matching expression if it occurs in a statement that is *after* the one that contains our target expression.

|  |
| --- |
| nlist<Expression> matchingExpressions = **new** nlist<Expression>; sequence<node<>> siblings = targetStmt.siblings.union(**new** singleton<node<Statement>>(stmt));  **foreach** s **in** siblings {  **if** (s.index >= index) { **foreach** e **in** s.descendants<Expression> { **if** (MatchingUtil.matchNodes(targetExpr, e)) { matchingExpressions.add(e);  } } } } |

.

The next step is to actually introduce the new local variable.

We create a new **LocalVariableDeclaration** using the API. We set the **name** to the one we’ve asked the user for (**varName**), we set its type to a copy of the type calculated by the type system for the target expression, and we initialize the variable with a copy of the target expression itself. We then add this new variable to the list of statements, just *before* the one which contains our target expression. We use the **add prev-sibling** built-in function for that.

node<LocalVariableDeclaration> lvd = **new** node<LocalVariableDeclaration>(); lvd.name = varName; lvd.type = targetExpr.type.copy; lvd.init = targetExpr.copy; targetStmt.add prev-sibling(lvd);

There is one more step we have to do. We have to replace all the occurrences of our target expression with a reference to the newly introduced local variable. We had collected the **matchingExpressions** above, so we can now iterate over this

collection38: 3

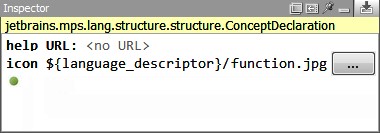
|  |
| --- |
| **foreach** e **in** matchingExpressions { node<LocalVarRef> ref = **new** node<LocalVarRef>(); ref.var = lvd;  e.replace with(ref);  } |

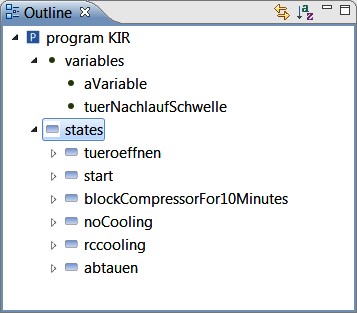
|  |  |
| --- | --- |
| All in all, building refactorings is straightforward with MPS’ refactoring support. The implementation effort is reduced to essentially the algorithmic complexity of the refactoring itself. Depending on the refactoring, this can be non-trivial.  *13.7 Labels and Icons*  Labels and icons for language concepts are used in several places, among them the outline view and the code completion menu.  *Labels and Icons in Xtext* Labels and icons are defined in the language’s **LabelProvider**, which is generated by Xtext for each language by default. To define the label text, you simply override the **text** method for your element, which returns either a **String** or a **StyledString** (which includes formatting information). For the icon, override the **image** method. Here |  |
| are a couple of examples from the cooling language39: | - |

|  |
| --- |
| **public class** CoolingLanguageLabelProvider **extends** DefaultEObjectLabelProvider {  String text(CoolingProgram prg) { |

|  |
| --- |
| **return** "program "+prg.getName();  }  String image(CoolingProgram prg) { **return** "program.png"; }  String text(Variable v) { **return** v.getName()+": "+v.getType();  }  String image(Variable v) { **return** "variable.png";  }  } |

*Labels and Icons in MPS* Labels are defined by overriding the **getPresentation** behavior method on the respective concept. This allows the label to also be adjusted dynamically. The icon can be selected in the inspector (see Fig. 13.10) if we select a language concept. The icon is fixed and cannot be changed dynamically.





### 13.8 Outline

The outline provides an overview over the contents of some part of the overall model, typically a file. By default, it usually shows more or less the AST, down to a specific level (the implementations of functions or methods are typically not shown). The contents of the outline view must be user-definable; at the very least, we have to define where to stop the tree. Also, the tree structure may be completely different from the nesting structure of the AST: the elements may have to be grouped based on their concept (first show all variables, then all functions) or they may have to be sorted alphabetically.

*Customizing the Structure in Xtext* Xtext provides an **OutlineTreeProvider** for your language that can be used to customize the outline view structure (labels and icons are taken from the **LabelProvider** discussed above). As an example, let us customize the outline view for cooling programs to look the one shown in Fig. 13.11.

The tree view organizes the contents of a file by first showing all programs and then all tests. To do this, we provide a suitable implementation of **\_createChildren**:

|  |
| --- |
| **protected void** \_createChildren(DocumentRootNode parentNode, Model m) { **for** (EObject prg : m.getCoolingPrograms()) { createNode(parentNode, prg);  } **for** (EObject t : m.getTests()) { |

Inside the method, we first grab all the **CoolingProgram**s from the root element **Model** and create a node for them using the **createNote** API, which takes the parent (in terms of the outline view) and the program element for which should be represented by the new outline node40. We then do the same for

tests.

Inside a program, we want to show variables and states in separate sections, i.e. under separate intermediate nodes (see Fig. 13.11). Here is how this works:

|  |
| --- |
| **protected void** \_createChildren(IOutlineNode parentNode, CoolingProgram p) { TextOnlyOutlineNode vNode = **new** TextOnlyOutlineNode(parentNode, imageHelper.getImage("variable.png"), "variables");  **for** (EObject v: p.getVariables()) { createNode(vNode, v);  }  TextOnlyOutlineNode sNode = **new** TextOnlyOutlineNode(parentNode, imageHelper.getImage("state.png"), "states");  **for** (EObject s: p.getStates()) { createNode(sNode, s);  }  } |

|  |
| --- |
| createNode(parentNode, t); }  } |

We introduce intermediate nodes that do not represent a program element; they are used purely for structuring the tree. The **TextOnlyOutlineNode** is a class we created; it simply extends the class **AbstractOutlineNode** provided by Xtext.

|  |
| --- |
| **public class** TextOnlyOutlineNode **extends** AbstractOutlineNode {  **protected** TextOnlyOutlineNode(IOutlineNode parent,  Image image, Object text) {  **super**(parent, image, text, **false**); }  } |

|  |  |
| --- | --- |
| Xtext provides alphabetical sorting for outlines by default. There is also support for styling the outline (i.e. using styled labels as opposed to simple text) as well as for filtering the tree.  *The Outline in Spoofax* With Spoofax, outlines can be specified declaratively in the editor specification. Abstract syntax tree nodes, which should appear in the outline, are selected based on their syntactic sort and constructor names. For exam- |  |
| ple, the following outline specification will include all entity declarations41: | **\_** |

ample, **Decl.** will include imports

**module** MoblLang-Outliner

**\_ imports** MoblLang-Outliner.generated **.Property** will include all nodes with

**outliner** Entity Outliner their syntactic sort.

Decl.Entity

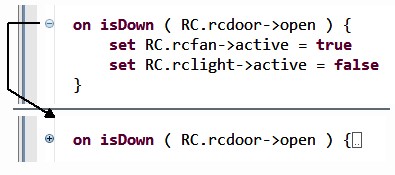
Spoofax analyses the syntax definition and tries to come up with a reasonable default outline specification. We can then either extend the generated specification with our own rules, or create a new one from scratch.

*The Outline in MPS* MPS does not have a customizable outline view. It shows the AST of the complete program as part of the project explorer, but the structure cannot be customized. However, it is of course possible to add arbitrary additional views (called *tools* in MPS) to MPS. The MPS tutorial at **bit.ly/xU78ys** shows how to implement your own outline view.

### 13.9 Code Folding

Code folding refers to the small minuses in the gutter of an editor that let you collapse code regions (see Fig. 13.12). The editor shows an ellipsis (**...**) for the folded parts of the code. Clicking on the **+** or on the ellipsis restores the full code.

*Folding in Xtext* Xtext automatically provides folding for all language concepts that stretch over more than one line42. To



turn off this default behavior, you have to implement your own subclass of **DefaultFoldingRegionProvider** and overwrite the method **isHandled** in a suitable way. For example, to *not* provide folding for **CustomState**s, you could do the following:

|  |
| --- |
| **public class** CLFoldingRegionProvider **extends** DefaultFoldingRegionProvider {  @Override  **protected boolean** isHandled(EObject eObject) { **if** ( eObject **instanceof** CustomState ) { **return false**;  } **return super**.isHandled(eObject); }  } |

*Folding in Spoofax* Spoofax allows to specify folding declaratively in the editor specification. Very similar to the specification of outlines, folding is specified in terms of syntactic sort and constructor names:

|  |
| --- |
| **module** Mobl-Folding  **folding**  Module.\_  Decl.Entity  \_.Function |

As for outlines, Spoofax analyses the syntax definition and tries to come up with a reasonable default specification. We can then either extend the generated specification with our own rules, disable particular specifications by adding a **(disabled)** annotation, or discard it completely. A **(folded)** annotation tells Spoofax to fold a node by default in a newly opened editor, which is typically seen for import sections.

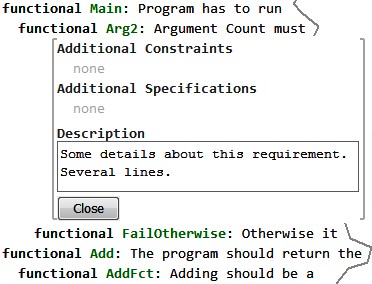
|  |
| --- |
| .  MPS provides a second mechanism that can be used to the  same effect. Since MPS is a projectional editor, some parts of  the editor may be projected conditionally. Fig.  13  .  14  shows a  list/tree of requirements. After pressing  **Ctrl-Shift-D**  on a re- |

|  |  |
| --- | --- |
| Once we’ve set the property to **true**, we have to provide a cell |  |
| that is rendered if the user requests the code to be folded. This allows the text shown as an ellipses to be customized beyond just showing three dots. As Fig. 13.13 shows, we use a **read only model access** cell, which allows us to access the underlying model and return an arbitrary string. In the example, we output the number of "hidden" transitions. |  |

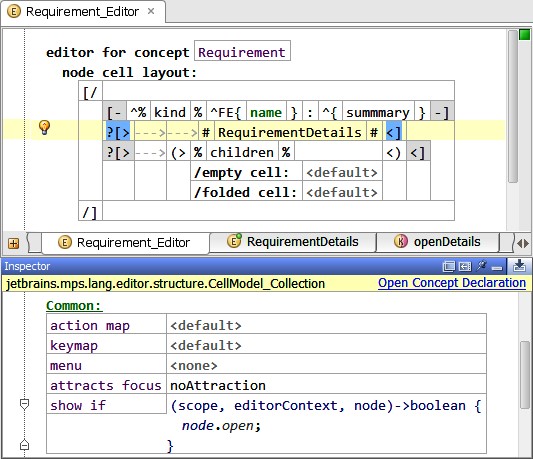
*Folding in MPS* In MPS, folding can be activated for any collection. For example, in a state machine, each state contains a vertical list of transitions. To enable folding for this collection, we set the **uses folding** property for the collection to **true**43. 43 It can also be set to **query**, in which case.

### 13.10 Tooltips/Hover

A tooltip, or hover, is a small, typically yellow window that is shown if the user hovers the mouse over a program element. A hover may show the documentation of the target element, or, when hovering over a reference, some information about the referenced element.



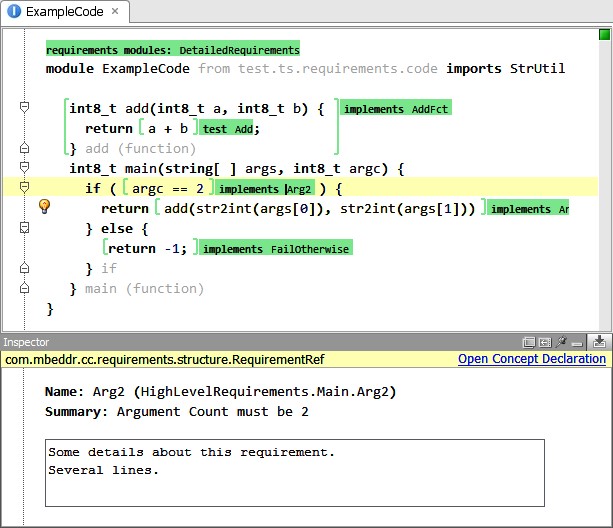
.

1

|  |  |
| --- | --- |
| *Xtext* In Xtext, hovers/tooltips can be customized to various extents. The simplest customization retains the default hover structure (a one-line summary plus a more extensive documentation) and just changes the respective texts.  To change the one-line summary, you override the **getFirst-**  **Line** method in **DefaultEObjectHoverProvider** and return a custom string. The program element for which the hover should be created is represented by the **EObject** passed into the method. To customize the documentation, you override **getDocumen-** |  |
| **tation** in **IEObjectDocumentationProvider**44. |  |
| *Spoofax* Spoofax supports tooltips directly. Tooltips are provided by rewrite rules, which need to be defined as hovers in the editor specification: |  |

**hover** \_: editor-hovering

|  |  |
| --- | --- |
| This line tells Spoofax to use a rewrite rule **editor-hovering** |  |
| to retrieve tooltips for all kinds of abstract syntax tree nodes45. |  |
| When we want to define different rewrite rules for particular constructors, we need to provide a **hover** specification for each constructor, replacing **\_** by **\_.<Constructor>**.  The specified rewrite rules have to follow the typical editor interface on their left-hand side and need to yield strings on |  |



their right-hand sides. The strings are then used as tooltips. For example, the following rewrite rule will provide type information for any typeable node:

|  |
| --- |
| editor-hovering:  (target, position, ast, path, project-path) ->  <type-of; pp-MoblLang-string> target |

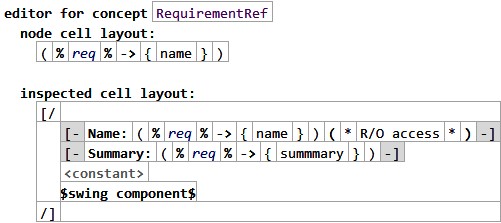
*MPS* MPS does not support tooltips at this time. However, there is an acceptable workaround: any additional information for a program element can be shown in the inspector. For example, if users click on a reference to a requirement in program code, the inspector shows information about the referenced requirement (see Fig. 13.17).

Looking at the editor definition for a **RequirementRef**, you can see that the actual editor (top in Fig. 13.18) shows only the name of the referenced element. The bottom part, the **inspected cell layout**, projects the details about the referenced element.

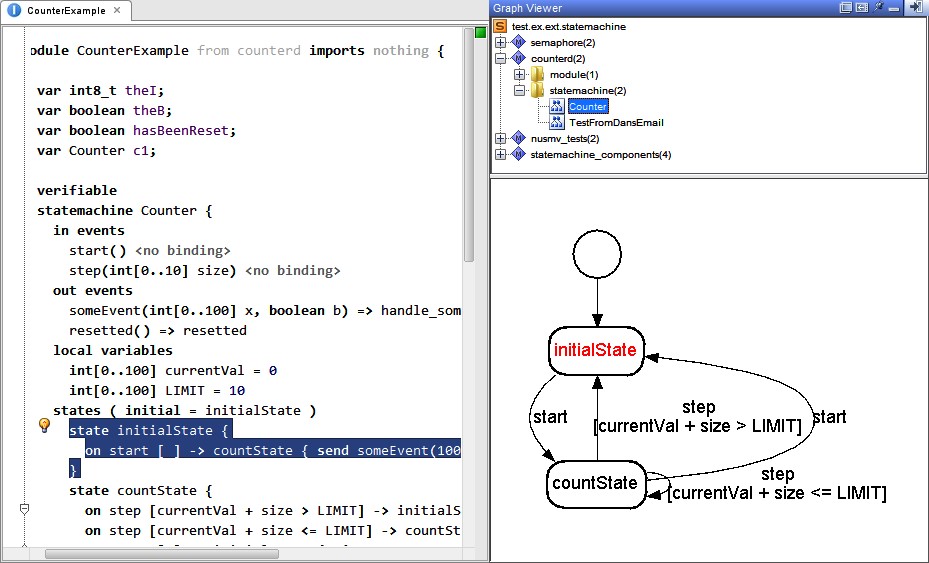
### 13.11 Visualizations

To provide an overview over the structure of programs, readonly graphical representations are useful. Note that these are

.



|  |  |
| --- | --- |
| not necessarily a workaround for not having graphical editors:  visualizations can provide real added value. |  |
| *MPS* In MPS we have integrated ZGRViewer46, a Java- |  |
| based renderer for GraphViz47 dot files. Fig. 13.19 shows an example. |  |



|  |  |
| --- | --- |
|  |  |
| As part of the transformations, we map the model to another model expressed in a graph description language. This model is then generated into a **dot** file. The graph viewer scans the output directory for **dot** files and shows them in the tree view at the top. Double-clicking on a graph node in the tree opens a rendered **dot** file in the graph view. |  |

*Xtext* In Xtext, Jan Koehnlein’s Generic Graph View48 can be used to render diagrams of Xtext models in real-time – the Generic Graph View is an interpreter, so changes in the model lead to updates in the graph immediately.

The mapping from the model to the graph is expressed with an Xtext-based mapping DSL that extends Xbase (Xtext’s reusable expression language), which means you can use Xbase expressions to traverse and query the model you want to visualize

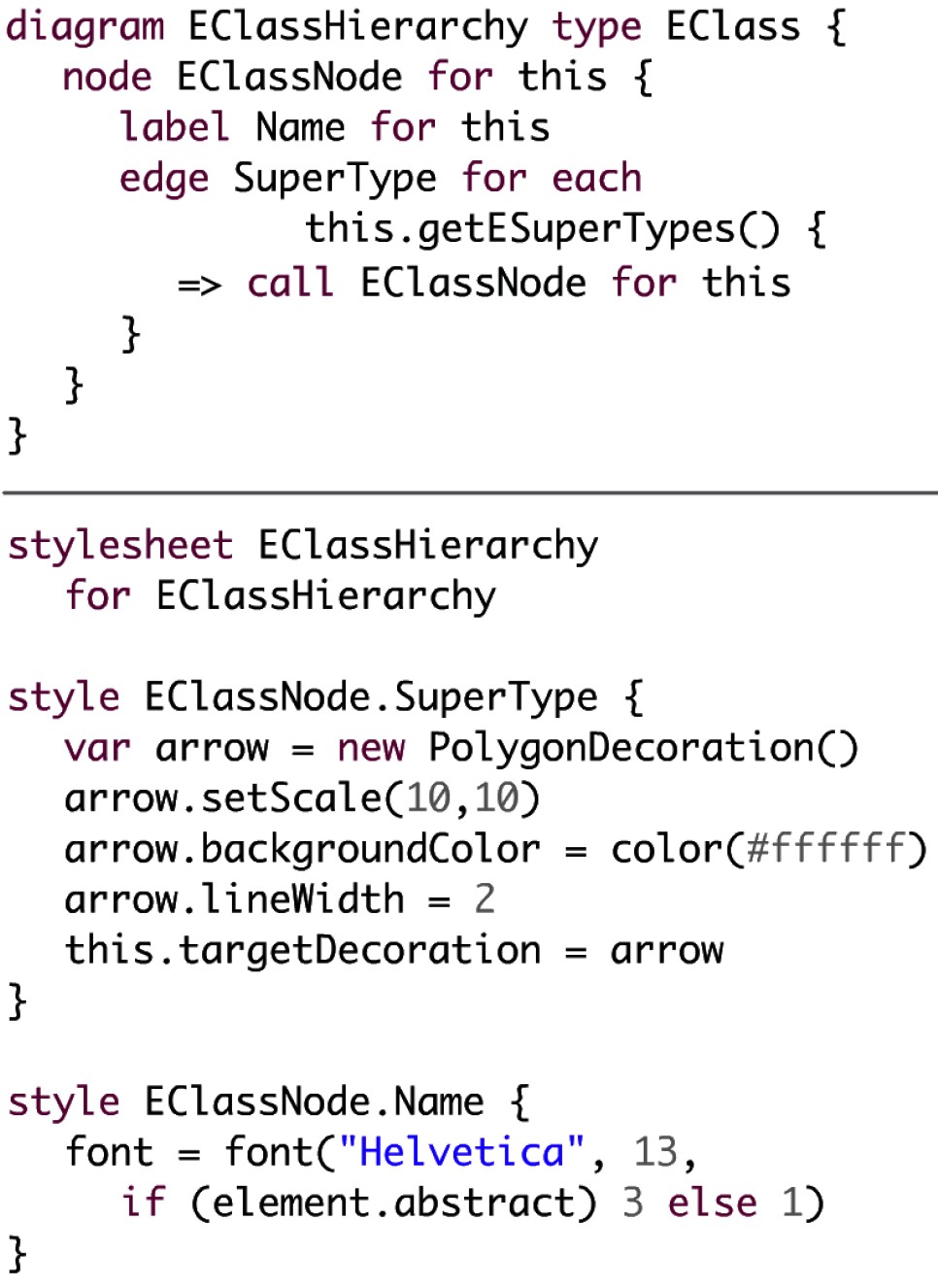
(in Fig. 13.20 an example would be the **this.eSuperTypes()** expression). In addition, a separate styling DSL supports the definition of shapes, colors and line styles. Double-clicking a node in the graph opens the corresponding program element in the Xtext editor.

### 13.12 Diff and Merge

Highlighting the differences between versions of a program and allowing the resolution of conflicts is important in the context of version control integration. For tools like Xtext or Spoofax that store models as plain text this is a non-issue: ex-

48 **github.com/JanKoehnlein/**

**Generic-Graph-View**



isting diff/merge tools can be used, be they in the IDE or on the command line.

For projectional editors such as MPS, the story is more complicated. Since they store the programs based on the abstract syntax (e.g., using XML), diff and merge have to be performed on the concrete projected syntax. MPS provides this feature (see Fig. 7.7 for an example). MPS also annotates the editor with gutter annotations that highlight whether a part of the program has changed relative to the last checkout.

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