MARKUS VOELTER, VOELTER@ACM.ORG

DSL ENGINEERING

Part I

DSL Implementation

This part of the book has been written together Lennart Kats and Guido Wachsmuth (who have contributed the material on Spoofax) and Christian Dietrich, who helped with the language modularization in Xtext.

In this part we describe language implementation with three language workbench tools which are all Open Source and representative for the current state of the art language workbenches: Spoofax, Xtext and MPS. For more example language implementations using more language workbenches, take a look at the Langauage Workbench Competition website at `http://languageworkbenches.net`.

Note that this description are not intended to serve as a tutorial for any of these tools, but as an illustration of the concepts and ideas involved with language implementation in general.

1
Concrete and Abstract Syntax

In this chapter we look at the definition of abstract syntax and concrete syntax, and the mapping between the two in parser-based and projectional editing. We also discuss the advantages and drawbacks of these two appraoches. We discuss the characteristics of typical AST definition formalisms. The meat of the chapter is made of extensive examples of defining language structure and syntax with Xtext, Spoofax and MPS.

The *concrete syntax* (short: CS) of a language is what the user interacts with to create programs. It may be textual, graphical, tabular or any combination thereof. In this book we focus mostly on textual concrete syntaxes for reasons described in . We refer to other forms where appropriate.

The *abstract syntax* (short: AS) of a language is a data structure that holds the information represented by the concrete syntax, but without any of the syntactic details. Keywords and symbols, layout (e.g., whitespace), and comments are typically not included in the AS. Note that the syntactic information that doesn't end up in the AS is often preserved in some "hidden" form so the CS can be reconstructed from the combination of the AS and this hidden information — this bidirectionality simplifies creating IDE features such as quick fixes or formatters a lot.

In most cases, the abstract syntax is a tree data structure. Instances that represent actual programs (i.e., sentences in the language) are hence often called an abstract syntax tree or AST. Some formalisms also support cross-references across the tree, in which case the data structure becomes a graph (with a primary containment hierarchy). It is still usually called an AST.

While the CS is the interface of the language to the user, the AS acts as the API to access programs by processing tools: it is used by developers of validators, transformations and code generators. The

concrete syntax is not relevant in these cases.

To illustrate the relationship between the concrete and abstract syntax, consider the following example program:

```
var x: int;
calc y: int = 1 + 2 * sqrt(x)
```

This program has a hierarchical structure: definitions of `x` and `y` at the top, inside `y` there's a nested expression. This structure is reflected in the corresponding abstract syntax tree. A possible AST is illustrated in Fig. 1.1[1]. The boxes represent program elements, and the names in the boxes represent language concepts that make up the abstract syntax.

[1] We say *possible* because there are typically several ways of structuring the abstract syntax.

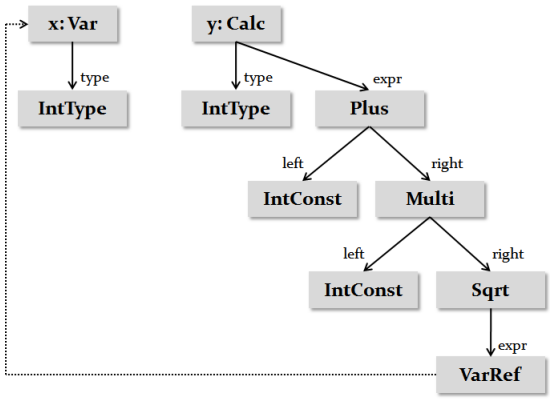


Figure 1.1: Abstract syntax tree for the above program. Boxes represent instances of language concepts, solid lines represent containment, dotted lines represent cross-references

There are two ways of defining the relationship between the CS and the AS as part of language development:

AS first We first define the AS. This is then annotated with concrete syntax definitions. Mapping rules determine how the concrete syntax maps to abstract syntax structures[2].

CS first From a concrete syntax definition, an abstract syntax is derived, either automatically or using hints in the concrete syntax specification[3].

Once the language is defined, there are again two ways how the abstract syntax and the concrete syntax can relate as the language is used to create programs[4]:

Parsing In parser-based systems, the abstract syntax tree is constructed from the concrete syntax of a program; a parser instantiates and populates the AS, based on the information in the program text. In this case, the (formal) definition of the CS is usually called a *grammar*.

[2] This is often done if the AS structure already exists, has to conform to externally imposed constraints or is developed by another party than the language developer.

[3] This is more convenient, but the resulting AS may not be as clean as if it were defined manually; it may contain idiosyncracies that result from the derivation from the CS.

[4] We will discuss these two approaches in more detail in the next subsection.

Projection In projectional systems, the abstract syntax tree is built directly by editor actions, and the concrete syntax is rendered from the AST via projection rules.

Fig. 1.2 shows the typical combinations of these two dimensions. In practice, parser-based systems typically derive the AS from the CS — i.e., CS first. In projectional systems, the CS is usually annotated onto the AS data structures - i.e., AS first.

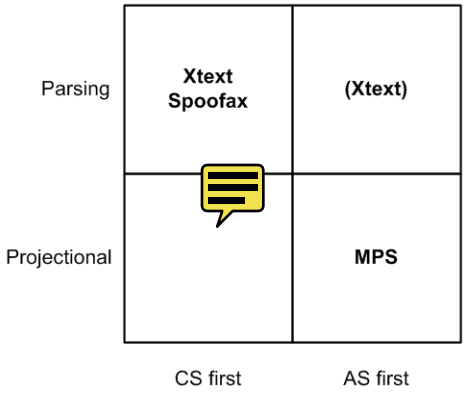


Figure 1.2: Dimensions of defining the concrete and abstract syntax of a language. Xtext is mentioned twice because it supports CS first and AS first, although CS first is the default.

1.1 *Fundamentals of Free Text Editing and Parsing*

Most general-purpose programming environments rely on free text editing, where programmers edit programs at the text/character level to form (key)words and phrases.

A *parser* is used to check the program text (concrete syntax) for syntactic correctness, and create the AST by populating the AS data structures from information extracted from the textual source. Most modern IDEs perform this task in real-time as the user edits the program and the AST is always kept in sync with the program text. Many IDE features — such as content assist, validation, navigation or refactoring support — are based on the always up-to-date AST.

A key characteristic of the free text editing approach is its strong separation between the concrete syntax (i.e., text) and the abstract syntax. The concrete syntax is the principal representation, used for both editing and persistence. The abstract syntax is used under the hood by the implementation of the DSL, e.g. for providing an outline view, validation, and for transformations and code generation.

Many different approaches to parser implementation exist. Each may restrict the syntactic freedom of a DSL, or constrain the way in

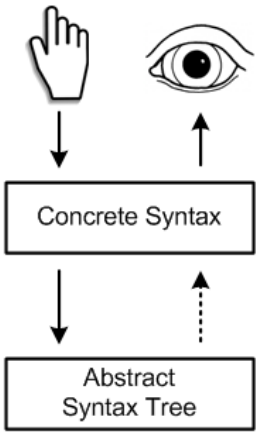


Figure 1.3: In parser-based systems, the user only interacts with the concrete syntax, and the AST is constructed from the information in the text via a parser.

which a particular syntax must be specified. It is important to be aware of these restrictions since not all languages can be comfortably, or even at all implemented by every parser implementation approach. You may have heard terms like context free, ambiguity, look-ahead, LL, (LA)LR or PEG. These all pertain to a certain class of parser implementation approaches. We provide more details on the various grammar and parser classes further on in this section.

1.1.1 *Parser Generation Technology*

In traditional compilers, parsers are often written by hand as a big program that reads a stream of characters and uses recursion to create a tree structure. Manually writing a parser in this way requires significant expertise in parsing and a large development effort. For standardized programming languages that don't change very often, and that have a large user community, this approach makes sense. It can lead to very fast parsers that also exhibit good error recovery (the ability to continue parsing after a syntax error has been found).

In contrast, language workbenches, and increasingly also traditional compilers, *generate* a parser from a grammar. A grammar is a syntax specification written in a DSL for formally defining textual concrete syntax. These generated parsers may not provide the same performance or error recovery as a hand-tailored parser constructed by an expert, but they provide bounded performance guarantees that make them (usually) more than fast enough for modern machines. However, the most important argument to use parser generation is that the effort for building a parser is **much** lower than manually writing a custom parser. The grammar definition is also much more readable and maintainable than the actual parser implementation (either custom-written or generated).

■ *Parsing versus scanning* Because of the complexity inherent in parsing, parser implementations tend to split the parsing process into a number of phases. In the majority of cases the text input is first separated into a sequence of *tokens* (i.e., keywords, identifiers, literals, comments, or whitespace) by a *scanner* (sometimes also called lexer). The *parser* then constructs the actual AST from the token sequence[5]. This simplifies the implementation compared to directly parsing at the character level. A scanner is usually implemented using direct recognition of keywords and a set of regular expressions to recognize all other valid input as tokens.

Both the scanner and parser can be generated from grammars (see below). A well-known example of a scanner (lexer) generation tool is `lex`[6]. Modern parsing frameworks, such as ANTLR[7] do their own scanner generation.

[5] Note that many parser generators allow you to add arbitrary code (called actions) to the grammar, for example to check constraints or interpret the program. We strongly recommend not to do this: instead, a parser should *only* check for syntactic correctness and build the AST. All other processing should be built on top of the AST. This separation between AST construction and AST processing results in much more maintainable language implementations.

[6] http://dinosaur.compilertools.net/

[7] http://www.antlr.org/

Note that the word "parser" now has more than one meaning: it can either refer to the combination of the scanner and the parser or to the post-scanner parser only. Usually the former meaning is intended (both in this book as well as in general) unless scanning and parsing are discussed specifically.

A separate scanning phase has direct consequences for the overall parser implementation, because the scanner typically isn't aware of the context of any part of the input — only the parser has this awareness. An example of a typical problem that arises from this is that keywords can't be used as identifiers even though often the use of a keyword wouldn't cause ambiguity in the actual parsing. The Java language is an example of this: it uses a fixed set of keywords, such as `class` and `public`, that cannot be used as identifiers.

A context-unaware scanner can also introduce problems when languages are extended or composed. In the case of Java, this was seen with the `assert` and `enum` keywords that were introduced in Java 1.4 and Java 5, respectively. Any programs that used identifiers with those names (such as unit testing APIs) were no longer valid. For composed languages, similar problems arise as constituent languages have different sets of keywords and can define incompatible regular expressions for lexicals such as identifiers and numbers.

The term "scannerless parsing" refers to the absence of a separate scanner and in this case we don't have the problems of context-unaware scanning illustrated above, because the parser operates at a character level and statefully processes lexicals and keywords. Spoofax (or rather: the underlying parser technology SDF) uses scannerless parsing.

■ *Grammars* Grammars are the formal definition for concrete textual syntax. They consist of so-called production rules which define how valid textual input ("sentences") looks like[8]. Grammars form the basis for syntax definitions in text-based workbenches such as Spoofax and Xtext[9].

Fundamentally, grammar production rules can be expressed in Backus-Naur Form (BNF)[10], written as $S ::= P_1 \ldots P_n$. This grammar defines a symbol S by a series of pattern expressions $P_1 \ldots P_n$. Each pattern expression can refer to another symbol or can be a literal such as a keyword or a punctuation symbol. If there are multiple possible patterns for a symbol, these can be written as separate productions (for the same symbol), or the patterns can be separated by the `|` operator to indicate a choice. An extension of BNF, called Extended BNF (EBNF)[11], adds a number of convenience operators such as `?` for an optional pattern, `*` to indicate zero or more occurrences, and `+` to indicate one or more occurrences of a pattern expression.

[8] They can also be used to "produce" valid input by executing them "the other way round", hence the name.

[9] In these systems, the production rules are enriched with information beyond the pure grammatical structure of the language, such as the semantical relation between references and declarations.

[10] http://en.wikipedia.org/wiki/Backus-Naur_Form

[11] http://en.wikipedia.org/wiki/Extended_Backus-Naur_Form

As an example, the following code is an example of a grammar for a simple arithmetic expression language using BNF notation. Basic expressions are built up of `NUM` number literals and the + and * operators.

```
Exp ::= NUM
      | Exp "+" Exp
      | Exp "*" Exp
```

Note how expression nesting is described using recursion in this grammar: the `Exp` rule calls itself, so sentences like `2 + 3 * 4` are possible. This poses two practical challenges for parser generation systems: first, the precedence and associativity of the operators is not described by this grammar. Second, not all parser generators provide full support for recursion. We elaborate on these issues in the remainder of the section and in the Spoofax and Xtext examples.

■ *Grammar classes* BNF can describe any grammar that maps textual sentences to trees based only on the input symbols. These are called *context-free grammars* and can be used to parse the majority of modern programming languages. In contrast, *context-sensitive grammars* are those that also depend on the context in which a partial sentence occurs, making them suitable for natural language processing but at the same time, making parsing itself a lot harder since the parser has to be aware of a lot more than just the syntax.

Parser generation was first applied in command-line tools such as `yacc` in the early seventies[12]. As a consequence of relatively slow computers, much attention was paid to the efficiency of the generated parsers. Various algorithms were designed that could parse text in a bounded amount of time and memory. However, these time and space guarantees could only be provided for certain subclasses of the context-free grammars, described by acronyms such as LL(1), LL(k), LR(1), and so on. A particular parser tool supports a specific class of grammars — e.g., ANTLR supports LL(k) and LL(*). In this naming scheme, the first L stands for left-to-right scanning, and the second L in LL and the R in LR stand for leftmost and rightmost derivation. The constant k in LL(k) and LR(k) indicates the maximum number (of tokens or characters) the parser will look ahead to decide which production rule it can recognize. Typically, grammars for "real" DSLs tend to need only finite look-ahead and many parser tools effectively compute the optimal value for k automatically. A special case is LL(*), where k is unbounded and the parser can look ahead arbitrarily many tokens to make decisions.

[12] http://dinosaur.compilertools.net

Supporting only a subclass of all possible context-free grammars poses restrictions on the languages that are supported by a parser generator. For some languages, it is not possible to write a grammar in

a certain subclass, making that particular language unparseable with a tool that only supports that particular class of grammars. For other languages, a natural context-free grammar exists, but it must be written in a different, often awkward way to conform to the subclass. This will be illustrated in the Xtext example, which uses ANTLR as the underlying LL(k) parser technology.

Parser generators can detect if a grammar conforms to a certain subclass, reporting conflicts that relate to the implementation of the algorithm: *shift/reduce* or *reduce/reduce* conflicts for LR parsers, and *first/first* or *first/follow* conflicts and direct or indirect *left recursion* for LL parsers. DSL developers can then attempt to manually refactor the grammar to address those errors[13]. As an example, consider a grammar for property or field access, expressions of the form `customer.name` or `"Tim".length`[14]:

```
Exp ::= ID
      | Exp "." ID
      | STRING
```

This grammar uses left-recursion: the left-most symbol of one of the definitions of `Exp` is a call to `Exp`, i.e. it is recursive. Left-recursion is not supported by LL parsers such as ANTLR.

The left-recursion can be removed by *left-factoring* the grammar, i.e. by changing it to a form where all left recursion is eliminated. The essence of left-factoring is that the grammar is rewritten in such a way that all recursive production rules consume at least one token or character before going into the recursion. Left-factoring introduces additional rules that act as intermediaries and often makes repetition explicit using the `+` and `*` operators. The example grammar uses recursion for repetition, which can be made explicit as follows:

```
Exp ::= ID
      | (Exp ".")+ ID
      | STRING
```

The resulting grammar is still left-recursive, but we can introduce an intermediate rule to eliminate the recursive call to `Exp`:

```
Exp ::= ID
      | (FieldPart ".")+ ID
      | STRING

FieldPart ::= ID
            | STRING
```

[13] Understanding these errors and then refactoring the grammar to address them can be non-trivial since it requires an understanding of the particular grammar class and the parsing algorithm.

[14] Note that we use ID to indicate identifier patterns and STRING to indicate string literal patterns in these examples.

Unfortunately, this resulting grammar still has overlapping rules (a first/first conflict) as the `ID` symbol matches more than one rule. This conflict can be eliminated by removing the `Exp ::= ID` rule and making the + (one or more) repetition into a * (zero or more) repetition:

```
Exp       ::= (FieldPart ".")* ID
            | STRING

FieldPart ::= ID
            | STRING
```

This last grammar now describes the same language as the original field access grammar shown above, but conforms to the LL(1) grammar class[15]. In the general case, not all context-free grammars can be mapped to one of the restricted classes. Valid, unambiguous grammars exist that cannot be factored to any of the restricted grammar classes. In practice, this means that some languages cannot be parsed with LL or LR parsers.

[15] Unfortunately, it is also much more verbose. Refactoring "clean" context free grammars to make them conform to a particular grammar class usually makes the grammars larger and/or uglier.

■ *General parsers* Research into parsing algorithms has produced parser generators specific to various grammar classes, but there has also been research in parsers for the full class of context-free grammars. A naive approach to avoid the restrictions of LL or LR parsers may be to add backtracking, so that if any input doesn't match a particular production, the parser can go back and try a different production. Unfortunately, this approach risks exponential execution times or non-termination and usually exhibits poor performance.

There are also general parsing algorithms that can *efficiently* parse the full class. In particular, generalized LR (GLR) parsers[16] and Earley parsers[17] can parse unambiguous grammars in linear time and gracefully cope with ambiguities with cubic ($O(n^3)$) time in the worst case. Spoofax is an example of a language workbench that uses GLR parsing.

[16] http://en.wikipedia.org/wiki/GLR_parser

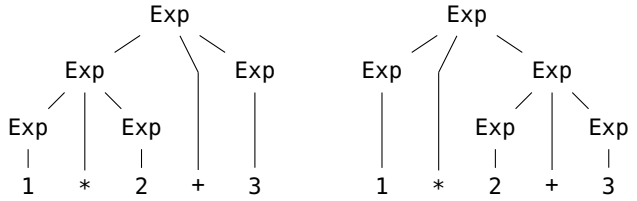
[17] http://en.wikipedia.org/wiki/Earley_parser

■ *Ambiguity* Grammars can be *ambiguous* meaning that at least one valid sentence in the language can be constructed in more than one (non-equivalent) way from the production rules[18], corresponding to multiple possible ASTs. This obviously is a problem for parser implementation as some decision has to be made on which AST is preferred. Consider again the expression language introduced above.

[18] This also means that this sentence can be parsed in more than one way.

```
Exp ::= NUM
      | Exp "+" Exp
      | Exp "*" Exp
```

It is ambiguous, since for a string `1*2+3` there are two possible trees (corresponding to different operator precedences).



The grammar does not describe which interpretation should be preferred. Parser generators for restricted grammar classes and generalized parsers handle ambiguity differently. We discuss both approaches.

■ *Ambiguity with grammar classes* LL and LR parsers are deterministic parsers: they can only return one possible tree for a given input. This means they can't handle any grammar that has ambiguities, including our simple expression grammar. Determining whether a grammar is ambiguous is a classic undecidable problem. However, it is possible to detect violations of the LL or LR grammar class restrictions, in the form of conflicts. These conflicts do not always indicate ambiguities (as seen with the field access grammar discussed above), but by resolving all conflicts (if possible) an unambiguous grammar can be formed.

Resolving grammar conflicts in the presence of associativity, precedence, and other risks of ambiguity requires carefully layering the grammar in such a way that it encodes the desired properties. To encode left-associativity and a lower priority for the + operator we can rewrite the grammar as follows:

```
Expr ::= Expr "+" Mult
       | Mult
Mult ::= Mult "*" NUM
       | NUM
```

The resulting grammar is a valid LR grammar. Note how it puts the + operator in the highest layer to give it the lowest priority[19], and how it uses left-recursion to encode left-associativity of the operators. The grammar can be left-factored to a corresponding LL grammar as follows. We will see more extensive examples of this approach in the section on Xtext (Section 1.5).

[19] A + will end up further up in the expression tree than a ∗; this means that the ∗ has higher precedence, since any interpreter or generator will encounter the ∗ first.

```
Expr ::= Mult ("+" Mult)*
Mult ::= NUM ("*" NUM)*
```

■ *Ambiguity with generalized parsers* General parsers accept grammars regardless of recursion or ambiguity. That is, the expression grammar is readily accepted as a valid grammar. In case of an ambiguity, the generated parser simply returns *all possible abstract syntax trees*, e.g. a left-associative tree and a right-associative tree for the expression `1*2+3`. The different trees can be manually inspected to determine what ambiguities exist in the grammar, or the desired tree can be programmatically selected.

A way of programmatically selecting one alternative is *disambiguation filters*. For example, left-associativity can be indicated on a per-production basis:

```
Exp ::= NUM
      | Exp "+" Exp {left}
      | Exp "*" Exp {left}
```

This indicates that both operators are left-associative (using the `{left}` annotation from Spoofax). Operator precedence can be indicated with relative priorities or with precedence annotations:

```
Exp ::= Exp "*" Exp {left}
>
Exp ::= Exp "+" Exp {left}
```

The > indicates that the * operator binds stronger than the + operator. This kind of declarative disambiguation is commonly found in GLR parsers, but typically not available in parsers that support only more limited grammar classes[20].

[20] As even these simple examples show, this style of specifying grammars leads to simpler, more readable grammars. It also makes language specification much simpler, since developers don't have to understand the errors mentioned above

■ *Grammar Evolution and Composition* Grammars evolve as languages change and new features are added. These features can be added by adding single, new productions, or by composing the grammar with an existing grammar. Composition of grammars is an efficient way of reusing grammars and quickly constructing or extending new grammars[21]. As a basic example of grammar composition, consider once again our simple grammar for arithmetic expressions:

[21] We discuss this extensively in Section **??**

```
Expr ::= NUM
       | Expr "*" Expr
       | Expr "+" Expr
```

Once more operators are added and the proper associativities and precedences are specified, such a grammar forms an excellent unit for reuse[22]. As an example, suppose we want to compose this grammar with the grammar for field access expressions[23]:

[22] For examople, expressions can be used as guard conditions in state machines, for pre- and postconditions in interface definitions or to specify derived attributes in a data definition language.

[23] Here we consider the case where two grammars use a symbol with the identical name `Expr`. Some grammar definition formalisms support mechanisms such as grammar mixins and renaming operators to work with grammar modules where the symbol names do not match

```
Expr ::= ID
       | STRING
       | Expr "." ID
```

In the ideal case, composing two such grammars should be trivial — just copy them into the same file. However, reality is often less than ideal. There are a number of challenges that arise in practice, related to ambiguity and to grammar class restrictions.

- Composing arbitrary grammars risks introducing ambiguities that did not exist in either of the two constituent grammars. In the case of the arithmetic expressions and field access grammars, care must specifically be taken to indicate the precedence order of all operators with respect to all others. With a general parser, new priority rules can be added without changing imported two grammars. When an LL or LR parser is used, it is often necessary to change one or both of the composed grammars to eliminate any conflicts. This is because in a general parser, the precedences are *declarative* (additional preference specification can simply be added at the end), whereas in LL or LR parsers the precedence information is encoded in the grammar structure (and invasive changes to this structure may be required).
- We have shown how grammars can be massaged with techniques such as left-factoring in order to conform to a certain grammar class. Likewise, any precedence order or associativity can be encoded by massaging the grammar to take a certain form. Unfortunately, all this massaging makes grammars very resistant to change and composition: after two grammars are composed together, the result is often no longer LL or LR, and another manual factorization step is required.
- Another challenge is in composing scanners. When two grammars that depend on a different lexical syntax are composed together, conflicts can arise. For example, consider what happens when we compose the grammar of Java with the grammar of SQL:

```
for (Customer c : SELECT customer FROM accounts WHERE balance < 0) {
  ...
}
```

The SQL grammar reserves keywords such as `SELECT`, even though they are not reserved in Java. Such a language change could break compatibility with existing Java programs which happen to use a

variable named SELECT. This problem can only be avoided by a scannerless parser, which cons the lexical syntax in the context it appears instead during a separate scanning stage, where no context is considered.

1.2 *Fundamentals of Projectional Editing*

In parser-based approaches, users use text editors to enter character sequences that represent programs. A parser then checks the program for syntactic correctness and constructs an abstract syntax tree from the character sequence. The AST contains all the semantic information expressed by the program.

In projectional editors, the process happens the other way round: as a user edits the program, the AST is modified directly. A projection engine then creates some representation of the AST with which the user interacts, and which reflects the changes. This approach is well-known from graphical editors in general, and the MVC pattern specifically. In editing a UML diagram, users don't draw pixels onto a canvas, and a "pixel parser" then creates the AST. Rather, the editor creates and instance of `uml.Class` as you drag a class from the palette to the canvas. A projection engine renders the diagram, in this case drawing a rectangle for the class. This approach can be generalized to work with any notation, including textual.

This explicit instantiation of AST objects happens by picking the respective concept from the code completion menu using a character sequence defined by the respective concept. If at any given program location two concepts can be instantiated using *the same character sequence*, then the projection editor prompts the user to decide[24]. Once a concept is instantiated, it is stored as a node with a unique ID (UID) in the AST. References between program elements are based on actual pointers (references to UIDs), and the projected syntax that represents the reference can be arbitrary. The AST is actually an , an abstract syntax graph, from the start because cross-references are first-class rather than being resolved after parsing. The program is stored using a generic tree persistence mechanism, often XML.

The projectional approach can deal with arbitrary syntactic forms including traditional text, symbols (as in mathematics), tables or graphics. Since no grammar is used, grammar classes are not relevant here. In principle, projectional editing is simpler than parsing, since there is no need to "extract" the program structure from a flat textual source. However, as we will see below, the challenge in projectional editing lies making the editing experience convenient[25].

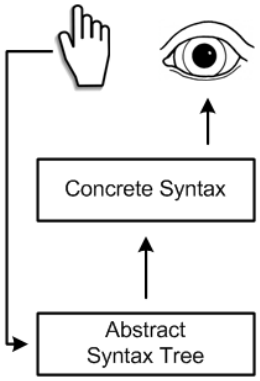


Figure 1.4: In projectional systems, the user sees the concrete syntax, but all editing gestures directly influence the AST. The AST is *not* extracted from the concrete syntax, which means the CS does not have to be parseable.

[24] As discussed above, this is the situation where many grammar-based systems run into problems from ambiguity.

[25] In particular, editing notations that look like text should be editable with the editing gestures known from text editors.

1.3 *Comparing Parsing and Projection*

1.3.1 *Editing Experience*

In free text editing, any regular text editor will do. However, users expect powerful IDE that includes support for syntax coloring, code completion, go to definition, find references, error annotations, refactoring and the like. Xtext and Spoofax provide such IDE support. However, you can always go back to any text editor to edit the programs.

In projectional editing, this is different since a normal text editor is obviously not sufficient as you would be editing some textual representation of the AST. A specialized editor has to be supplied. Like in free text editing, it has to provide the IDE support features mentioned above. MPS provides those. However, there is another challenge: for textual-looking notations, it is important that the editor tries and makes the editing experience as text-like as possible, i.e. the keyboard actions we have gotten used to from free-text editing should work as far as possible. MPS does a decent job here, using, among others, the following strategies:

- Every language concept that is legal at a given program location is available in the code completion menu. In naive implementations, users have to select the language concept (based on its name) and instantiate it. This is inconvenient. In MPS, languages can instead define aliases for language concepts, allowing users to "just type" the alias, after which the concept is immediately instantiated[26].
- So-called side transforms make sure that expressions can be entered conveniently. Consider a local variable declaration `int a = 2;`. If this should be changed to `int a = 2+3;` the `2` in the init expression needs to be replaced by an instance of the binary `+` operator, with the `2` in the left slot and the `3` in the right. Instead of removing the `2` and manually inserting a `+`, users can simply type `+` on the right side of the `2`; the system performs the tree refactoring that moves the `+` to the root of the subtree, puts the `2` in the left slot, and then puts the cursor into the right slot, to accept the second argument. This means that expressions (or anything else) can be entered linearly, as expected. For this to work, operator precedence has to be specified, and the tree has to be constructed taking these precedences into account. Precedence is typically specified by a number associated with each operator, and whenever using a side transformation to build an expression, the tree is automatically reshuffled to make sure that those operators with a higher precedence number are further down in the tree.
- Delete actions are used to similar effect when elements are deleted.

[26] By making the alias the same as the leading keyword (e.g. `if` for an `IfStatement`), users can "just type" the code.

Deleting the 3 in 2+3 first keeps the plus, with an empty right slot. Deleting the + then removes the + and puts the 2 at the root of the subtree.

- Wrappers support instantiation of concepts that are actually children of the concepts allowed at a given location. Consider again a local variable declaration `int a;`. The respective concept could be `LocalVariableDeclaration`, a subconcept of `Statement`, to make it legal in method bodies (for example). However, users simply want to start typing `int`, i.e. selecting the content of the `type` field of the `LocalVariableDeclaration`. A wrapper can be used to support entering `Types` where `LocalVariableDeclarations` are expected. Once a `Type` is selected, the wrapper implementation creates a `LocalVariableDeclaration`, puts the `Type` into its `type` field, and moves the cursor into the `name` slot.
- Smart references achieve a similar effect for references (as opposed to children). Consider pressing `Ctrl-Space` after the + in 2+3. Assume further, that a couple of local variables are in scope and that these can be used instead of the 3. These should be available in the code completion menu. However, technically, a `VariableReference` has to be instantiated, whose `variable` slot then is made to point to any of the variables in scope. This is tedious. Smart references trigger special editor behavior: if in a given context a `VariableReference` is allowed, the editor *first* evaluates its scope to find the possible targets and then puts those targets into the code completion menu. If a user selects one, *then* the `VariableReference` is created, and the selected element is put into its `variable` slot. This makes the reference object effectively invisible in the editor.
- Smart delimiters are used to simplify inputting list-like data that is separated with a specific separator symbol, such as parameter lists. Once a parameter is entered, users can press comma, i.e. the list delimiter, to instantiate the next element.

Notice that, except for having to get used to the somewhat different way of editing programs, all other problems can be remedied with good tool support. Traditionally, this tool support has not always existed or been sufficient, and projectional editors have gotten a bit of a bad reputation because of that. In case of MPS this tool support is available, and hence, MPS provides a productive and pleasant working environment.

1.3.2 Language Modularity

As we have seen in , language modularization and composition is an important building block in working with DSLs. Parser-based and

projectional editors come with different trade-offs in this respect. Notice that is this section we only consider syntax issues — semantics has been covered in .

In parser-based systems it depends on the grammar class supported by the tool whether composition can be supported well. As we have said above, the problem is that the result of combining two or more independently developed grammars into one may become ambiguous, for example, because the same character sequence is defined as two different tokens. The resulting grammar cannot be parsed and has to be disambiguated manually, typically by invasively changing the composite grammar. This of course breaks modularity and hence is not an option. Parsers that do not support the full set of context-free grammars, such as ANTLR, and hence Xtext, have this problem. Parsers that do support the full set of context-free grammars, such as the GLR parser used as part of Spoofax, do not have this problem. While a grammar may become ambiguous in the sense that a program may be parseable in more than one way, this can be resolved by declaratively specifying which alternative should be used. This specification can be made externally, *without* invasively changing the resulting grammar, retaining modularity.

In projectional editors, language modularity and composition is not a problem at all. There is no grammar, no parsing, no grammar classes, and hence no problem with composed grammars becoming ambiguous. Any combination of languages will be syntactically valid. If a composed language would be ambiguous, the user has to make a disambiguating decision as the program is entered. For example, in MPS, if in a given location two language concepts are available under the same alias, just typing the alias won't bind, and the user has to manually decide by picking one alternative from the code completion menu.

1.3.3 Notational Freedom

Parser-based systems process linear sequences of character symbols. Traditionally, the character symbols were taken from the ASCII character set, resulting in textual programs being made up from "plain text". With the advent of unicode, a much wider variety of characters is available while still sticking to the linear sequence of character symbols approach. For example, the Fortress programming language[27] makes use of this: greek letters and a wide variety of different bracket styles can be used in the programs. However, character layout is always ignored. For example it is not possible to use parsers to handle tabular notations, fraction bars or even graphics[28].

In projectional editing, this limitation does not exist. A projectional editor never has to extract the AST from the concrete syntax; editing gestures directly influence the AST, and the concrete syntax is ren-

[27] http://en.wikipedia.org/wiki/Fortress_(programming_

[28] There have been experimental parsers for two-dimensional structures such as tables and even for graphical shapes, but these have never made it beyond the experimental stage.

```
component AComp extends nothing {
  ports:
    provides Decider decider
  contents:
    int8_t decide(int8_t x, int8_t y) ⬅ op decider.decide {
      return int8_t, 0
```

	x == 0	x > 0
y == 0	0	1
y > 0	1	2

```
    }
}
```

Figure 1.5: A table embedded in an oth-

dered from the AST. This mechanism is basically like a graphical editor. Notations other than text can be used. For example, MPS supports tables as well as simple diagrams. Since these non-textual notations are handled the same way as the textual ones (possibly with other input gestures), they can be mixed easily: tables can be embedded into textual source, and textual languages can be used within table cells (see Fig. 1.5). Textual notations can also be used inside boxes or as connection labels in diagrams.

1.3.4 *Language Evolution*

If the language changes, existing instance models temporarily become outdated in the sense that they had been developed for the old version of the language. If the new language is not backward compatible, these existing models have to be migrated to conform to the updated language.

Since projectional editors store the models as structured data where each program node points to the language concept it is an instance of, the tools have to take special care that such "incompatible" models can still be opened and the migrated, manually or by a script, to the new version of the language. MPS supports this feature, and it is even possible to distribute migration scripts with (updated) languages to run the migration automatically[29].

[29] It is also possible to define quick fixes that run *automatically*; so whenever a concept that is marked as `deprecated`, this quick fix can trigger an automatic migration to a new concpet.

Free text editing does not have this problem. A model, essentially a sequence of characters, can *always* be opened in the editor. The program may not be parseable, but users can always update the program manually, or with global search and replace using regular expressions.

1.3.5 *Infrastructure Integration*

Today's software development infrastructure is typically text-oriented. Many tools used for diff and merge, or tools like `grep` and regular expressions are geared towards textual storage. This means that DSLs that use free text editing integrate nicely with these tools.

In projectional IDEs, special support needs to be provided for infrastructure integration. Since the CS is not pure text, a generic persistence format is used, typically based on XML. While XML is technically text as well, it is not practical to perform diff, merge and the like on the level of the XML. Therefore, special tools need to be provided for diff and merge. MPS provides integration with the usual version control systems and handles diff and merge in the IDE, using the concrete, projected syntax. Note that since every program element has a unique ID, *move* can be distinguished from *delete*, providing richer semantics for diff and merge. Fig. 1.6 shows an example of an MPS diff.

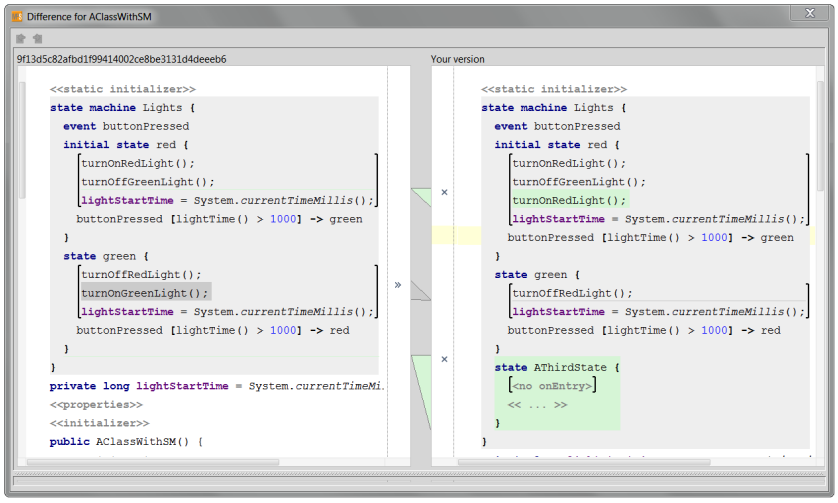


Figure 1.6: The diff/merge tool presents MPS programs in their concrete syntax, i.e. text for textual notations. However, other notations, such as tables, would also be rendered in their native form.

1.3.6 *Other*

In parser-based systems, the complete AST has to be reconstructable from the CS. This implies that there can be no information in the tree that is *not* obtained from parsing text. This is different in projectional editors. For example, the textual notation could only project a subset of the information in the tree. The same information can be projected with different projections, each possibly tailored to a different stakeholder, and showing a different subset from the overall data. Since the tree uses a generic persistence mechanism, it can hold data that has not been planned for in the original language definition. All kinds of meta data (documentation, presence conditions, requirements traces) can be stored, and projected if required. MPS supports so-called annotations, where additional data can be added to model elements of existing languages and projecting that data inside the original projection, all without changing the original language specification.

1.4 *Characteristics of AST formalisms*

Most AST formalisms, aka meta meta models[30], are ways to represent trees or graphs. Usually, such an AST formalism is "meta circular" in the sense that it can describe itself.

Note that this section is really just a very brief overview over the three AST formalisms relevant to Xtext, Spoofax and MPS. We will illustrate them in more detail in the respective tool example sections.

[30] *Abstract syntax* and *meta model* are typically considered synonyms, even though they have different histories (the former comes from the parser/grammar community whereas the latter comes from the modeling community). Consequently, the formalisms for defining ASTs are conceptually similar to meta meta models.

1.4.1 EMF Ecore

The Eclipse Modeling Framework[31] (EMF) is at the core of all Eclipse Modeling tools. Its core component is the Ecore meta model, a version of the EMOF standard[32]. The central concepts are: `EClass` (representing AS elements or language concepts), `EAttribute` (representing primitive properties of `EClasses`), `EReference` (representing associations between `EClasses`) and `EObject` (representing instances of `EClasses`, i.e. AST nodes. `EReferences` can be containing or not – each `EObject` can be contained by at most one `EReference` instance. Fig. 1.7 shows a class diagram of Ecore.

[31] http://www.eclipse.org/modeling/emf/

[32] http://en.wikipedia.org/wiki/Meta-Object_Facility

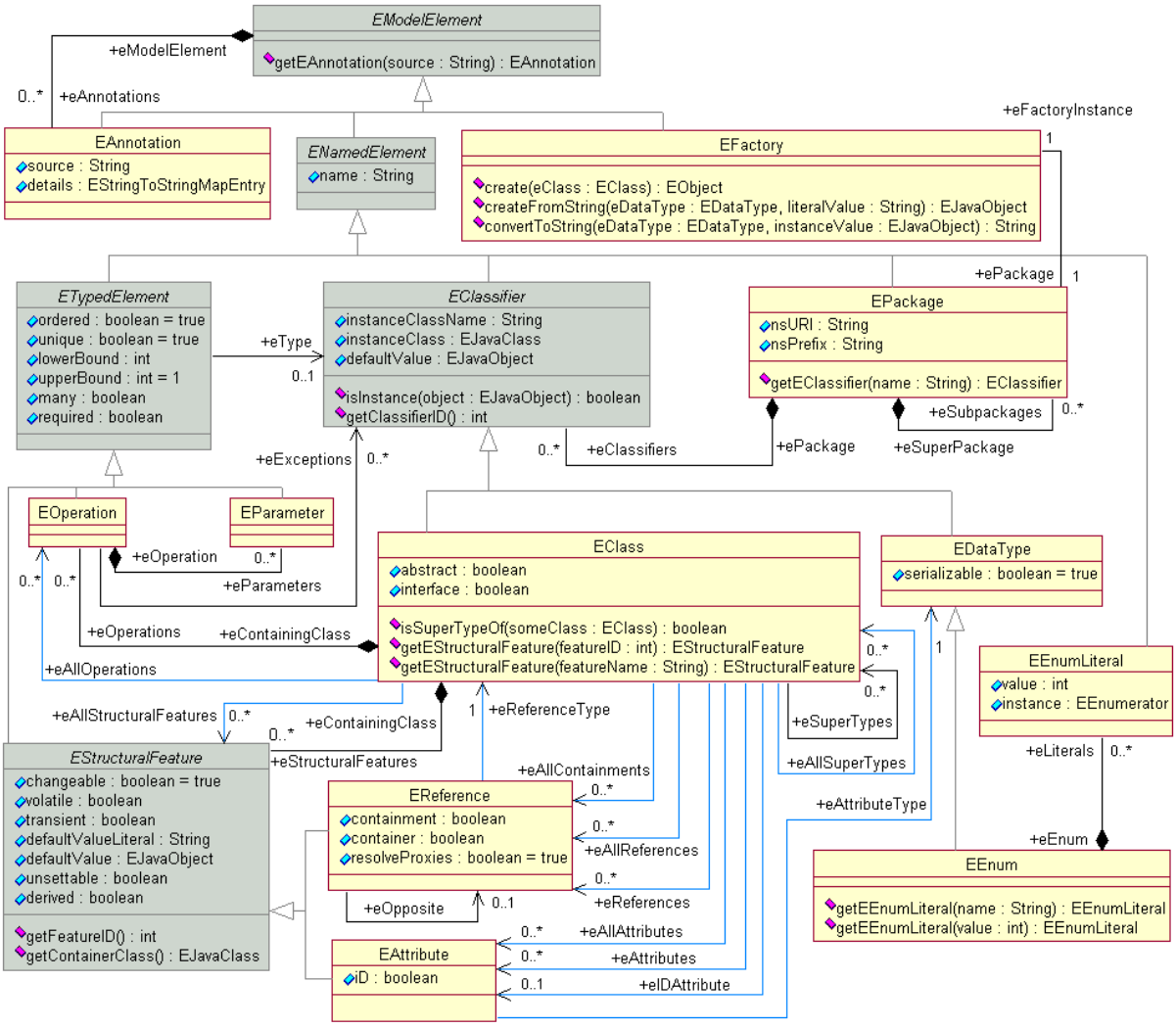


Figure 1.7: The Ecore meta model rendered as a UML diagram.

When working with EMF, the Ecore file plays a central role. From that, all kinds of other aspects are derived, specifically, a generic tree editor, and a generated Java API for accessing an AST. This also forms the

basis for Xtext's model processing: The Ecore file is derived from the grammar, and the parser, when executed, builds an in-memory tree of `EObjects` representing the AST of the parsed program.

EMF has grown to be a fairly large ecosystem within the Eclipse community and numerous projects use EMF as a basis for model manipulation and persistence, with Ecore for meta model definition.

1.4.2 Spoofax' ATerm

Spoofax uses the ATerm format to represent abstract syntax. ATerm provides a generic tree structure representation format that can be serialized textually similar to XML or JSON. Each tree node is called an ATerm, or simply a *term*. Terms consist of the following elements: Strings (`"Mr. White"`), Numbers (`15`), Lists (`[1,2,3]`) and constructor applications (`Order(5, 15, "Mr. White")`) for labelled tree nodes with a fixed number of children.

Compared to XML or JSON, perhaps the most significant distinction is that ATerms often rely on the order of subterms rather than on labels. For example, a product may be modeled in JSON as follows:

```
{
  "product": {
    "itemnumber": 5,
    "quantity": 15,
    "customer": "Mr. White"
  }
}
```

Note how this specification includes the actual data describing the particular product (the model), but also a description of each of the elements (the meta model). With XML, a product would be modeled in a similar fashion. An equivalent of the JSON above written in ATerm format would be the following:

```
Order([ItemNumber(5), Quantity(15), Customer("Mr.\ White")])
```

However, this representation contains a lot of redundant information that also exists in the grammar. Instead, such a product can be written as `Order(5, 15, "Mr. White")`. This more concise notation tends to make it slightly more convenient to use in handwritten transformations.

The textual notation of ATerms can be used for exchanging data between tools and as a notation for model transformations or code generation rules. In memory, ATerms can be stored in a tool-specific way (i.e., simple Java objects in the case of Spoofax). The generic structure and serializability of ATerms also allows them to be converted to

other data formats. For example, the `aterm2xml` and `xml2aterm` tools can convert between ATerms and XML.

In addition to the basic elements above, ATerms support annotations to add additional information to terms. These are similar to attributes in XML. For example, it is possible to annotate a product number with its product name:

```
Order(5{ProductName("Apples")}, 15, "Mr. White")
```

Spoofax also uses annotations to add information about references to other parts of a model to an abstract syntax tree. While ATerms only form trees, the annotations are used to represent the graph-like references.

1.4.3 MPS' Structure Definition

Every program element in MPS is a *node*. A node has a structure definition and projection rules for rendering. This is also true for language definitions. Nodes are instances of *concepts*, which conceptually corresponds to EMF's `EClass`. Like `EClasses`, concepts are meta circular, i.e. there is a concept that defines the properties of concepts:

```
concept ConceptDeclaration extends AbstractConceptDeclaration
                          implements INamedConcept
                                     IStructureDeprecatable

  instance can be root: false

  properties:
    helpURL : string
    rootable : boolean

  children:
    InterfaceConceptReference  implements                0..n
    LinkDeclaration            linkDeclaration           0..n
    PropertyDeclaration        propertyDeclaration       0..n
    ConceptProperty            conceptProperty           0..n
    ConceptLink                conceptLink               0..n
    ConceptPropertyDeclaration conceptPropertyDeclaration 0..n
    ConceptLinkDeclaration     conceptLinkDeclaration    0..n

  references:
    ConceptDeclaration         extends                  0..1

  concept properties:
    alias = concept
```

A concept may extend a single other concept and implement any number of interfaces[33]. It can declare references and child collections. It may also have a number of primitive-type properties as well as a couple of "static" features. In addition, concepts can have behavior methods.

[33] Note that interfaces can provide implementations for the methods they specify — they are hence more like Scala traits.

While the MPS structure definition is proprietary to MPS and does

not implement any accepted industry stanard, it is conceptually very close to Ecore[34].

[34] This is illustrated by the fact the exporters and importers to and from Ecore have been written.

1.5 *Xtext Example*

Cooling programs[35] represent the behavioral aspect of the refrigerator descriptions. Here is a trivial program that can be used to illustrate some of the features of the language. The program is basically a state machine.

[35] This and the other examples refer back to the case studies introduced at the beginning of the book in section

```
cooling program HelloWorld uses stdlib {

    var v: int
    event e

    init { set v = 1 }

    start:
       on e { state s }

    state s:
        entry { set v = 0 }

}
```

The program declares a variable `v` and an event `e`. When the program starts up, the `init` section is executed, setting `v` to `1`. The system then (automatically) transitions into the `start` state. There it waits until it receives the `e` event. It then transition to the state `s`, where it uses an entry action to set `v` back to `0`. More complex programs include checks of changes of properties of hardware elements (`aCompartment->currentTemp`) and commands to the hardware `set aCompartment->isCooling = true`, as shown in the next snippet:

```
start:
    check ( aCompartment->currentTemp > maxTemp ) {
        set aCompartment->isCooling = true
        state initialCooling
    }
    check ( aCompartment->currentTemp <= maxTemp ) {
        state normalCooling
    }

state initialCooling:
    check ( aCompartment->currentTemp < maxTemp ) {
        state normalCooling
    }
```

■ *Grammar basics* In Xtext, the syntax is specified using a grammar, a collection of productions that are typically called *parser rules*. These rules specify the concrete syntax of a program element, as well as its

It is also possible to first create the Ecore meta model and then define a grammar for it. While this is a bit more work, it is also more flexible because not all possible Ecore meta models can be described implicitly by a grammar. For example, Ecore interfaces cannot be expressed from the grammar.

mapping to the AS. From the grammar, Xtext generates an Ecore meta model to describe the AS. Here is the definition of the `CoolingProgram` rule:

```
CoolingProgram:
    "cooling" "program" name=ID "{"
      (events+=CustomEvent |
        variables+=Variable)*
      (initBlock=InitBlock)?
      (states+=State)*
    "}";
```

Rules begin with the name (`CoolingProgram`), a colon, and then the rule body. The body defines the syntactic structure of the language concept defined by the rule. In our case, we expect the keywords `cooling` and `program`, followed by an `ID`. `ID` is a *terminal rule* that is defined in the parent grammar from which we inherit (not shown). `ID` is defined as an unbounded sequence of lowercase and uppercase characters, digits, and the underscore, although it may not start with a digit. This terminal rule is defined as follows:

```
terminal ID: ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
```

In pure grammar languages, one would write `"cooling" "program" ID "{ ..."`, specifying that after the two keywords we expect an `ID`. However, Xtext grammars don't just express the concrete syntax — they also determine the mapping to the AS. We have encountered two such mappings so far. The first one is implicit (although it can be made explicit as well): the name of the rule will be the name of the generated meta class. So we will get a meta class `CoolingProgram`. The second mapping we have encountered in `name=ID`. It specifies that the meta class gets a property `name` that holds the contents of the `ID` from the parsed program text. Since nothing else is specified in the `ID` terminal rule, the type of this property defaults to `EString`, Ecore's version of a string data type.

The rest of the definition of a cooling program is enclosed in curly braces. It contains three elements: first the program contains a collection of events and variables (the `*` specifies unbounded multiplicity), an optional init block (optionality is specified by the `?`) and a list of states. Let us inspect each of these in more detail.

The expression `(states+=State)*` specifies that there can be any number of `State` instances in the program. The `CoolingProgram` meta class gets a property `states`, it is of type `State` (the meta class derived from the `State` rule). Since we use the `+=` operator, the `states` property will be typed to be a *list* of `States`. In case of the optional

`init` block, the meta class will have an `initBlock` property, typed as `InitBlock` (whose parser rule we don't show here), with a multiplicity of 0..1. Events and variables are more interesting, since the vertical bar operator is used within the parentheses. The asterisk expresses that whatever is inside the parentheses can occur any number of times — note that the use of a `*` usually goes hand in hand in hand with the use of a `+=`. Inside the parentheses we expect either a `CustomEvent` *or* a `Variable`, which is expressed with the `|`. Variables are assigned to the variables collection, events are assigned to the events collection.

This notation means that we can mix events and variables in any order. The following alternative notation would first expect all events, and then all variables.

```
    (events+=CustomEvent)*
    (variables+=Variable)*
```

The definition of `State` is interesting, since `State` is intended to be an abstract meta class with several subtypes.

```
State:
    BackgroundState | StartState | CustomState;
```

The vertical bar operator is used here to express syntactic alternatives. This is translated to inheritance in the meta model. The definition of `CustomState` is shown in the following code snippet. It uses the set of grammar language features explained above.

```
CustomState:
    "state" name=ID ":"
        (invariants+=Invariant)*
        ("entry" "{"
            (entryStatements+=Statement)*
        "}")?
        ("eachTime" "{"
            (eachTimeStatements+=Statement)*
        "}")?
        (events+=EventHandler | signals+=SignalHandler)*;
```

`StartState` and `BackgroundState`, the other two subtypes of `State`, share some properties. Consequently, Xtext's AS derivation algorithm pulls them up into the abstract `State` meta class so they can be accessed polymorphically. Fig. 1.8 shows the resulting meta model using EMF's tree view.

■ *References* Let us now look at statements and expressions. `States` have entry and exit actions, procedural statements that are executed when a state is entered and left, respectively. `set v = 1` in the hello

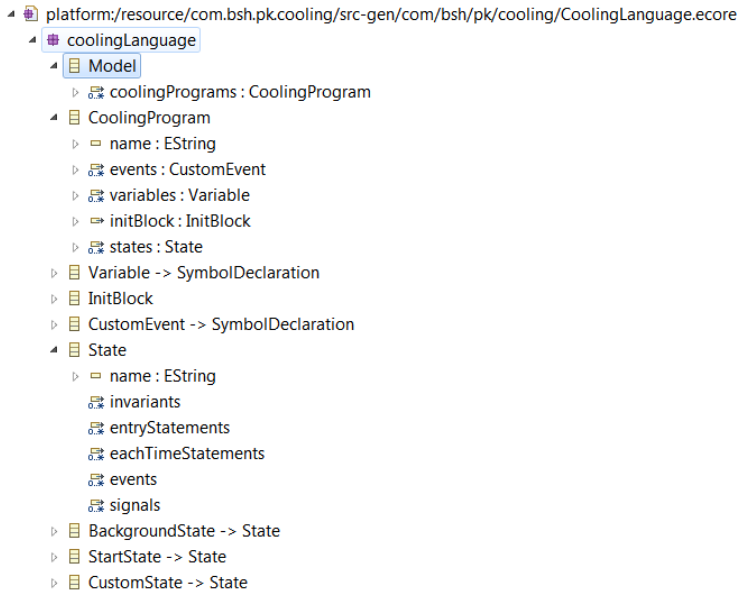


Figure 1.8: Part of the Ecore meta model derived from the Xtext grammar for the cooling program. Grammar rules become `EClasses`, assignments in the grammar rules become `EProperties` (i.e. attributes and references).

world program is an example. `Statement` itself is `abstract` and has the various kinds of statements as subtypes/alternatives:

```
Statement:
    IfStatement | AssignmentStatement | PerformAsyncStatement |
    ChangeStateStatement | AssertStatement;

ChangeStateStatement:
    "state" targetState=[State];

AssignmentStatement:
    "set" left=Expr "=" right=Expr;
```

The `ChangeStateStatement` is used to transition into another state. It uses the keyword `state` followed by a reference to the actual target state. Notice how Xtext uses square brackets to express the fact that the `targetState` property points to an *existing* state as opposed to containing a new one (which would be written as `targetState=State`), i.e. the square brackets leads to non-containing cross-references.

This is another example of where the Xtext grammar language goes beyond classical grammar languages, where one would write `"state" targetStateName=ID;`. Writing it this way only specifies that we expect an `ID` after the `state` keyword. The fact that we call it `targetStateName` communicates *to the programmer* that we expect this text string to correspond to the name of a state — a later phase in model processing *resolves* the name to an actual state reference. Typically, the code to resolve the reference has to be written manually, because there is no way for the tool to automatically derive from the grammar that this `ID` is actually a reference to a `State`. In Xtext, the `targetState=[State]`

notation makes this explicit, so the resolution of the reference can be automatic (taking into account manually written scopes, as discussed in the next chapter). This notation also has the advantage that the resulting meta class types the `targetState` property to `State` (and not just to a string), which makes processing the models much easier.

Note that the cross-reference definition only specifies the target type (`State`) of the cross-reference, but not the concrete syntax of the reference itself. By default, the `ID` terminal is used for the reference syntax, i.e. a simple (identifier-like) text string is expected. However, this can be overridden by specifying the concrete syntax terminal behind a vertical bar in the reference[36].

[36] Notice that in this case the vertical bar does not represent an *alternative*, it is merely used as a *separator* between the target type and the terminal used to represent the reference.

```
ChangeStateStatement:
    "state" targetState=[State|QID];

QID: ID ("." ID)*;
```

The other remaining detail is scoping. During the linking phase, where the text of `ID` (or `QID`) is used to find the target node, several objects with the same name might exist, or some target elements are not visible based on visibility rules of the language. To constrain the possible reference targets, scoping functions are used. These will be explained in the next chapter.

■ *Expressions* The `AssignmentStatement` shown above is one of the statements that uses expressions. The following snippet is a subset of the actual definition of expressions (we have omitted some additional expressions that don't add anything to the description here).

```
Expr:
    ComparisonLevel;

ComparisonLevel returns Expression:
    AdditionLevel ((({Equals.left=current} "==") |
                   ({LogicalAnd.left=current} "&&") |
                   ({Smaller.left=current} "<"))
                                    right=AdditionLevel)?;

AdditionLevel returns Expre
    MultiplicationLevel ((( left=current} "+") |
                         ({Minus.left=current} "-")) right=Multi ionLevel)?;

MultiplicationLevel returns Expression:
    PrefixOpLevel ((({Multi.left=current} "*") |
                   ({Div.left=current} "/")) right=PrefixOpLevel)?;

PrefixOpLevel returns Expression:
    ({NotExpression} "!" "(" expr=Expr ")") |
    AtomicLevel;

AtomicLevel returns Expression:
    ({TrueExpr} "true") |
    ({F r} "false") |
    ({P r} "(" expr=Expr ")") |
    ({Nu erLiteral} value=DECIMAL_NUMBER) |
    ({SymbolRef} symbol=[SymbolDeclaration|QID]);
```

To understand the above definition, we first have to explain in more detail how AST construction works in Xtext. Obviously, as the text is parsed, meta classes are instantiated and the AST is assembled. However, instantiation of the respective meta class happens only upon the first assignment to one of its properties. If no assignment is performed at all, no object is created. For example in grammar rule `TrueLiteral: "true";` no instance of `TrueLiteral` will ever be created, because there is nothing to assign. In this case, an action can be used to force instantiation: `TrueLiteral: {TrueLiteral} "true";`[37]. Unless otherwise specified, an assignment such as `name=ID` is always interpreted as an assignment on the object that has been created most recently. The `current` keyword can be used to access that object in case it *itself* needs to be assigned to a property of another AST object.

[37] Notice that the action can instantiate meta classes other than those that are derived from the rule name (we could write `TrueLiteral: {SomeOtherThing} "true";`. While this would not make sense in this case, we'll use this feature later.

Now we know enough about AST construction to understand how expressions are encoded and parsed. In the expression grammar above, for the rules with the `Level` suffix, no meta classes are created, because (as Xtext is able to find out statically) they are never instantiated. They merely act as a way to encode precedence. To understand this, let's consider how `2 * 3` is parsed:

- The `AssignmentStatement` refers to the `Expr` rule in its `left` and `right` properties, so we "enter" the expression tree at the level of `Expr` (which is the root of the expression hierarchy).
- The `Expr` rule just calls the `ComparisonLevel` rule, which calls `AdditionLevel`, and so on. No objects are created at this point, since no assignment to any property is performed.
- The parser "dives down" until it finds something that matches the first symbol in the parsed text: the `2`. This occurs on `AtomicLevel`, as it matches the `DECIMAL_NUMBER` terminal. At this point it creates an instance of `NumberLiteral` and assigns the number `2` to the `value` property. It also sets the `current` object to point to the just created `NumberLiteral`.
- The `AtomicLevel` rule ends, and the stack is unwound. We're back at `PrefixOpLevel`, in the second branch. Since nothing else is specified after the call to `AtomicLevel`, we unwind once more.
- We're now back at the `MultiplicationLevel`. The rule is not finished yet and we try to match a `*` and a `/`. The match on `*` succeeds. At this point the so-called assignment action on the left side of the `*` kicks in (`Multi.left=current`). This action creates an instance of `Multi`, and assigns the `current` (the `NumberLiteral` created before)

to its `left` property. Then it makes the newly created `Multi` the new `current`. At this point we have a subtree with the `*` at the root, and the `NumerLiteral` in the left property.

- The rule hasn't ended yet. We dive down to `PrefixOpLevel` and `AtomicLevel` once more, matching the `3` in the same way as the two before. The `NumerLiteral` for `3` is assigned to the `right` property as we unwind the stack.
- At this point we unwind the stack further, and since no more text is present, no more objects are created. The tree structure is as we had expected.

If we'd parsed `4 + 2*3` the `+` would have matched before the `*`, because it is "mentioned earlier" in the grammar. It is in a lower-precedence group, the `AdditionLevel`. Once we're a the `4 +` tree, we'd go down again to match the `2`. As we unwind the stack after matching the `2` we'd match the `*`, creating a `Multi` again. The `current`, at this point, would be the `2`, so it would be put onto the `left` side of the `*`, making the `*` the `current`. Unwinding further, that `*` would be put onto the `right` side of the `+`, building the tree just as we'd expect.

Notice how a rule at a given level only always delegates to rules at higher precedence levels. So higher precedence rules always end up further down in the tree. If we want to change this, we can use parentheses (see the `ParenExpr` in the `AtomicLevel`): inside those, we can again embed an `Expr`, i.e. we jump back to the lowest precedence level[38].

Note that once you understand the basic approach, it is easy to add new expressions with a precedence similar to another one (just add it as an alternative to the respective `Level` rule) or to introduce a new precedence level (just interject a new `Level` rule between two existing ones).

[38] This somewhat convoluted approach to parsing expressions and encoding precedence is a consequence of the LL(k) grammar class support by ANTLR, which underlies Xtext. We have discussed this topic earlier . Xtext also supports syntactic predicates which are annotations in the grammar that tell the parser which alternative to take in case of an ambiguity. We don't discuss this any further in the book.

1.6 Spoofax Example

Mobl's [39] data modeling language provides entities, properties and functions. To illustrate the language, below are two data type definitions related to a shopping list app. It supports lists of items that can be favorited, checked, and so on, and are associated with some `Store`.

[39] Mobl is a DSL for defining applications for mobile devices. It is based on HTML 5 and is closely related to WebDSL, which has been introduced earlier.

```
module shopping

entity Item {
  name     : String
  checked  : Bool
  favorite : Bool
```

```
  onlist   : Bool
  order    : Num
  store    : Store
}
```

In mobl, most files starts with a module header, which can be followed by a list of entity type definitions. In turn, each entity can have one or more property or function definitions (shown in the next example snippet).

■ *Grammar basics* In Spoofax, the syntax of languages is described using SDF[40]. SDF is a modular and flexible syntax definition formalism, supported by the SGLR parser generator. It can generate efficient, Java-based scannerless and general parsers. An example of a production written in SDF is

[40] http://www.syntax-definition.org/

```
"module" ID Entity* -> Start {"Module"}
```

The pattern on the left-hand side of the arrow is matched by the symbol `Start` on the right-hand side. Note that SDF uses the exact opposite order for productions as the grammars we've discussed so far, switching the left-hand and right-hand side. After the right-hand side, SDF productions may specify annotations using curly brackets. Most productions specify a quoted *constructor label* that is used for the abstract syntax. This particular production creates a tree node with the label `Module`. The tree node will have two children that represent the `ID` and the list of `Entities`, respectively. In contrast to Xtext, these children are not named; instead, they are identified via the position in the child collection (the `ID` is fist, the `Entity` list is second).

The left-hand side of an SDF production is the pattern it matches against. SDF supports symbols, literals, and character classes in this pattern. Symbols are references to other productions, such as `ID`. Literals are quoted strings such as `"module"` that must appear in the input literally. Character classes specify a range of characters expected in the input, e.g. `[A-Za-z]` specifies that an alphabetic character is expected. We discuss character classes in more detail below.

The basic elements of SDF productions can be combined using operators. The `A*` operator shown above specifies that zero or more occurrences of `A` are expected. `A+` specifies that one or more are expected. `A?` specifies that zero or one are expected. `{A B}*` specifies zero or more `A` symbols, separated by `B` symbols are expected. As an example, `{ID ","}*` is a comma-separated list of identifiers. `{A B}+` specifies one or more `A` symbols separated by `B` symbols.

Fig. 1.9 shows an SDF grammar for a subset of mobl's entities and functions syntax. The productions in this grammar should have few

surprises, but it is interesting to note how SDF groups a grammar in different sections. First, the `context-free start symbols` section indicates the start symbol of the grammar. Then, the `context-free syntax` section list the context-free syntax productions, forming the main part of the grammar. Terminals are defined in the `lexical syntax` section.

Figure 1.9: A basic SDF grammar for a subset of Mobl. The grammar does not yet specify the associativity, priority, or name bindings of the language.

```
module MoblEntities

context-free start symbols

  Module

context-free syntax

  "module" ID Decl*                    -> Module {"Module"}
  "import" ID                          -> Decl   {"Import"}
  "entity" ID "{" EntityBodyDecl* "}" -> Decl {"Entity"}
  ID ":" ID                            -> EntityBodyDecl {"Property"}

  "function" ID "(" {Param ","}* ")" ":" ID "{" Statement* "}"
                                          -> EntityBodyDecl {"Function"}
  ID ":" ID                            -> Param {"Param"}
  "var" ID "=" Expr ";"                -> Statement {"Declare"}
  "return" Exp ";"                     -> Statement {"Return"}

  Exp "." ID "(" Exp  ")" -> Exp {"MethodCall"}
  Exp "." ID              -> Exp {"FieldAccess"}
  Exp "+" Exp             -> Exp {"Plus"}
  Exp "*" Exp             -> Exp {"Mul"}
  ID                      -> Exp {"Var"}
  INT                     -> Exp {"Int"}

lexical syntax

  [A-Za-z][A-Za-z0-9]* -> ID
  [0-9]+               -> INT
  [\ \t\n]             -> LAYOUT
```

■ *Lexical syntax* As Spoofax uses a scannerless parser, all lexical syntax can be customized in the SDF grammar. It provides default definitions for common lexical syntax elements such as strings, integers, floats, whitespace and comments, but when needed, custom syntax can be specified.

Most lexical syntax is specified using character classes such as `[0-9]`. Each character class is enclosed in square brackets, and can consist of ranges of characters (c_1`-`c_2), letters and digits (e.g., `x` or `4`), non-alphabetic literal characters (e.g., `_`), and escapes (e.g. `\n`). A complement of a character class can be obtained using the ~ operator, e.g. `~[A-Za-z]` matches all non-alphabetic characters. For whitespace and comments a special terminal `LAYOUT` can be used.

SDF implicitly inserts `LAYOUT` in between all symbols in context-free

productions. This behavior is the key distinguishing feature between context-free and lexical productions: lexical symbols such as identifiers and integer literals cannot be interleaved with layout. The second distinguishing feature is that lexical syntax productions usually do not have a constructor label in the abstract syntax, as they form terminals in the abstract syntax trees (i.e. they don't own any child nodes).

■ *Abstract syntax* To produce abstract syntax trees, Spoofax uses the ATerm format, described in Section 1.4.2. SDF combines the specification of concrete and abstract syntax, primarily through the specification of constructor labels. The way this works is most easily understood using an example. Spoofax allows users to view the abstract syntax of any input file. As an example, the following is the textual representation of an abridged abstract syntax term for the shopping module shown at the beginning of this section:

```
Module(
  "shopping",
  [ Entity(
      "Item",
      [Property("name", "String"), Property("checked", "Bool"), ...]
    )
  ]
])
```

Note how this term uses the constructor labels of the syntax above: `Module`, `Entity`, and `Property`. The children of each node correspond to the symbols referenced in the production: the `Module` production first referenced `ID` symbol for the module name and then included a list of `Decl` symbols (lists are in square brackets).

In addition to constructor labels, productions that specify parentheses can use the special `bracket` annotation:

```
"(" Exp ")" -> Exp {bracket}
```

The `bracket` annotation specifies that there should not be a separate tree node in the abstract syntax for the production. This means that an expression `1 + (2)` would produce `Plus("1","2")` in the abstract syntax, and not `Plus("1",Parens("2"))`.

■ *Precedence and associativity* SDF provides special support for specifying the associativity and precedence of operators or other syntactic constructs. As an example, let us consider the production of the `Plus` operator. So far, it has been defined as

```
Exp "+" Exp -> Exp {"Plus"}
```

Based on this operator, a parser can be generated that can parse an expression such as `1 + 2` to a term `Plus("1", "2")`. However, the production does not specify if an expression `1 + 2 + 3` should be parsed to a term `Plus("1", Plus("2", "3"))` or `Plus(Plus("1", "2"), "3")`. If you try the grammar in Spoofax, it will show *both* interpretations using the special `amb` constructor:

```
amb([
  Plus("1", Plus("2", "3")),
  Plus(Plus("1", "2"), "3")
)]
```

The `amb` node indicates an *ambiguity* and shows all possible interpretations. Whenever an ambiguity is encountered in a file, it is marked with a warning in the editor.

Ambiguities can be resolved by adding annotations to the grammar that describe the intended interpretation. For the `Plus` operator, we can resolve the ambiguity by specifying that it is left-associative, using the `left` annotation:

```
Exp "+" Exp -> Exp {"Plus", left}
```

In a similar fashion, SDF makes it possible to describe the precedence order of operators. For this, the productions can be placed in a `context-free priorities` section:

```
context-free priorities
  Exp "*" Exp -> Exp {"Mul", left}
>
  Exp "+" Exp -> Exp {"Plus", left}
```

This example specifies that the `Mul` operator has a higher priority than the `Plus` operator, resolving the ambiguity that arises for an expression such as `1+2*3`.

■ *Reserved keywords and production preference* Parsers generated with SDF do not use a scanner, but include processing of lexical syntax in the parser. Since scanners operate without any context information, they will simply recognize any token that corresponds to a keyword in the grammar as a reserved keyword, *irrespective of its location in the program*. In SDF, it is also possible to use keywords that are not reserved, or keywords that are only reserved in a certain context. As an example, the following is a legal entity in mobl:

```
entity entity {

}
```

Since our grammar did not specify that `entity` is a reserved word, it can be used as a normal `ID` identifier. However, there are cases where it is useful to reserve keywords, for example to prevent ambiguities. Consider what would happen if we added a new production for a `true` literal:

```
"true" -> Exp {"True"}
```

If we would now parse an expression `true`, it would be ambiguous: it matches the `True` production above, but it also matches the `Var` production, as `true` is a legal variable identifier[41]. Keywords can be reserved in SDF by using a production that rejects a certain interpretation:

[41] So it is ambiguous because *at the same location in a program* both interpretations are possible.

```
"true" -> ID {reject}
```

This expresses that `true` can never be interpreted as an identifier. Alternatively, we could say that we prefer the one interpretation over the other:

```
"true" -> Exp {"True", prefer}
```

This means that this production is to be preferred if there are any other interpretations. However, since these interpretations cannot always be foreseen as grammars are extended, it is considered good practice to use the more specific `reject` approach instead[42].

[42] This is the situation where a projectional editor like MPS is more flexible, since instead of running into an ambiguity, it would prompt the user to decide which interpretation is correct as he types `true`.

■ *Longest match* Most scanners apply a *longest match* policy for scanning tokens. This means that if it is possible to include the next character in the current token, the scanner will always do so. For most languages, this is the expected behavior, but in some cases longest match doesn't work. SDF instead allows the grammar to specify the intended behavior. In Spoofax, the default is specified in the *Common* syntax module using a `lexical restrictions` section:

```
lexical restrictions
  ID -/- [A-Za-z0-9]
```

This section restricts the grammar by specifying that any `ID` cannot be directly followed by a character that matches `[A-Za-z0-9]`. Effectively, it enforces a longest match policy for the `ID` symbol. SDF also allows the use of lexical restrictions for keywords. By default it does not enforce longest match, which means it allows the following definition of a mobl entity:

```
entityMoblEntity {}
```

As there is no longest match, the parser can recognize the `entity` keyword even if it is not followed by a space. To avoid this behavior, we can specify a longest match policy for the `entity` keyword:

```
lexical restrictions
  "entity" -/- [A-Za-z0-9]
```

■ *Name bindings* So far we have discussed purely syntax specification in SDF. Spoofax also allows the specification of name binding annotations in SDF grammars, which specify semantic relations between productions. We discuss how these relations are specified in Chapter 2.

1.7 MPS Example

We start by defining a simple language for state machines, roughly similar to the one used in the mbeddr extendible C. Core concepts include `StateMachine`, `State`, `Transition` and `Trigger`. The state machine can be embedded in C code as we will see later. The language supports the definition of state machines as shown in the following piece of code:

```
module LineFollowerStatemachine {

  statemachine LineFollower {
    events unblocked()
           blocked()
           bumped()
           initialized()
    states (initial = initializing) {
      state initializing {
        on initialized [ ] -> running {  }
      }
      state paused {
        on unblocked [ ] -> running {  }
      }
      state running {
```

```
        on blocked [ ] -> paused {  }
        on bumped [ ] -> crashed {  }
      }
      state crashed {
      }
   }
  }
}
```

■ *Concept Definition* MPS is projectional, so we start with the definition of the AS. In MPS, AS elements are called concepts. The code below shows the definition of the concept `Statemachine`. It contains a collection of `States` and a collection of `InEvents`. It also contains a reference to one of the states to mark it as the `initial` state. The `alias` is defined as `statemachine`, so typing this word inside C modules instantiates a state machine (it picks the `Statemachine` concept from the code completion menu). State machines also implements a couple of interfaces; `IIdentifierNamedElement` contributes a property `name`, `IModuleContent` makes the state machine embeddable in C `Modules` — the module owns a collection of `IModuleContents`, just like the state machine contains states and events.

```
concept Statemachine extends BaseConcept
                     implements IModuleContent
                                ILocalVarScopeProvider
  children:
    State   states   0..n
    InEvent inEvents 0..n

  references:
    State   initial  1 specializes: <none>

  concept properties:
    alias = statemachine
```

A `State` contains two `StatementLists` as `entryActions` and `exitActions`. `StatementList` is a concept defined by the `com.mbeddr.core.statements` language. To make that visible, our statemachine language extends `com.mbeddr.core.statements`. Finally, a `State` contains a collection of `Transitions`.

```
concept Transition
  children:
    Trigger       trigger  1
    Expression    guard    1
    StatementList actions  1

  references:
    State         target   1

  concept properties:
    alias = on
```

`Transitions` contain a `Trigger`, a guard condition, transition actions and a reference to the target state. The trigger is an abstract concept; various specializations are possible, the default implementation is the `EventTrigger`, which references an `Event`. The guard condition is an `Expression`, a concept reused from `com.mbeddr.core.expressions`. A type system rule will be defined later to constrain this expression to be Boolean. The target state is a reference, i.e. we point to an existing state instead of owning it. `actions` is another `StatementList` that can contain arbitrary C statements used as the transition actions.

■ *Editor Definition* Editors, i.e. the projection rules, are made of cells. When defining editors, various cell types are arranged so that the resulting syntax has the desired structure. Fig. 1.10 shows the editor definition for the `State` concept. It uses an `indent` collection of cells with various style attributes to arrange the `state` keyword and name, the entry actions, the transitions and the exit actions in a vertical list. Entry and exit actions are shown only if the respective `StatementList` is not empty (a condition is attached to the respective cells, marked by the `?` in front of the cell). An intention is used (see next section) to add a new statement and hence make the respective list visible.

```
editor for concept State
  node cell layout:
    [-
    state { name } {
      ?[- entry % entryAction % -]
      (- % transitions % /empty cell: * R/O model access * -)
      ?[- exit ?% exitAction % -]
    }
    -]
```

Figure 1.10: The definition of the editor for the `State` concept. In MPS, editors are made from cells. In the editor definition your arrange the cells and define what they project; this defined the projection rule that is then used when instances of the concept are edited.

Fig. 1.11 shows the definition of the editor for a `Transition`. It arranges the keyword `on`, the trigger, the guard condition, target state and the actions in a horizontal list of cells, the guard surrounded by brackets, and an arrow (`->`) in front of the target state. The editor for the actions `StatementList` comes with its own set of curly braces.

```
editor for concept Transition
  node cell layout:
    [- on % trigger % [ % guard % ] -> ( % targetState % -> { name } ) % actions % -]
```

Figure 1.11: The editor for transitions. Note how we embed the guard condition expression simply by referring to the `guard` child relationship. We "inherit" the syntax for expressions from the `com.mbeddr.core.expressions` language.

The `%targetState% -> {name}` part is interesting; it expresses that in order to render the target state, the target's state's `name` attribute should be shown. We could use any text string to refer to the target state[43].

Note how we use `on` both as the leading keyword for a transition

[43] We could even use the symbol `X` to render *all* target state references. The reference would still work, because the underlying data structure uses the target's unique ID to establish the reference. It does not matter what we use to represent the target in the model. Using `X` for all references would of course be bad for human readability, but technically it would work

and as the alias. This way, if a user types the `on` alias to instantiate a transition, it feels as if she would just type the leading keyword of a transition (as in a regular text editor).

If a language extension defined a new concept `SpecialKindOfTransition`, they could use another alias to uniquely identify this concept in the code completion menu. When the user enters a transition, he has to decide which alias to use, depending on whether he wants to instantiate a `Transition` or a `SpecialKindOfTransition`. Alternatively, the `SpecialKindOfTransition` could use *the same alias* `on`. In this case, if the user types `on`, the code completion menu pops open and the user has to decide which of the two concepts to instantiate. As we have discussed above, this means that there is never an ambiguity that cannot be handled — as long as the user is willing and able to make the decision which concept should be instantiated.

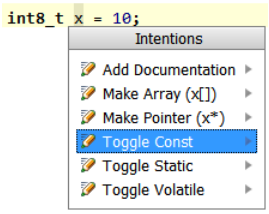


Figure 1.12: The intentions menu for a local variable declaration. It can be opened via `Alt-Enter`. Note that, to select an action from the menu, you can just start typing the action label, so this is very keyboard-friendly.

■ *Intentions* Intentions are MPS' term for what is otherwise known as a Quick Fix: a little menu can be popped up on a program element that contains a set of actions that typically change the underlying program (see Fig. 1.12). In MPS the intentions menu is opened via `Alt-Enter`. In MPS, intentions play an important role in the editor. Some changes to the program can *only* be made via an intention[44]. For example, in the previous section we mentioned that we use them to add entry actiom statements to a `State`. Here is the intention code:

[44] This is mostly because building a just-type-along solution would be a lot of work in a projectional editor.

```
intention addEntryActions for concept State {
  error intention : false
  available in child nodes : true

  description(editorContext, node)->string {
    "Add Entry Action";
  }

  isApplicable(editorContext, node)->boolean {
    node.entryAction.isNull;
  }

  execute(editorContext, node)->void {
    node.entryAction.set new(<default>);
    editorContext.selectWRTFocusPolicy(node.entryAction);
  }
}
```

An intention is defined for a specific language concept (`State` in the example). It can then be invoked by pressing `Alt-Enter` on any instance of this concept. Optionally it is possible to also make it available in child nodes. For example, if you are in the guard expression of an transition, an intention for `State` with `available in child nodes` set to `true` will be available as well. The intention implementation also specifies an expression used as the title in the menu and an applicability condition. In the example the intention is only applicable if the corresponding state does not yet have any entry action. Finally, the

`execute` section contains procedural code that performs the respective change on the model. In this case we simply create a new instance of `StatementList` in the `entryAction` child. We also set the cursor into this new `StatementList`.

Notice how we don't have to specify any formatter or serializer for our language. Remember how a projectional editor *always* goes from AS to CS. So after changing the AS procedurally, the respective piece of the tree is simply rerendered to update the representation of the program in the editor.

■ *Expressions* Since we inherit the expression structure and syntax from the C core language, we don't have to define expressions ourselves so we can use them in guards. It is nonetheless interesting to look at their implementation in `com.mbeddr.core.expressions`.

Expressions are arranged into a hierarchy starting with the abstract concept `Expression`. All other kinds of expressions extend `Expression`, directly or indirectly. For example, `PlusExpression` extends `BinaryExpression` which in turn extends `Expression`. `BinaryExpressions` have `left` and `right` child `Expressions`. This way, arbitrarily complex expressions can be built. Representing expressions as trees is a standard approach; in that sense, the abstract syntax of MPS expressions is not very interesing. The editors are also trivial — in case of the + expression, they are a horizontal list of: editor for `left` argument, the + symbol, and the editor for the `right` argument.

As we have explained in the general discussion about projectional editing , MPS supports linear input of hierarchical expressions using so-called side transforms. The code below shows the right side transformation for expressions that transforms an arbitrary expression into a `PlusExpression` by putting the `PlusExpression` "on top" of the current node.

```
side transform actions makeArithmeticExpression

  right transformed node: Expression tag: default_

  actions :
    add custom items  (output concept: PlusExpression)
      simple item
        matching text
          +
        do transform
          (operationContext, scope, model, sourceNode, pattern)->node< > {
            node<PlusExpression> expr = new node<PlusExpression>();
            sourceNode.replace with(expr);
            expr.left = sourceNode;
            expr.right.set new(<default>);
            return expr.right;
          }
```

Using the alias (i.e. the operator symbol) of the respective `BinaryExpression` and the inheritance hierarchy, it is possible to factor all side transfor-

mations for all binary operations into one single action implementation, resulting in much less implementation effort.

The fact that you can enter expressions linearly, leads to a problem not unlike the one found in grammars regarding operator precedence. If you enter `2 + 3 * 4` by typing these characters sequentially, there are two ways how the tree could look, depending on whether + or * binds more tightly[45].

[45] Note how this really is a consequence of the linear input method; you could build the tree by first typing the + and then filling in the left and right arguments, in which case it would be clear that the * is lower in the tree and hence binds tighter. However, this is tedious and hence not an option in practice.

To deal with this problem, we proceed as follows: each subconcept of `BinaryExpression` has a numerical value associated with it that expresses its precedence. The higher the number, the higher the precedence (i.e. the lower in the tree). The action code shown above is change to include a call to a helper function that rearranges the tree according to the precedence values.

```
do transform
  (operationContext, scope, model, sourceNode, pattern)->node< > {
    node<PlusExpression> expr = new node<PlusExpression>();
    sourceNode.replace with(expr);
    expr.left = sourceNode;
    expr.right.set new(<default>);
    PrecedenceHelper.rearrange(expr);
    return expr.right;
  }
```

This helper function scans through an expression tree and checks for cases where a binary expression with a higher precedence is an ancestor of a binary expression with a lower precedence value. If it finds one, it simply rearranges the tree to resolve the problem. Since the problem can only arise as a consequence of the linear input method, it is sufficient to include this rearrangement in the side transformation shown above.

■ *Context Restrictions* MPS makes strong use of polymorphism. If a language concept defines a child relationship to another concept `C`, then any subtype of C can also be used in this child relationship. For example, a function has a `body` which is typed to `StatementList`, which contains a list of `Statements`. So every subtype of `Statement` can be used inside a function body. In general, this is the desired behavior, but in some cases, it is not. Consider test cases. Here is a simple example:

```
module UnitTestDemo from cdesignpaper.unittest imports nothing {

  test case testMultiply {
    assert(0) times2(21) == 42;
  }

  int8_t times2(int8_t a) {
    return 2 * a;
  }
}
```

Test cases are defined in a separate language `com.mbeddr.core.unittest`. The language defines the `TestCase` concept, as well as the `assert` statement. `AssertStatement` extends `Statement`, so by default, an `assert` can be used wherever a `Statement` is expected, once the `com.mbeddr.core.unittest` is used in a program. However, this is not what we want: `assert` statements should be restricted to be used inside a `UnitTest`[46]. To support such a use case, MPS supports a set of constraints. Here is the implementation for `AssertStatement`:

[46] This is, among other reasons, because the transformation of the `assert` statement to C expects code generated from the `UnitTest` to surround it

```
concepts constraints AssertStatement {
  can be child
    (operationContext, scope, parentNode, link, childConcept)->boolean {
      parentNode.ancestor<concept = TestCase, +>.isNotNull;
    }
}
```

This constraint checks that a `TestCase` is among the ancestors of a to-be-inserted `AssertStatement`. The constraint is checked *before* the new `AssertStatement` is inserted and *prevents* insertion if not under a `TestCase`. The constraint is written from the perspective of the potential child element.

For reasons of dependency management, it is also possible to write the constraint from the perspective of the parent or an ancestor. This is useful if a new container concept wants to restrict the use of *existing* child concepts without changing those concepts. For example, the `Closure` concept, which contains a statement list as well, prohibits the use of `LocalVariableRefs`, in any of its statements.

2
Scoping and Linking

Linking refers to the resolution of name-based references to the references 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. The set of visible elements is called the scope. In this chapter we discuss the topic in general and then provide examples with Spoofax, Xtext and MPS.

As we have elaborated in the previous section, 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-references. Examples abound and include variable references, procedure calls and target states in transitions of state machines. 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 where every program element has a unique ID, a reference is simply 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 reference. Of The code completion menu shows some human-readable (qualified) name of the target, but the persisted program uses the unique ID once the user makes a selection.

Typically, a language's structure definition specifies which concepts constitute valid target concepts for references (e.g., a `Function`, a `Variable`, or a `State`), but this is usually too imprecise. Language-specific visibility rules determine which *instances* of these concepts are actually permitted as a reference target. For example, only the function and variables *in the local module* or the states *in the same state machine as the transition* may be valid targets.

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 cross-reference, 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 can follow or mimic the structure of the language itself: e.g., the innermost scope of a reference consists of all the elements present in the directly-encompassing "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".

Instead of looking at scopes from the perspective of the reference (and hence calculating a set of candidate target elements), one can also look at scopes from the perspective of visibility. In this case, we (at least conceptually) compute for each location in the program, the set of visible elements. A reference is then restricted to refer to any element from those visible at the particular location. Our notion is more convenient from the cross-reference viewpoint, however, as it centers

around resolving particular cross-references one at a time. From an implementation point of view, both perspective are exchangable.

2.1 Scoping in Spoofax

In Spoofax, each language concept has its own private namespace. For example, functions may live in the `Functions` namespace and properties may live in the `Property` namespace. Namespaces can be defined as part of a grammar definition by adding annotations to productions. Spoofax supports three different forms of annotations: annotations for language elements that define something in a namespace, for elements that reference something in a namespace, and for block constructs.

2.1.1 Definitions and References

In the previous chapter we described how to specify a grammar for a subset of the mobl language. This chapter shows how to add annotations and rules to define name resolution for this language.

To understand naming in Spoofax, the notion of a *namespace* is essential. A namespace is a collection of names. A namespace is not necessarily connected to a specific language concept. Different concepts can contribute names to a single namespace. During reference resolution, names are resolved against namespaces.

Language elements that contribute names to a a namespace can be annotated with `Namespace@=Symbol`, where `Namespace` is the namespace of the definition, and `Symbol` is the grammatical symbol that represents the name of the concept, and that is contributed to the namespace. As an example, we can add this annotation to the production for `Entity` definitions:

```
"entity" Type@=ID "{" EntityBodyDecl* "}" -> Decl {"Entity"}
```

The annotation indicates that every program element that corresponds to this production defines a name in the `Type` namespace. It also indicates the name of the program element: the identifier `ID` that follows after the `entity` keyword.

Note how Spoofax distinguishes the name of a namespace from the type of a program element: the type of the program element above is `Decl`, but it lives in the `Type` namespace. By distinguishing these two things, it becomes easy to add or exclude program elements from a namespace. For example, `import` statements are also of syntactic type `Decl`, but they do not live in the `Type` namespace. Likewise, we

could add the primitive integer and string types to the `Type` namespace, even though they are syntactically a very different concept from entity declarations.

References to definitions are defined in a similar fashion, using the notation `Namespace@Symbol`:

```
Property@=ID ":" Type@ID -> EntityBodyDecl {"Property"}
```

The first annotation here says that the production defines something in the `Property` namespace, while the second says that the `ID` after the colon references something in the `Type` namespace.

2.1.2 *Basic Scoping Rules*

Scoping annotations[1] define the block structures of a language. Block structures can be nested and determine the visibility of names. For example, functions are block structures that can contain local variable definitions that are not visible from outside the function. Internally, the local variables are then assigned a qualified name based on the name of the function. The same principle applies to fields in a Java class, properties in a mobl entity, and so on. Block structures can be specified by adding `scope(Namespace)` at the end of a production:

[1] Unfortunately the term *scope* is used in a slightly different way in Spoofax.

```
"entity" Type@=ID "{" EntityBodyDecl* "}" -> Entity {"Entity", scope(Property)}
```

Here, the `scope` annotation indicates that every entity scopes the properties inside it. These scopes are used to qualify the names of every property inside an entity:

```
entity Customer {
  name : String // Customer.name
}

entity Product {
  name : String  // Product.name
}
```

In this example, the two `name` properties both live the `Property` namespace, but we can still distinguish them: one can be reached as `Customer.name`, the other as `Product.name`. This means that if `name` is referenced in a function inside `Customer`, then it references the first one, not the one in `Product`[2]. The full grammar of the mobl entities language with annotations is the following (leaving out the lexical syntax):

[2] Note that Spoofax does not require each name inside a block structure to be unique: by default, it is not an error if users add an additional property called name to `Product`. However, in most languages it is desirable to prevent assigning duplicate (qualified) names. We show how to specify a constraint for that later on .

```
module MoblEntities
```

```
context-free start symbols

  Module

context-free syntax

  "module" Module@=ID Decl* -> Module {"Module", scope(Entity)}
  "entity" Type@=ID "{" EntityBodyDecl* "}"
                            -> Decl {"Entity", scope(Property,Function)}
  "import" Module@ID        -> Decl {"import"}
  Property@=ID ":" Type@ID  -> EntityBodyDecl {"Property"}

  "function" Function@=ID "(" {Param ","}* ")" ":" Type@ID "{" Statement* "}"
                            -> EntityBodyDecl {"Function", scope(Var)}
  Var@=ID ":" Type@ID       -> Param {"Param"}
  "var" Var@=ID "=" Expr ";"  -> Statement {"Declare"}
  "return" Exp ";"          -> Statement {"Return"}

  Exp "." Function@ID "(" Exp  ")" -> Exp {"MethodCall"}
  Exp "." Property@ID           -> Exp {"FieldAccess"}
  Exp "+" Exp                   -> Exp {"Plus"}
  Exp "*" Exp                   -> Exp {"Mul"}
  _@ID                          -> Exp {"Var"}
  INT                           -> Exp {"Int"}
```

A special case in this grammar is the rule for `"Var"`, which can refer to multiple namespaces (the underscore represents a wildcard). Consider the statement `return x.y;`. In such a statement, `x` can be a local variable (`Var`) or a local property (`Property`). In the full mobl language, it can even be a type (`Type`). As this more complicated scenario is not covered by the annotations, we specify an `_` in the grammar instead. The `_` indicates that we specify additional, manual rules for resolving this case. We show how to specify these at the end of this section.

2.1.3 URIs for Program Elements

Each program element that defines something is assigned a URI. The URI uniquely identifies the element across a project. By default, these URIs are constructed automatically, based on the namespace and scope annotations in the grammar. As an example, consider the following entity.

```
module storage

entity Store {
  name    : String
  address : Address
}
```

Following the annotated mobl grammar, there are two block constructs in this fragment: one at the module level and one at the entity level. We can assign names to these blocks (`storage` and `Store`) by using the namespace definition annotations. By creating a hierarchy of these names, Spoofax creates URIs: the URI for `Store` is `Type://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 names[3]:

[3] The reverse order used in the representation makes it easier to efficiently store and manipulate URIs in memory: every tail of such a list can share the same memory space.

```
[Property(), "name", "Store", "storage"]
```

In most cases, parts of a path are simple strings, such as `"Store"`, but they can be complex terms if needed. An example is *anonymous* blocks that do not have a name associated with them. Block statements in C-like languages are an example of this. These scopes are handled differently from named scopes. We can add a block statement to our grammar by adding the following production:

```
"{" Stm* "}" -> Stm {"Block", Scope(Var)}
```

The block statement does not define a definition name, but can still become part of a URI. Instead of a name, that part will then have the form `Anon(n)` where `n` is a generated number that identifies the block. `Anon(n)` is an example of a complex term.

URIs can be inspected using the default hover help popups, or using the analyzed syntax view. The analyzed syntax view shows the abstract syntax with all URIs as annotations. Consider the following example with both named and anonymous blocks:

```
module banking

entity BankAccount {
  name   : String
  number : Num

  function toCapitals() : String {
    { // anonymous block
      var result = name.toUpperCase();
      return result;
    }
  }
}
```

The analyzed abstract syntax for this example is the following:

```
Module(
  "banking"{[Module(),"banking"]},
  [ Entity(
      "BankAccount"{[Type(),"BankAccount","banking"]},
      [ Property(
          "name"{[Property(),"name","BankAccount","banking"]},
          "String"{[Type(),"String","mobl"]}
        ),
        Property(
          "number"{[Property(),"number","BankAccount","banking"]},
          "Num"{[Type(),"Num","mobl"]}
        ),
        Function(
```

```
          "toCapitals"{[Function(),"toCapitals","BankAccount","banking"]},
          [],
          "String"{[Type(),"String","mobl"]},
          Block([
            Declare(
              "result"{[Var(),"result",Anon(125),"toCapitals","BankAccount","banking"]},
              MethodCall(..., ...{[Function(),"toUpperCase","String","mobl"]})
            ),
            Return(
              Var("result"{[Var(),"result",Anon(125),"toCapitals","BankAccount","banking"]})
            )
          ])
        )
      ]
    )
  ]
)
```

The annotations indicate the URIs of each definition and reference. They can be used to get an unrefined view of all URIs at a glance, but also have a role in transformations on abstract syntax, as we discuss later.

Any references that cannot be resolved are annotated with a special `Unresolved` constructor. For example, a variable `nonexistent` could be represented as `Var("nonexistent"{[Unresolved(Var()),"nonexistent",...]})`. This makes it easy to recognize any unresolved references in analyses or transformations.

2.1.4 *Persistence of Naming Information*

Spoofax stores all definitions and references in an in-memory data structure called the index[4]. By collecting all this summary information about files in a project together, it ensures fast access to global information. The index is updated automatically with changes to the file system (e.g., files being deleted or removed) and is persisted as Eclipse exits. All entries in the index have a URI as we showed previously. Index entries can be represented in the ATerm format (see). For example, `Def([Module(),"banking"])` is a definition entry for a banking module. In the form of terms, they can be used in the same way as abstract syntax tree fragments in transformations and analyses. Chapter 3 shows how the index can be used for such tasks.

Internally, index entries are stored in tables for efficient random access. They also contain meta-data such as the file name and line number of the definition. With this meta-data, Spoofax can provide editor services such as reference resolving. Note that there is no requirement that file names correspond to the URIs: a `banking.Bank` entity could be defined in any file, as long as there are no constraints in the language that prevent that.

[4] It can also store information about definitions, such as type information, as we show in the next chapter.

2.1.5 *The Default Name Resolution Strategy*

The structure of URIs forms the basis of the default name resolution strategy of Spoofax[5]. Consider the `address` property in the `Store` entity:

[5] The default strategy can be customized, but we first take a closer look at how name resolution works just based on the annotations in the grammar.

```
module storage

entity Store {
  name    : String
  address : Address
}
```

The `address` property references something named `Address`. According to the grammar, that something lives in the `Type` namespace. Spoofax resolves this name by looking at the context: is there an `Address` in `Type://storage.Store`? If not, it will keep trying the parent URI: `Type://storage` and finally `Type://`. If it is successful in finding it this way, it can resolve the reference, otherwise it displays an error marker in the editor.

The default resolution strategy is designed to be generic and makes few assumptions about a specific language[6]. By default, it does not care about file boundaries: if an `Address` is defined in a different file, it will still find it. There is also no default constraint for duplicate names: two definitions of `Address` are fine (in which case the reference has two targets). Lastly, the default strategy knows nothing about import definitions for a specific language, as these can vary quite a bit among languages.

[6] Spoofax expects language designers to *manage by exception*.

Spoofax provides support for adding constraints and for customization of the behavior of the default resolution strategy by means of *rewrite rules*. Rewrite rules are functions that operate on terms, transforming one term to another. We provide a primer on rewrite rules next, and show how they can be used to customize name resolution[7].

[7] Rewrite rules are used for all kinds of other purposes in Spoofax, and we will encounter them again, for example in the chapter on transformation and code generation . This is why we explain them in quite some detail here.

2.1.6 *Rewrite Rules*

Rewrite rules in Spoofax are provided as part of the Stratego program transformation language. A basic rewrite rule that transforms a term pattern `term1` to a term pattern `term2` has the following form:

```
rule-name:
  term1 -> term2
```

Term patterns have the same form as terms: any term is a legal term pattern. In addition to the basic constructors, string literals, integer literals, and so on, they also support variables (e.g., `v` or `name`) and

wildcards (indicated by _). As an example, the following rewrite rule rewrites an `Entity` to the list of properties contained in that entity:

```
get-properties-types:
  Entity(name, properties) -> properties
```

So, for an entity `Entity("User", [Property("name", String)])`, it binds `"User"` to the variable `name`, and `[Property("name", "String")]` to the variable `properties`. It then returns `properties`. While rewrite rules can be viewed as functions, they have one important difference: they can be defined multiple times for different patterns. In the case of `get-properties`, we could add another definition that works for property access expressions:

```
get-properties:
  FieldAccess(expr, property) -> property
```

Rules can have complex patterns. For example, it is possible to write a rule that succeeds only for entities with *only* a `name` property[8]:

[8] Note how this rule uses a wildcard since it doesn't care about the name of the entity.

```
is-name-only-entity:
  Entity(_, [Property("name", "String")]) -> True()
```

Rewrite rules can be invoked using the syntax `<rule-name> term`. For example, `<get-properties> Entity("Unit", [])` would return an empty list of properties. The angle brackets make it easy to distinguish rule invocations from terms, and makes it possible to use invocations in term expressions.

Stratego provides a `with` clause that can be used for additional code that should be considered for rewrite rules. The `with` clause is most commonly used for assignments and calls to other rules. As an example, we can write the rule above using a `with`. This rule assigns the value of `get-properties` to a variable `result` and returns that as the result value of the rule:

```
invoke-get-properties:
  Entity(name, properties) -> result
  with
    result := <get-properties> Entity(name, properties)
```

Rules can also have conditions. These can be specified using `where`[9]. These clauses typically use the operators listed in Fig. 2.1. An example of a rule with a `where` clause is the following:

[9] If the pattern of a rule does not match, or if its conditions do not succeed, a rule is said to *fail*.

Expression	Description
`<e> t`	Applies `e` to `t`, or fails if `e` is unsuccessful.
`v := t`	Assign a term expression `t` to a variable `v`.
`!t => p`	Match a term `t` against a pattern `p`, or fail.
`not(e)`	Succeeds if `e` does not succeed.
`e1; e2`	Sequence: apply `e1`. If it succeeds, apply `e2`.
`e1 <+ e2`	Choice: apply `e1`, if it fails apply `e2` instead.

Figure 2.1: Operators in Stratego expressions.

```
has-properties:
  Entity(name, properties) -> True()
  with
    properties := <get-properties> Entity(name, properties);
  where
    not(!properties => [])
```

This rule only succeeds for entities where the condition `not(!properties => [])` holds. That is, it succeeds as long as an entity does not have an empty list (indicated by `[]`) of properties. Rewrite rules can have any number of `where` and `with` clauses, and they are evaluated in the order they appear.

Like functions or methods in other languages, rewrite rules can have parameters. Stratego distinguishes between parameters that pass other rules and parameters that pass terms, using a vertical bar to separate them two separate lists. Rules that take both rule and term parameters have a signature of the form `rule(r|t)`, those with only rule parameters use `rule(r)`, and those with only term parameters use `rule(|t)`. The Stratego standard library provides a number of higher-order rules, i.e. rules that take other rules as their argument. These rules are used for common operations on abstract syntax trees: for example, `map(r)` applies a rule `r` to all elements of a list:

```
get-property-types:
  Entity(_, properties) -> types
  with
    types := <map(get-property-type)> properties

get-property-type:
  Property(_, type) -> type
```

Rules like `map` specify a *traversal* on a certain term structure: they specify how a certain rule should be applied to a term and its subterms. Rules that specify traversals are also called *strategies*[10]. In Spoofax, strategies are used to control traversals in constraints, transformation, and code generation.

[10] This is where the name of the *Stratego* transformation language comes from.

2.1.7 *Customizing the Name Resolution Strategy*

To customize the name resolution strategy, Spoofax language definitions can include `adjust-index-lookup` rules. These rules have a spe-

cific signature, and are invoked by the name resolution strategy for each reference that is resolved[11]. As an example, consider how a reference to a local property is resolved in mobl:

[11] This is a rewriting rule with a predefined name and signature so it can be invoked by Spoofax. It can be redefined to apply to specific langauage concepts.

```
entity BankAccount {
  function getNumber() : number {
    return number;
  }

  name   : String
  number : Num
}
```

Here, the function `getNumber` references the local `number` property. The production rule we previously specified for these kinds of references used an underscore to indicate that we would manually specify how to resolve it[12]:

[12] Because properties and variables can be referenced.

```
_@ID -> Exp {"Var"}
```

So, we need to specify a `adjust-index-lookup` rule for the variables:

```
adjust-index-lookup(target |namespace, path, name):
  Var(x) -> adjusted-uris
  where
    <target> x
  with
    adjusted-uris := ...
```

Each `adjust-index-lookup` rule has this exact signature: it gets a `target` rule argument, and term arguments with the `namespace`, `path`, and *name* of the reference to be resolved. On the left-hand side, it specifies what construct it matches against. In this case, we want to resolve a variable. Next, we need to specify which part of the node is the name we want to resolve. [13] In a variable reference `Var(x)`, the `x` is the name we want to resolve. We specify this explicitely as the first condition of the rule: `where <target> x`. On the right-hand side, the rule returns a list of adjusted URIs. Spoofax will then use these URIs to further resolve the element.

[13] There might be nodes with more than one name to resolve, for example in a qualified package name, each segment refers to a package.

A variable might refer either to a variable (namespace `Var`) or a property (namespace `Property`). Since we got the current `path` of the variable as a parameter, we can construct two possible URIs, where the definition site can be found: `Var://path` and `Property://path`[14]:

[14] Remember that URIs are represented as lists. We use the namespace as the head of the list and the path as the tail. The variable name is not part of these URIs.

```
adjust-index-lookup(target |namespace, path, prefix):
  Var(x) -> adjusted-uris
  where
    <target> x
```

```
with
  var-uri       := [Var() | path];
  property-uri  := [Property() | path];
  adjusted-uris := [var-uri, property-uri]
```

Note, that the URIs do not contain the variable name `x`. This is, because the URIs specify containers telling Spoofax where to look for the variable. We return both URIs in a list of adjusted URIs. The order in this list is important. Spoofax will first look only for variables, before it will look for properties.

The example code we have seen so far is pretty verbose for explanation. A typical shorter way to adjust variable lookup, would look like this:

```
adjust-index-lookup(target |namespace, path, prefix):
  Var(<target>) -> [[Var() | path], [Property() | path]]
```

Here, the `target` is marked in the left-hand side of the rule. On the right-hand side, we construct the list of adjusted URIs right away.

2.2 *Scoping in Xtext*

Xtext uses Java code for implementing all aspects of languages except the grammar[15]. Language developers implement various classes that implement Xtext-provided interfaces and then contribute those to Xtext using Google Guice, a dependency injection framework[16]. A lot of functionality is provided out-of-the-box with minimal configuration, but it's easy to swap out specific parts by binding another or a custom class through Guice.

[15] Most likely, some of these aspects will be implementable in Xtend2 in the future.

[16] http://code.google.com/p/google-guice/

2.2.1 *Simple, Local Scopes*

To implement scopes, language developers have to contribute a class that implements the `IScopeProvider` interface. It has one method 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 method takes the `EReference` (which identifies the reference for which the scope that needs to be calculated) as well as the current instance of the language concept whose reference should be scoped.

```
public interface IScopeProvider {
    IScope getScope(EObject context, EReference reference);
}
```

To make the scoping implementation easier, Xtext provides so-called 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>: we are trying to scope the <R> reference of the <X> concept
public IScope scope_<X>_<R>(<X> ctx, EReference ref );

// <X>: the language concept we are looking for
// <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 as follows:

```
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 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*. So we could write an even more general alternative, which scopes the visible `States` from anywhere in a `CoolingProgram`, independent of the actual reference[17].

```
public IScope scope_State(CoolingProgram ctx, EReference ref) {
    ...
}
```

[17] Depending on the structure of your language, Xtext may have a hard time finding out the current location, and hence, the reference that needs to be scoped. In this case, the tighter versions of the scoping method (`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, ... )`. It is a good idea to always use the most general variants, unless you

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 `States` defined in that `CoolingProgram`. Here is a possible implementation:

```
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 we use will use the `name` of the target element as the text by which it will be referenced[18]. So if a state is called `normalCooling`, then we'd have to write `state normalCooling` in a `ChangeStateStatement`. The text `normalCooling` acts as the reference — pressing `Ctrl-F3` on that program element will go to the referenced state.

[18] You can pass in code that creates other strings than then name from the target element.

2.2.2 *Nested Scopes*

The approach to scoping shown above is suitable for simple cases, such as the `targetState` reference shown above. However, in languages with nestes 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 `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, certain 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. The symbols in outer blocks are either not accessible at all, or a special name has to be used, for example, by prefixing them with some `outer` keyword (as in `outer.x`).

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 element[19]. The string is used as the reference text. By default, the string is the same as the target element's `name`. 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:

[19] In addition, the text shown in the code completion window can be different from the text that will be used as the reference once an element is selected. In fact, it can be a rich string that includes formatting, and it can contain an icon.

```
// 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 )
}
```

2.2.3 *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 explicity import file `B` to make the elements in `B` available as reference targets. This resulted

in several problems. First, for internal reasons, scalability was limited. Second, as a consequence of the explicit file imports, if the referenced element was moved into another file, the import statements in all referencing files had to be updated.

Since Xtext 1.0 both of these problems are solved using the so-called index[20]. 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. 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 loaded[21]. The index is updated whenever a file is saved, or a rebuild is performed. 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.

[20] This is similar to Spoofax'es index discussed above

[21] This is what improved scalability; files are only loaded if a reference target is accessed, not to check a reference for validity

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 referencable. 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 configured[22], which provides the possibility to reference model elements outside of the current file either through their (fully) qualified name or through their unqualified name using an `import` statement[23].

[22] The specific implementation is configured through a Guice binding.

[23] That `import` statement is different from the one mentioned earlier: it makes the contents of the respective namespace visible, it does not refer to the a particular file.

■ *Polymorphic References* In the cooling language, expressions also include references to various other entities, such as configuration parameters, variables and hardware elements (compressors or fans defined in a different model). All of these referenceble elements extend the `SymbolDeclaration` meta class. This means that all of them can be referenced by the single `SymbolRef` construct.

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

The problem with this situation is that the reference itself does not encode the kind of thing that is referenced. By looking at the refer-

ence alone we only know that we reference some kind of symbol. 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 sym eference. A more natural design of the language would use differ eference constructs for the different referencable elements. In this case, the reference itself is specific to the referenced element, making proc much easier[24]:

[24] It would also r riting the scopes and extending th guage simpler

```
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. 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 referencable elements extend `SymbolDeclaration` and use a single reference concept, as shown above.

2.3 *Scoping in MPS*

Making references work in MPS requires several ingredients. First of all, developers define a reference as part of the language structure. Then, an editor is defined that determines how the referenced element is rendered at the referencing site[25]. To determine which instances of the referenced concept are allowed, a scoping function has to be implemented. It simply returns a list of all the elements that are considered valid targets for the reference, as well a an optional text string used to represent the respective element in the code completion menu.

[25] We have shown this in the previous chapter.

As we have explained above , 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 function `search scope` 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 simply 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 hierarchy until it finds a `Statemachine` and then returns its `states`[26].

[26] The code used to express scopes can be arbitrarily complex and is implemented in MPS' BaseLanguge, an extended version of Java.

```
link {target}
  referent set handler:
    <none>
  search scope:
    (model, scope, referenceNode, linkTarget, enclosingNode)
     ->join(ISearchScope | sequence<node<State>>) {
      enclosingNode.ancestor<concept = Statemachine>.states;
    }
  validator:
    <default>
  presentation :
    <no presentation>
;
```

Note that for a smart reference, where the refernece object is created only *after* selecting the target, the `referenceNode` argument is `null`! This is why we write the scope using the `enclosingNode` argument.

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`).

■ *Nested Scopes* In a more complex, block oriented language with nested scopes, a different implementation pattern is recommended:

- All program elements that contribute elements that can be referenced (such as blocks, functions or methods) implement an interface `IScopeProvider`.
- This interface provides a method `getVisibleElements(concept<> c)` 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`, until there is none anymore. 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:
    (model, scope, referenceNode, linkTarget, enclosingNode, operationContext)->join(ISearchScope | sequence<node<LocalVariableDeclaration>>) {
      // find the statement that contains this (future) local variable reference
      node<Statement> s = enclosingNode.ancestor<concept = Statement, +>;

      // find the first containing ILocalVariableScopeProvider; this is
      // typically next next higher statement that owns a StatementList, such
      // as a ForStatement or an IfStatement
      node<ILocalVarScopeProvider> scopeProvider = enclosingNode.ancestor<concept = ILocalVarScopeProvider, +>;

      // if 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 create the scope and get the visible 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 expressions:

```
var int a;
var int b;
```

```
statemachine linefollower {
  in event initialized(int timestamp);
  states {
    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: simply a text string that represents the name of the respective target element. In Xtext, this has to be implemented with a single reference concept, typically called `SymbolReference`, that can reference to any kind of `Symbol`. `LocalVariableDefintions` and `EventParameters` would both extend `Symbol`, and scopes would make sure both kinds are visible from within guard expressions. The problem with this approach is that the reference itself contains no type information about what it references, it is simply a `SymbolReference`. Processing code has to inspect the type of the referenced symbol to find out what a particular `SymbolReference` actually means[27].

In projectional editors 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 simply renders the name of the referenced element, and each of them has `their own` scope definition! So adding new kinds of references to existing expression languages can be done in a modular fashion, since the new reference expression comes with its own, independent scoping rule. The following is the respective code for the `EventArgRef` expression:

[27] It can also be a problem regarding modularity, because every referencable concept must extend `Symbol`. Referencable elements contributed by an independently developed language which we may want to embed into the C language will *not* extend `Symbol`, though! We discuss language modularization and composition in .

```
concept EventArgRef extends Expression // this is the Expression concept from C

...

link {arg}
  search scope:
    (model, scope, referenceNode, linkTarget, enclosingNode, operationContext)
                              ->join(ISearchScope | sequence<node<EventArg>>) {
      enclosingNode.ancestor<concept = 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 where there's a `LocalVariable` and a `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 `LocalVariable` or to an `EventParameter`, because the reference is encoded using the ID of the target.

3
Constraints

Constraints are Boolean expressions that must be true for every instance of a language. Together with type systems, which are discussed in the next chapter, they ensure the static semantics of a language. The chapter introduces the concept, some considerations regarding languages suitable for expressing constraints and provides examples with Xtext, MPS and Spoofax.

Constraints are Boolean conditions that have to evaluate to `true` in order for the model to be correct ("does `expr` hold?"). An error message is reported if the expression evaluates to `false` ("`expr` does not hold!"). Constraints are typically associated a particular language concept ("for each instance of `x`, `expr-with-x` must hold"). Constraints address model correctness beyond syntax and scoping/linking, but short of actual execution semantics. Typical examples for constraints are:

- uniqueness of names in lists of elements (e.g. functions in a namespace);
- every non-start state of a state machine has at least one incoming transition;
- a variable is defined before it is used (statement ordering);
- the assignee is type-compatible with the right hand side expression.

The last item is an example of a type system rule: these verify the correctness of types in programs, e.g. they make sure you don't assign a `float` to an `int`. Particularly in expression languages, type calculation and checking can become quite complicated and therefore warrants special support. This is why we distinguish between simple constraints (covered in this chapter) and type systems (which we cover in the next chapter).

Constraints can be implemented with any language or framework that is able to query a model and report errors to the user. To make constraint checking efficient, it is useful if the language has the following characteristics:

- It should be able to effectively navigate and filter the model. Support for path expressions (as in `aClass.operations.arguments.type` as a way to find out the types of all arguments of all operations in a class) is extremely useful.
- Support for higher-order functions is useful so one can write generic algorithms and traversal strategies
- A good collection language, often making use of higher-order functions, is very useful, so it is easily possible to filter collections, create subsets or get the set of distinct values in a list.
- Finally, it is helpful to be able to associate a constraint declaratively with the language concept for whose instances it should be executed

Here is an example constraint written in a pseudo-language:

```
constraint for:
  Class
expression:
  this.operations.arguments.type.filter(ComplexNumber).isNotEmpty &&
  !this.imports.any(i|i.name == "ComplexNumberSupportLib")
message:
  "class "+this.name+" uses complex numbers, so the ComplexNumberSupportLib must
  be imported"
```

Some kinds of constraints require specialized data structures to be built or maintained in sync with the program. Examples include dead code detection, missing returns in some branches of a method's body or read access to an uninitialized variable. To be able to find these kinds of errors statically, a dataflow graph has to be constructed from the program. A dataflow graph models the various execution paths through a (part of a) program. Once a dataflow graph is constructed, it can be used to check whether there exists a path from program start to a variable read without coming across a write to the same variable. We show an example of using a data flow graph in the MPS example (Section 3.2).

3.1 *Constraints in Xtext*

Just like scopes, constraints are implemented in Java. Developers add methods to a validator class generated by the Xtext project wizard.

In the end, these validations plug into the EMF validation frameworkOther EMF `EValidator` implementations can be used in Xtext as well..

A constraint checking method is a Java method with the following characteristics: it is public, returns `void`, has any name, it has a single argument of the type for which the check should apply, and it has the `@Check` annotation. For example, the following method is a check that is invoked for all instances of `CustomState` (i.e. not for start states and background states). It checks that each such state can actually be reached by verifying that it has incoming transitions (expressed via a `ChangeStateStatement`):

```
@Check(CheckType.NORMAL)
public void checkOrphanEndState( CustomState ctx ) {
    CoolingProgram coopro = Utils.ancestor(ctx, CoolingProgram.class);
    TreeIterator<EObject> all = coopro.eAllContents();
    while ( all.hasNext() ) {
        EObject s = all.next();
        if ( s instanceof ChangeSt tement ) {
            ChangeStateStatement c ChangeStateStatement) s;
            if ( css.getTargetState == ctx ) return;
        }
    }
    error("no transition ever leads into this state",
     CoolingLanguagePackage.eINSTANCE.getState_Name());
}
```

The method retrieves the cooling program that owns the `ctx` state, then retrieves all of its descendants and iterates over them. If the descendant is a `ChangeStateStatement`, and if the `targetState` property of the `ChangeStateStatement` references the current state, then we return: we have found a transition leading into the current state. If we don't find one of these, we repo error. The `CheckType.NORMAL` in the annotation defines when this check should run:

- CheckType.NORMAL: run when the file is saved
- CheckType.FAST: run after each model change (i.e. after each keypress)
- CheckType.EXPENSIVE: run only if requested via the context menu

3.2 *Constraints in MPS*

3.2.1 *Simple Constraints*

MPS' approach to constraints is very similar to Xtext's. The main difference is that the constraint is written in `BaseLanguage`, which is an extended version of Java that has some of the features that makes constraints more concise. Here is the code for the same *state unreachable*

constraint, which we can make use of in the state machines extension to C:

```
checking rule stateUnreachable {
  applicable for concept = State as state
  do {
    if (!state.initial &&
        state.ancestor<concept = Statemachine>.
            descendants<concept = Transition>.
            where({~it => it.target == state; }).isEmpty) {
        error "orphan state - can never be reached" -> state;
    }
  }
}
```

Currently there is no way to control when a constraint is run, it is decided based on some MPS-internal algorithm. However, pressing `F5` in a program or explicitly running the model checker forces all constraints to be reevaluated.

3.2.2 *Dataflow*

The foundation for data flow analysis is the so-called data flow graph. This is a data structure that describes the flow of data through a program's code. For example, in `int i = 42; j = i + 1;` the `42` is "flowing" from the `init` expression in the local variable declaration into the variable `i` and then, after adding `1`, into `j`. Data flow analysis consists of two tasks: building a data flow graph for a program, and then performing analysis on this data flow graph to detect problems in the program.

MPS comes with predefined data structures for data flow graphs, a DSL for defining how the graph can be derived from language concepts (and hence, programs) and a set of default analyses that can be integrated into your language[1]. We will look at all these ingredients in this section[2].

[1] MPS also comes with a framework for developing custom analyses; however, this is beyond the scope of this book.

[2] To play with the data flow graph, you can select a method in a Java program and then use the context menu on the method; select `Language Debug -> Show Data Flow Graph`. This will render the data flow graph graphically and constitutes a good debugging tool when building your own data flow graphs and analyses.

■ *Building a Data Flow Graph* Data flow is specified in the *Dataflow* aspect of language definitions. There you can add data flow builders (DFBs) for your language concepts. These are programs expressed in MPS' data flow DSL that build the data flow graph for instances of those concepts in programs. Here is the DFB for `LocalVariableDeclaration`:

```
data flow builder for LocalVariableDeclaration {
  (node)->void {
    if (node.init != null) {
      code for node.init
      write node = node.init
    } else {
      nop
    }
  }
}
```

If the `LocalVariableDecaration` has an `init` expression (it is optional!), then the DFB for the `init` expression has to be executed. The `code for` statement calls the DFB for the node that is passed as its argument. Then we perform an actual data flow definition: the `write node = node.init` specifies that write access is performed on the current node. The statement also expresses that whatever value was in the `init` expression is now in the node itself. If there is no `init` expression, we still want to mark the `LocalVariableDeclaration` node as visited by the data flow builder — the program flow has come across this node. A subsequent analysis reports all program nodes that have *not* been visited by a DFB as dead code. So even if a node has no further effect on a program's data flow, it has to be marked as visited using `nop`.

To illustrate a `read` statement, we can take a look at the `LocalVariableRef` expression which read-accesses the variable it references. Its data flow is defined as `read node.var`, where `var` is the name of the reference that points to the referenced variable.

In an `AssignmentStatement`, we first execute the DFB for the `rvalue` and then "flow" the `rvalue` into the `lvalue` — the purpose of an assignment:

```
data flow builder for AssigmentStatement {
  (node)->void {
    code for node.rvalue
    write node.lvalue = node.rvalue
  }
}
```

For a `StatementList`, we simply mark the list as visited and then execute the DFBs for each statement in the list. We are now ready to inspect the data flow graph for a simple function. Fig. 3.1 shows the function and the graph.

```
void trivialFunction() {
  int8_t i = 10;
  i = i + 1;
} trivialFunction (function)
```

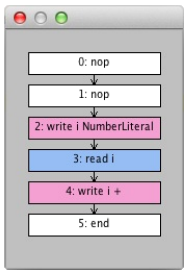


Figure 3.1: An example for a data flow for a simple C function. You can show the graph by select `Language Debug -> Show Data Flow Graph` from the context menu.

Most interesting data flow analysis has to do with loops and branch-

ing. So specifying the correct DFBs for things like `if`, `switch` and `for` is important. As an example, we look at the DFB for the `IfStatement`. We start with the obligatory `nop` to make the node as visited. Then we run the DFB for the condition, because that is evaluated in any case. Then it becomes interesting: depending on whether the condition is `true` or `false`, we either run the `thenPart` or we jump to where the `else if` parts begin. Here is the code so far:

```
nop
code for node.condition
ifjump after elseIfBlock // elseIfBlock is a label defined later
code for node.thenPart
{ jump after node }
```

The `ifjump` statement means that we may jump to the specified label (i.e. we then execute the `else if`s). If not (we just "run over" the `ifjmp`), then we execute the `thenPart`. If we execute the `thenPart`, we are finished with the whole `IfStatement` — no `else if`s or `else` parts are relevant, so we jump after the current node (the `IfStatement`) and we're done. However, there is an additional catch: in the `thenPart`, there may be a `return` statement. So we may never actually arrive at the `jump after node` statement. This is why it is enclosed in curly braces: this says that the code in the braces is optional. If the data flow does not visit it, that's fine (typically because we return from the method before we get a chance to execute this code).

Let's continue with the `else if`s. We arrive at the `elseIfBlock` label if the condition was `false`, i.e. the above `ifjump` actually happened. We then iterate over the `elseIf`s and execute their DFB. After that, we run the code for the `elsePart`, if there is one. The following code can only be understood if we know that, if we execute one of the `else if`s, then we jump *after the whole* `IfStatement`. This is specified in the DFB for the `ElseIfPart`, which we'll illustrate below. Here is the rest of the code for the `IfStatement`'s DFB:

```
label elseIfBlock
foreach elseIf in node.elseIfs {
  code for elseIf
}
if (node.elsePart != null) {
  code for node.elsePart
}
```

We can now inspect the DFB for the `ElseIfPart`. We first run the DFB for the condition. Then we may jump to after that `else if`, because the condition may be `false` and we want to try the next `else if`, if there is one. Alternatively, if the condition is `true`, we run the DFB for the body of the `ElseIfPart`. Then two things can happen: either

we jump to after the whole `IfStatement` (after all, we have found an `else if` that is `true`), or we don't do anything at all anymore because the current `else if` contains a `return` statement. So we have to use the curly braces again for the jump to after the whole `if`. The code is below, and an example data flow graph is shown on figure Fig. 3.2.

```
code for node.condition
ifjump after node
code for node.body
{ jump after node.ancestor<concept = IfStatement> }
```

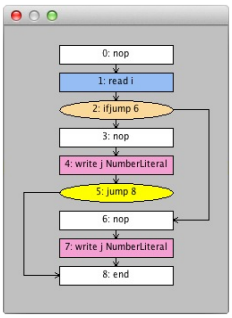


Figure 3.2: A data flow graph for an `if` statement `if ( i > 0 ) j = 1; else j = 2;`

The DFB for a loop make s use of the fact that loops can be represented using conditional branching. Here is the DFB for the `for` loop:

```
code for node.iterator
label start
code for node.condition
ifjump after node
code for node.body
code for node.incr
jump after start
```

We first execute the DFB for the `iterator` (which is a subconcept of `LocalVariableDeclaration`, so the DFB shown above works for it as well). Then we define a label `start` so we can jump to this place from further down. We then execute the `condition`. Then we have an `ifjmp` to after the whole loop (which covers the case where the condition is `false` and the loop ends). In the other case (where the condition is still `true`) we execute the code for the `body` and the `incr` part of the `for` loop. We then jump to after the `start` label we defined above.

■ *Analyses* MPS supports a number of data flow analyses out of the box. These analyses operate only on the data flow graph, so the same analyses can be used for any language, once the DFBs for that language map programs to data flow graphs. The following utility class uses the unreachable code analysis:

```
public class DataflowUtil {

  private Program prog;

  public DataflowUtil(node<> root) {
    prog = DataFlow.buildProgram(root); // build a program object and store it
  }

  public void checkForUnreachableNodes() {
    // grab all instructions that are unreachable (predefined functionality)
    sequence<Instruction> allUnreachableInstructions =
             ((sequence<Instruction>) prog.getUnreachableInstructions());
    // remove those that may legally be unreachable
    sequence<Instruction> allWithoutMayBeUnreachable =
            allUnreachableInstructions.where({~instruction =>
                !(Boolean.TRUE.equals(instruction.
```

```
                    getUserObject("mayBeUnreachable"))); });

    // get the program nodes that correspond to the unreachable instructions
    sequence<node<>> unreachableNodes = allWithoutMayBeUnreachable.
            select({~instruction => ((node<>) instruction.getSource()); });

    // output errors for each of those unreachable nodes
    foreach unreachableNode in unreachableNodes {
      error "unreachable code" -> unreachableNode;
    }
  }
}
```

The class constructs a `Program` object in the constructor. `Programs` are wrappers around the data flow graph and provide access to a set of predefined analyses on the graph. We will make use of one of them here in the `checkForUnreachableNodes` method. This method extracts all unreachable nodes from the graph (see comments in the code above) and reports errors for them. To actually run the check, we call this method from a `NonTypesystemRule` for C functions:

```
checking rule check_DataFlow {
  applicable for concept = Function as fct
  overrides false
  do {
    new DataflowUtil(fct.body).checkForUnreachableNodes();
  }
}
```

3.3 *Constraints in Spoofax*

Spoofax uses *rewrite rules* to specify all semantic parts of a language definition. In Section 2.1.6 we provide a primer on rewrite rules, and in this section we show how they can be used to specify constraints in language definitions.

3.3.1 *Basic Constraint Rules*

By convention, Spoofax uses rules by the name `constraint-error` to indicate constraints that trigger errors, `constraint-warning` for warnings, and `constraint-note` for notes. To report an error, warning or information note, these rules have to be overwritten for the relevant term patterns. The following example is created by default by the Spoofax project wizard. It simply reports a note for any module named `example`:

```
constraint-note:
  Module(name, _) -> (name, "This is just an example program.")
  where
    !name => "example"
```

The condition checks if the module `name` matches the string `"example"`[3] On its right-hand side, the rule returns a tuple with the tree node where the marker should appear and a string message that should be shown. All constraint rules have this form.

[3] Remember, `!x => y` matches a term `x` against a pattern `y`.

Most constraint rules use string interpolation for error messages. Interpolated strings have the form `$[...]` where variables can be escaped using `[...]`. The following example uses string interpolation to report a warning. A standard library rule `string-starts-with-capitals` is used[4].

[4] These and other library rules are documented on the Spoofax website at http://www.spoofax.org/.

```
constraint-warning:
  Entity(theName, _) -> (theName, $[Entity [theName] does not have a capitalized name])
  where
    not(<string-starts-with-capital> theName)
```

3.3.2 *Index-Based Constraint Rules*

Some constraint rules interact with the Spoofax index discussed in Section 2.1.4. Notable examples include constraints that forbid references to undefined program elements and duplicate definitions. Newly created Spoofax projects provide default constraint rules for these cases, which can be customized.

One way to interact with Spoofax's name resolution index and resolution algorithm is using the URI annotations on the abstract syntax. These are placed on each reference and definition. For example, a valid reference to a mobl variable `v` is represented in abstract syntax with an annotated term such as

```
Var("v"{[Var(),"v","function","module"]})
```

Without an annotation, the term reads `Var("v")`. The annotation is added directly to the name, surrounded with curly braces. The annotation itself is a URI `Var://module/function/v`, represented as a list consisting of the namespace, the name, and the path in reverse order.
Unresolved references are represented by terms such as the following. Notice the `Unresolved` term, surrounding the namespace:

```
Var("u"{[Unresolved(Var()),"u","function","module"]})
```

In most languages, references that cannot be statically resolved indicate an error. The following constraint rule reports an error for these cases:

```
constraint-error:
  x -> (x, $[Unable to resolve reference.])
  where
    !x => _{[Unresolved(t) | _]}
```

This rule matches any term `x` in the abstract syntax, and reports an error if it has an `Unresolved` annotation[5]. Note how the pattern `_{[Unresolved(t) | _]}` matches any term (indicated by the wildcard `_`) that has a list annotation where the head of the list is `Unresolved(t)` and the tail matches `_`.

[5] In dynamic languages, or languages with optional types, the constraint could be removed or relaxed. In those cases, name resolution may only play a role in providing editor services such as code completion.

In addition to annotations, the Spoofax index provides an API for inspecting naming relations in programs. Fig. 3.3 shows some of the key rules the index provides.

`index-uri`	Gets the URI of a term.
`index-namespace`	Gets the namespace of a term.
`index-lookup`	Returns the first definition of a reference.
`index-lookup-all`	Returns all definitions of a reference.
`index-get-files-of`	Gets all files a definition occurred in.
`index-get-all-in-file`	Gets all definitions for a given file path.
`index-get-current-file`	Gets the path of the current file.

Figure 3.3: API rules for working with the Spoofax index.

We can use the index API to detect duplicate definitions. In most languages, duplicate definitions are always disallowed. In the case of mobl, duplicate definitions are not allowed for functions or entities, but they are allowed for variables, just as in JavaScript. The following constraint rules checks for duplicates definitions:

```
constraint-error:
  name -> (name, $[Duplicate definition])
  where
    namespace := <index-namespace> name;
    not(<is-duplicates-allowed> namespace);
    defs := <index-lookup-all> name;
    <gt> (<length> defs, 1)

is-duplicates-allowed:
  Var() -> True()
```

Again, this rule is created automatically in a new Spoofax project, but it provides an interesting showcase of the index API. This rule matches any tree node `name` and starts by determining its namespace using `index-namespace`. Then, it fires a helper rule `is-duplicates-allowed`.

We only report duplicate definitions if the helper does not succeed (indicated by the `not`)[6]. For mobl, that means we never report this error for definitions in the `Var()` namespace. Next, the constraint rule determines all definitions of the reference. If the list has more than one element, the rule reports an error. This is checked by comparing the length of the list with `1`.

[6] Since there is no definition for anything else than `Var()`, the rule will fail in these cases.

Another important constraint for names is *definition before use*: for many program elements, it is an error if it is used before it is defined. In mobl, this property is enforced for variables, but not for entities or functions. The rule to check for this property follows the same pattern as the rule for duplicates, and should normally not be changed by users. Instead, like before, the rule uses a helper rule that can be customized to determine which namespaces require definition before use:

```
is-ordered-namespace:
  Var() -> True()
```

More sophisticated constraints and error messages can be specified using a type system, as we show in the next chapter.

4
Type Systems

Type systems can be considered a sophisticated version of constraints — they also define the static semantics of a language. However, the calculation of types for a language is complex enough to warrant special support beyond what simple constraints provide. In this chapter we discuss what type systems have to be able to do, various strategies for computing types and we provide the usual examples with Xtext, MPS and Spoofax.

Type systems are basically sophisticated constraints that check typing rules in programs. Here is a definition from Wikipedia:

> A type system may be defined as a tractable syntactic framework for classifying phrases according to the kinds of values they compute. A type system associates types with each computed value. By examining the flow of these values, a type system attempts to prove that no type errors can occur. The type system in question determines what constitutes a type error, but a type system generally seeks to guarantee that operations expecting a certain kind of value are not used with values for which that operation makes no sense.

In summary, type systems associate types with program elements and then check whether these types conform to predefined typing rules. We distinguish between dynamic type systems which perform the type checks as the program executes, and static type systems, where type checks are performed ahead of execution, mostly based on type specifications in the program. This chapter focuses exclusively on static type checks.

4.1 Type Systems Basics

To introduce the basic concepts of type systems, let us go back to the example used at the beginning of the section on syntax. As a reminder

here is the example code, and Fig. 4.1 shows the abstract syntax tree.

```
var x: int;
calc y: int = 1 + 2 * sqrt(x)
```

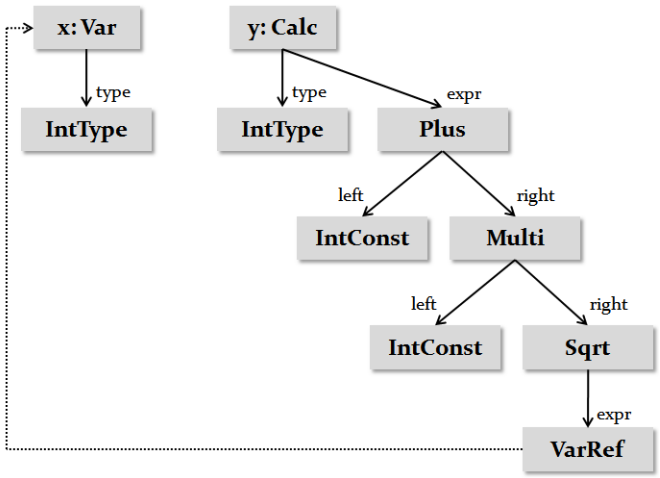


Figure 4.1: Abstract syntax tree for the above program. Boxes represent instances of language concepts, solid lines represent containment, dotted lines represent cross-references

Using this example, we can illustrate in more detail what type systems have to do:

Declare Fixed Types Some program elements have fixed types. They don't have to be derived or calculated, they are always the same and known in advance. Examples include the integer constants `IntConst` (whose type is `IntType`), the square root concept `sqrt` (whose type is `double`), as well as the type declarations themselves (the type of `IntType` is `IntType`, the type of `DoubleType` is `DoubleType`).

Derive Types For some program elements, the type has to be derived from the types of other elements. For example, the type of a `VarRef` (the variable reference) is the type of the referenced variable. The type of a variable is the type of it's declared type. In the example above, the type of `x` and the reference to `x` is `IntType`.

Calculate Common Types Most type systems have some kind of type hierarchy. In the example, `IntType` is a subtype of `DoubleType` (so `IntType` can be used wherever `DoubleType` is expected). A type system has to support the specification of such subtype relationships. Also, the type of certain program elements may be calculated from the arguments passed to them; in many cases the resulting type will be the "more general one" based on the subtyping relationship. Examples include the `Plus` and `Multi` concepts: if the left and right

arguments are two `IntTypes`, the result is an `IntType`. In case of two `DoubleTypes`, the result is a `DoubleType`. If an `IntType` and a `DoubleType` are used, the result is a `DoubleType`, the more general of the two.

Type Checks Finally, a type system has check for type errors and report them to the user. In the example, a type error would occur if something with a `DoubleType` were assigned to an `IntType` variable.

Note that the type of a program element is generally not the same as it's language concept. For example, the concept (meta class) of the number `1` is `IntConst` and its type is `IntType`. The type of the `sqrt` is `DoubleType` and its concept is `Sqrt`. Only for type declarations themselves the two are (usually) the same: the type of an `IntType` is `IntType`. Different instances of the same concept can have different types: a + calculates it's type as the more general of the two arguments. So the type of each + instance depends on the types of the arguments of that particular instance.

Conceptually, the core of a type system can be considered to be a function `typeof` that calculates the type for a program element. Types are often represented with the same technology as the language concepts. As we will see, in case of MPS types are just nodes, i.e. instances of concepts. In Xtext, we use `EObjects`, i.e. instances of `EClasses` as types. In Spoofax, any ATerm can be used as a type. In all cases, we can even define the concepts as part of the language. This is useful because most of the concepts used as types also have to be used in the program text whenever types are explicitly declared (as in `var x: int`).

4.2 *Type Calculation Strategies*

In the end, the `typeof` function can be implemented in any way suitable; after all, it is just program code. However, in practice, three approaches seem to be used most: recursion, unification and pattern matching. We will explore each of these conceptually, and then provide examples in the tool sections.

4.2.1 *Recursion*

Recursion is widely used in computer science. According to Wikipedia, it refers to

> a method of defining functions in which the function being defined is applied within its own definition.

The standard example is the calculation of factorial, where the `factorial` function calls itself:

```
int factorial( int n ) {
    if ( n <= 1 ) return 1;
    else return n * factorial( n-1 );
}
```

In the context of type systems, the recursive approach for calculating a type defines a polymorhphic function `typeof`, which takes a program element and returns its type, while calling itself[1] to calculate the types of those elements on which its own type depends. Let us consider the following example grammar (we use Xtext notation here):

[1] or, most likely, one of the polymorphic overloads

```
LocalVarDecl:
    "var" name=ID ":" type=Type ("=" init=Expr)?;
```

The following examples are structurally valid example sentences:

```
var i: int          // 1
var i: int = 42     // 2
var i: int = 33.33  // 3
var i = 42          // 4
```

Let's develop the pseudo-code for `typeof` function the `LocalVarDecl`. A first attempt could look as follows:

```
typeof( LocalVarDecl lvd ) {
    return typeof( lvd.type )
}

typeof( IntType it ) { return it }
typeof( DoubleType dt ) { return dt }
```

Notice how the `typeof` for `LocalVarDecl` recursively calls `typeof` for its `type` property. Recursion ends with the `typeof` functions for the types; they return themselves. This implementation successfully calculates the type of the `LocalVarDecl`, but it does not address the type check that makes sure that, if an `init` expression is specified, it has the same type (or a subtype) of the `type` property. This could be achieved as follows:

```
typeof( LocalVarDecl lvd ) {
    if isSpecified( lvd.init ) {
        assert typeof( lvd.init ) isSameOrSubtypeOf typeof( lvd.type )
    }
    return typeof( lvd.type )
}
```

Notice (in the grammar) that the specification of the variable type (in the `type` property) is also optional. So we have create a somewhat more elaborate version of the function:

```
typeof( LocalVarDecl lvd ) {
    if !isSpecified( lvd.type ) && !isSpecified( lvd.init )
        raise error

    if isSpecified( lvd.type ) && !isSpecified( lvd.init )
     return typeof( lvd.type )

    if !isSpecified( lvd.type ) && isSpecified( lvd.init )
     return typeof( lvd.init )

    // otherwise...
    assert typeof( lvd.init ) isSameOrSubtypeOf typeof( lvd.type )
    return typeof( lvd.type )
}
```

This code is quite verbose. Assuming that assertions are ignored if one of the called `typeof` functions returns `null` because the argument is not specified, we can simplify this to the following code. This is probably the shortest version of the code that can be imagined using recursive function calls and a suitable language[2].

[2] Note how ignoring an assertion if an argument is `null` is a good example of custom semantics that is useful for a given purpose. Essentially, we have just defined a DSL for type calculations based on recursive function calls)

```
typeof( LocalVarDecl lvd ) {
    assert isSpecified lvd.type || isSpecified lvd.init
    assert typeof( lvd.init ) isSameOrSubtypeOf typeof( lvd.type )
    return typeof( lvd.type )
}
```

4.2.2 *Unification*

Unification is the second well-known approach to type calculation. Once again we start with a definition from Wikipedia:

> Unification is an operation [..] which produces from [..] logic terms a substitution which [..] makes the terms equal modulo some equational theory.

While this sounds quite sophisticated, we have all used unification in high-school for solving sets of linear equations. The *equational theory* is algebra. Here is an example:

```
(1) 2 * x == 10
(2) x + x == 10
(3) x + y == 2 * x + 5
```

Substitution refers to assignment of values to `x` and `y`. A solution for this set of equations is `x := 5, y := 10`.

Using unification for type systems means that language developers specify a number of type equations which contain type variables (cf the `x` and `y`) as well as type values (the numbers in the above example). Some kind of engine is then trying to make all equations `true` by assigning type values to the type variables in the type equations. The interesting property of this approach is that there is no distinction between typing rules and type checks. We simply specify a set of equations that must be `true` for the types to be valid. If an equation cannot be satisfied for any assignment of type values to type variables, a type error is detected. To illustrate this, we return to the `LocalVarDecl` example introduced above.

```
var i: int          // 1
var i: int = 42     // 2
var i: int = 33.33  // 3
var i = 42          // 4
```

The following two type equations constitute the complete type system specification. The `:==:` operator expresses type equation (left side must be the same type as right side), `:<=:` refers to subtype-equation (left side must be same type or subtype of right side, the pointed side of < points to the "smaller", the more specialized type). The operators are taken from MPS, which uses this unification for the type system.

```
typeof( LocalVarDecl.init ) :<=: typeof( LocalVarDecl.type)
typeof( LocalVarDecl ) :==: typeof( LocalVarDecl.type )
```

Let us look at the four examples cases. We use capital letters for free type variables. In the first case, the `init` expression is not given, so the first equation is ignored. The second equation can be satisfied by assigning `T`, the type of the variable declaration, to be `int`. The second equations acts as a type derivation rule and defines the type of the overall `LocalVarDecl` to be `int`.

```
// var i: int
typeof( int ) :<=: typeof( -null- ) // ignore
typeof( T ) :==: typeof( int )      // T := int
```

In the second case the `type` and the `init` expression are given, and both have types that can be calculated independent of the equations specified for the `LocalVarDecl` (they are fixed). So the first equation has no free type variables, but it is `true` with the type values specified (two `int`s). Notice how in this case the equation acts as a type check:

if the equation were not `true` for the two given values, a type error would be reported. The second equation works the same as above, deriving `T` to be `int`.

```
// var i: int = 42
typeof( int ) :<=: typeof( int )    // true
typeof( T ) :==: typeof( int )      // T := int
```

The third case is similar to the second case; but the first equation, in which all types are specified, is not `true`, so a type error is raised.

```
// var i: int = 33.33
typeof( int ) :<=: typeof( double ) // error!
typeof( T ) :==: typeof( int )      // T := int
```

Case four is interesting because no variable type is explicitly specified; the idea is to use what's known as type inference to derive the type from the `init` expression. In this case there are two free variables in the equations, substituting both with `int` solves both equations. Notice how the unification approach automatically leads to support for type inference!

```
// var i = 42
typeof( U ) :<=: typeof( int )  // U := int
typeof( T ) :==: typeof( U )    // T := int
```

To further illustrate how unification works, consider the following example where we try to provide typing rules for array types and array initializers.

```
var i: int[]
var i: int[] = {1, 2, 3}
var i = {1, 2, 3}
```

Compared to the `LocalVarDecl` example above, the additional complication in this case is that we need to make sure that *all* the initialization expressions (inside the curly braces) have the same or compatible types. Here are the typing equations:

```
type var T
foreach ( e: init.elements )
    T :<=: typeof(e)
typeof( LocalVarDecl.type ) :<=: T
typeof( LocalVarDecl ) :==: typeof( LocalVarDecl.type )
```

We introduce an additional type variable `T` and iterate over all the expression in the array initializer, establishing an equation between each of these elements and `T`. This results in a set of equations that *each* must be satisfied[3]. The only way to achieve this is that all array initializer members are of the same (sub-)type. In the examples, this makes `T` to be `int`. The rest of the equations works as explained above. Notice that if we'd write `var i = {1, 33.33, 3}`, then `T := double`, but the equations would still work because we use the `:<=:` operator.

[3] This clearly illustrates that the `:<=:` operator is *not* an assignment, since if it were, only the last of the `init.elements` would be assigned to `T`, which clearly makes no sense.

4.2.3 *Pattern Matching*

In pattern matching, we simply list the possible combinations of types in a big table. Cases that are not listed in the table will result in errors. For our `LocalVarDecl` example, such a table could look like the following:

typeof(type)	typeof(init)	typeof(LocalVarDecl)
int	int	int
int	-	int
-	int	int
double	double	double
double	-	double
-	double	double
int	double	int
double	int	int

To avoid repeating everything for all valid types, variables could be used. `T+` refers to `T` or subtypes of `T`.

typeof(type)	typeof(init)	typeof(LocalVarDecl)
T	T	T
T	-	T
-	T	T
T	T+	T

Pattern matching is used for binary operators in MPS and also for matching terms in Spoofax.

4.3 *Xtext Example*

Up until version 1.0, Xtext provided no support for implementing type systems (beyond implementing everything manually and plugging it into the constraints). In version 2.0, a type system integrated with the JVM's type system is available. Since it is limited to JVM-related types,

it is not as versatile as it could be[4], since it cannot easily be used for languages that have no relationship with the JVM, such as C or C++.

As a consequence, two third-party libraries have been developed: the Xtext Typesystem Framework developed by the author[5], and XTypes (developed by Lorenzo Bettini[6]. In the remainder of this section we will look at the Xtext Typesystem Framework.

[4] We illustrate it to some extent in the section on language modularity .

[5] http://code.google.com/a/eclipselabs.org/p/xtext-typesystem/

[6] http://xtypes.sourceforge.net/

■ *Xtext Typesystem Framework* The Xtext Typesystem Framework is fundamentally based on the recursive approach. It provides an interface `ITypesystem` with a method `typeof(EObject)` which returns the type for the program element passed in as an argument. In its simplest form, the interface can be implemented manually with arbitrary Java code. To make sure type errors are reported as part of the Xtext validation, the type system framework has to be integrated into the Xtext validation framework manually:

```
@Inject
private ITypesystem ts;

@Check(CheckType.NORMAL)
public void validateTypes( EObject m ) {
    ts.checkTypesystemConstraints( m, this );
}
```

As we have discussed above, most typesystems are built from a limited set of typing strategies (assigning fixed types, deriving the type of an element from one of its properties, calculating the type as the common type of its two arguments). The `DefaultTypesystem` class implements `ITypesystem` and provides support for declaratively specifying these strategies. In the code below, a simplified version of the type system specification for the cooling language, the `initialize` method defines one type (the type of the `IntType` is a clone of itself) and defines one typing constraint (the `expr` property of the `IfStatement` must be a Boolean). Also, for types which cannot be specified declaratively, an operation `type(..)` can be implemented to programmatically define types.

```
public class CLTypesystem extends DefaultTypesystem {

    private CoolingLanguagePackage cl = CoolingLanguagePackage.eINSTANCE;

    @Override
    protected void initialize() {
        useCloneAsType(cl.getIntType());
        ensureFeatureType(cl.getIfStatement(),
            cl.getIfStatement_Expr(), cl.getBoolType());
    }

    public EObject type( NumberLiteral s, TypeCalculationTrace trace ) {
        if ( s.getValue().contains(".")) {
            return create(cl.getDoubleType());
        }
```

```
        return create(cl.getIntType());
    }
}
```

In addition to the API used in the code above, the Typesystem Framework also comes with a textual DSL to express typing rules. From the textual type system specification, a generator generates the implementation of the Java class that implements the type system using the APIs. In that sense, the DSL is just a facade on top of a framework; however, this is a nice example of how a DSL can provide added value over a framework or API (Fig. 4.2 shows a screenshot):

- the notation is much more concise compared to the API
- referential integrity and code completion with the target language meta model is provided
- if the typing rules are incomplete, a static error is shown in the editor, as opposed to getting runtime error during initialization of the framework (see the warning in (Fig. 4.2)
- `Ctrl-Click` on a property jumps to the typing rule that defines the type for that property.

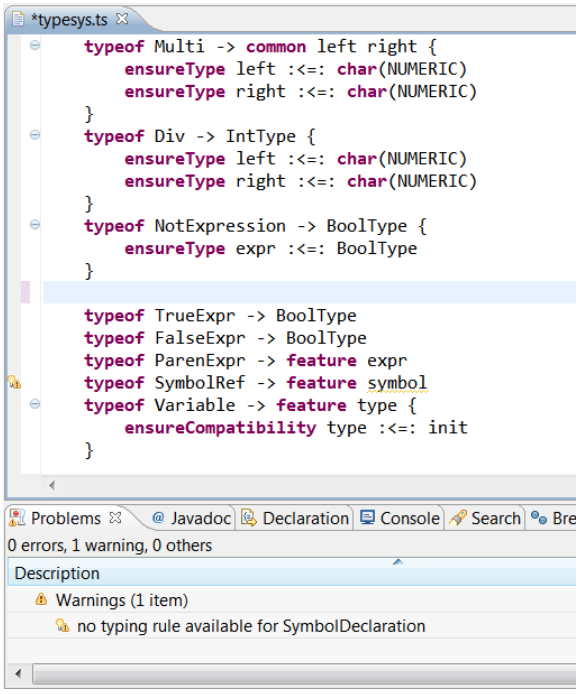


Figure 4.2: The Xtext-based editor for the type system specification DSL provided by the Xtext Typesystem Framework. It is a nice example of the benefits of a DSL over an API (on which it is based), since it can statically show inconsistencies in the type system definition, has a more concise syntax and provides customized go-to-definition functionality.

■ *Type system for the Cooling language* The complete type system for the cooling language is 200 lines of DSL code, and another 100 lines of Java code. We'll take a look at some representative examples.

Primitive types usually use a copy of themselves as their type[7]. This is specified as follows:

[7] It has to be a copy as opposed to the object itself, because the actual program element must not be pulled out of the EMF containment tree.

```
typeof BoolType -> clone
typeof IntType -> clone
typeof DoubleType -> clone
typeof StringType -> clone
```

Alternatively, since all primitive types extend an abstract meta class `PrimitiveType`, this could be shortened to the following, where the + operator specifies that the rule applied for the specified concept and all its subconcepts:

```
typeof PrimitiveType + -> clone
```

For concepts that have a fixed type that is different from the concept itself (or a clone), the type can simply be specified:

```
typeof StringLiteral -> StringType
```

Type systems are most important, and most interesting, in the context of expressions. Since all expressions derive from the abstract `Expr` concept, we can declare that this class is abstract, and hence no typing rule is given. However, the editor reports a warning if there are concrete subclasses of an abstract class for which no type is specified either.

```
typeof Expr -> abstract
```

The notation provided by the DSL groups typing rules and type checks for a single concept. The following is the typing information for the `Plus` concept. It declares the type of `Plus` to be the common type of the `left` and `right` arguments ("the more general one") and then adds two constraints that check that the `left` and `right` argument are either `strings`, `ints` or `doubles`.

```
typeof Plus -> common left right {
    ensureType left :<=: StringType, IntType, DoubleType
    ensureType right :<=: StringType, IntType, DoubleType
}
```

The typing rules for `Equals` are also interesting. It specifies that the resulting type is `boolean` that the `left` and `right` arguments must be `COMPARABLE`, and that the left and right arguments be compatible. `COMPARABLE` is a so-called type characteristic: this can be considered as collection of types. In this case it is `IntType`, `DoubleType`, and `BoolType`. The `:<=>:` operator describes unordered compatibility: the types of the two properties `left` and `right` must either be the same, or `left` must be a subtype or `right`, or vice versa.

```
characteristic COMPARABLE {
    IntType, DoubleType, BoolType
}
typeof Equals -> BoolType {
    ensureType left :<=: char(COMPARABLE)
    ensureType right :<=: char(COMPARABLE)
    ensureCompatibility left :<=>: right
}
```

There is also support for *ordered* compatibility, as can be seen from the typing rule for `AssignmentStatement` below. It has no type (it is a statement!), but the `left` and `right` argument must exhibit ordered compatibility: they either have to be the same types, or `right` must be a subtype of `left`, *but not vice versa*.

```
typeof AssignmentStatement -> none {
    ensureCompatibility right :<=: left
}
```

The framework uses the generation gap pattern, i.e. from the DSL-based type specification, a generator creates a class `CLTypesystemGenerated` (for the cooling language) that contains all the code that can be derived from the type system specification. Additional specifications that cannot be expressed with the DSL (such as the typing rule for `NumberLiteral` or type coercions) can be implemented in Java[8].

[8]. The type system DSL is *incomplete*, since some aspects of type systems have to be coded in the "lower level". However, in this case this is appropriate since it keeps the type system DSL simple and, since the DSL users are programmers, it is not a problem for them to write a few lines of Java code.

4.4 MPS Example

MPS includes a DSL for type system rule definition. It is based on unification, and pattern matching for binary operators.

The type of a `LocalVariableReference` is calculated with the following typing rule[9]. It establishes an equation between the type of the `LocalVariableReference` itself and the variable it references. `typeof` is a built-in operator returns the type for its argument.

[9] Since only the expression within the `do {...}` block has to be written by the developer, we'll only show that expression in the remaining examples.

```
rule typeof_LocalVariableReference {
```

```
  applicable for concept = LocalVariableReference as lvr
  overrides false

  do {
    typeof(lvr) :==: typeof(lvr.variable);
  }
}
```

The rules for the Boolean `NotExpression` contains two equations. The first one makes sure that the negated expression is Boolean. The second one types the `NotExpression` itself to be Boolean. Just as in Xtext, in MPS types are instances of language concepts.

```
typeof(notExpr.expression) :==: new node<BooleanType>();
typeof(notExpr) :==: <boolean>;
```

In MPS there are two different ways how language concepts can be instantiated. The first one (as shown in the first equation above) uses the BaseLanguage `new` expression. The second one uses a quotation, where "a piece of tree" can be inlined into program code. It uses the concrete syntax of the quoted construct — here: a `BooleanType` — in the quotation.

A more interesting example is the typing of `structs`. Consider the following C code:

```
struct Person {
    char* name;
    int age;
}

int addToAge( Person p, int delta ) {
    return p.age + delta;
}
```

At least two program elements have to be typed: the parameter `p` as well as the `p.age` expression. The type of the `FunctionParameter` concept is the type of its type. This is not specific to the fact that the parameter refers to a `struct`.

```
typeof(parameter) :==: typeof(parameter.type);
```

The language concept that represents the `Person` type in the parameter is a `StructType`. A `StructType` refers to the `StructDeclaration` whose type it represents, and extends `Type`, which acts as the super type for all types in mbeddr C[10].

`p.age` is an instance of a `StructAttributeReference`. It is defined as follows (see Fig. 4.4 as well as the code below). It is an `Expression`,

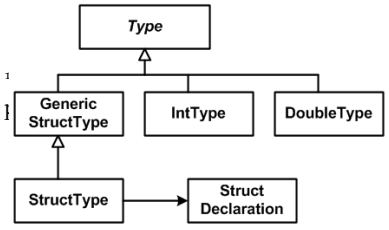


Figure 4.3: Structure Diagram of the language concepts involved in typing `structs`.

owns another expression property (on the left of the dot) as well as a reference to a `StructAttribute` (`name` or `age` in the example).

```
concept StructAttributeReference extends Expression
                                 implements ILValue
  children:
    Expression context 1 specializes: <none>

  references:
    StructAttribute attribute 1 specializes: <none>
```

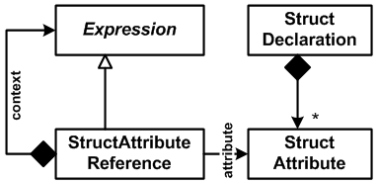


Figure 4.4: Structure Diagram of the language concepts involved in references to `struct` attributes.

The typing rule for the `StructAttributeReference` is shown in the code below. The `context`, the expression on which we use the dot operator, has to be a `GenericStructType`, or a subtype thereof (i.e. a `StructType` which points to an actual `StructDeclaration`). Second, the type of the whole expression is the type of the reference `attribute` (e.g. `int` in case of `p.age`).

```
typeof(structAttrRef.context) :<=: new node<GenericStructType>();
typeof(structAttrRef) :==: typeof(structAttrRef.attribute);
```

This example also illustrates the interplay between the type system and other aspects of language definition, specifically scopes. The referenced `StructAttribute` (on the right side of the dot) may only reference a `StructAttribute` that is part of the the `StructDeclaration` that is referenced from the `StructType`. The following scope definition illustrates this:

```
link {attribute}
  search scope:
    (model, scope, referenceNode, linkTarget, enclosingNode)->join(ISearchScope | sequence<node< >>) {
      node<> exprType = referenceNode.expression.type;
      if (exprType.isInstanceOf(StructType)) {
        return (exprType as StructType).struct.attributes;
      } else {
        return null;
      }
    }
```

As we will discuss in the chapter on language extension and composition, MPS supports incremental extension of existing languages. Extensions may also introduce new types, and, specifically, may allow existing operators to be used with these new types. This is facilitated by MPS' use for pattern matching in the type system, specifically for binary operators such as +, > or ==. As an example consider the introduction of complex numbers into C. It should be possible to write code like this:

```
complex c1 = (1, 2i);
complex c2 = (3, 5i);
complex c3 = c1 + c2; // results in (4, 7i)
```

The + in `c1 + c2` should be the + defined by the original C language[11]. Reusing the original + requires that the typing rules defined for `PlusExpression` in the original C language will now have to accept complex numbers; the original typing rules must be extended. To enable this, MPS supports so-called overloaded operations containers. The following container, taking from the mbeddr C core language, defines the type of + and - if both arguments are `int` or `double`.

[11] Alternatively, we could define a new + for complex numbers. While this would work technically (remember there is no parser ambiguity problems), it would mean that users, when entering a +, would have to decide between the original plus and the new plus for complex numbers. This would not be very convenient from a usability perspective. By reusing the original plus we avoid this problem.

```
overloaded operations rules binaryOperation

operation concepts: PlusExpression | MinusExpression
left operand type: <int>
right operand type: <int>
operation type: (operation, leftOperandType, rightOperandType)->node<> {
  <int>;
}

operation concepts: PlusExpression | MinusExpression
left operand type: <double>
right operand type: <double>
operation type: (operation, leftOperandType, rightOperandType)->node<> {
  <double>;
}
```

To integrate these definitions with the regular typing rules, the following typing rule must be written[12]. Using the `operation type` construct, the typing rules ties in with overloaded operation containers.

[12] Note that only one such rule must be written for all binary operations. Everything else will be handled with the overloaded operations containers

```
rule typeof_BinaryExpression {
  applicable for concept = BinaryExpression as be

  do {
    node<> optype = operation type( be , left , right );
    if (optype != null) {
        typeof(be) :==: optype;
    } else {
        error "operator " + be.concept.name + " cannot be applied to " +
              left.concept.name + "/" + right.concept.name -> be;
    }
  }
}
```

The important aspect of this approach is that overloaded operation containers are *additive*. Language extensions can simply contribute *additional* containers. For the complex number example, this could look like the following: we declare that as soon as one of the arguments is of type complex, the resulting type will be complex as well.

```
operation concepts: PlusExpression | MinusExpression
one operand type: <complex>
operation type: (operation, leftOperandType, rightOperandType)->node<> {
```

```
  <complex>;
}
```

The type system DSL in MPS covers a large fraction of the type system rules encountered in practice. The type system for BaseLanguage, which is an extension of Java, is implemented this way. However, for exceptional cases, procedural BaseLanguage code can be used to implement typing rules as well.

4.5 Spoofax Example

Spoofax' rewrite rules support both the recursive approach and pattern matching in specifying type systems. However, in most projects the recursive approach will be found. Therefor, we will focus on it in the remainder of the section.

■ *Typing Rules in Spoofax* For typing rules in Spoofax, the basic idea is to use rewrite rules to rewrite language constructs to their types. For example, the following rule rewrites integer numbers to the integer type:

```
type-of: Int(value) -> IntType()
```

This is an example for assigning a fixed type to a language element. Similarly, we can rewrite a + expression to the integer type:

```
type-of: Add(exp1, exp2) -> IntType()
```

However, it is good practice to assign types only to well-typed language constructs. Thus, we should add type checks for the subexpressions:

```
type-of:
  Add(exp1, exp2) -> IntType()
  where
    <type-of> exp1 => IntType() ;
    <type-of> exp2 => IntType()
```

Spoofax allows for multiple typing rules for the same language construct. This is particular useful for typing overloaded operators, since each case can be handled by a separate typing rule. For example, when the operator + is overloaded to support string concatenation, we can add the following typing rule:

```
type-of:
  Add(exp1, exp2) -> StringType()
  where
    <type-of> exp1 => StringType() ;
    <type-of> exp2 => StringType()
```

■ *Persistence of Typing Information* As we discussed earlier, Spoofax stores information about definitions in an in-memory data structure called the index. This typically includes information about types. Consider the following typing rules, which rewrites property and variable declarations to their declared types:

```
type-of: Property(name, type) -> type
type-of: VarDecl(name, type) -> type
type-of: VarDeclInit(name, type, init-exp) -> type
type-of: Param(name, type) -> type
```

Spoofax relies on these rules storing type information about property and variable declarations in the index. This is achieved by the following generic rewrite rule:

```
adjust-index-def-data(store-results |namespace, path):
  def -> <store-results> DefData([namespace | path], Type(), type)
  where
    type := <type-of> def
```

Spoofax tries to apply this rule for every definition. If the definition has a type, `type-of` will succeed and the returned type is stored in the index. If the definition has no type, `type-of` will fail. By this, the whole rule will fail and nothing is stored. The stored type information can then be used in the typing rules for variable references and property accesses:

```
type-of:
  Var(name) -> <index-lookup-type> name

type-of:
  PropAccess(exp, name) -> <index-lookup-type> name
```

Both rules rewrite references to the type of their definition sites. First, the definition of a reference is looked-up in the index. Next, this definition is rewritten to its type. This uses the `index-lookup-type` rule, which is another generic rewrite rule:

```
index-lookup-type:
  ref -> type
```

```
  where
    def  := <index-get-data(|Type())> ref ;
    type := <index-lookup> def
```

■ *Additional Types* In Spoofax, types are represented as terms. The constructors for these terms are specified in the syntax definition as labels to productions. Without the ability to define additional constructors, type systems are restricted to types which users can explicitly state in programs, for example in variable declarations. But many type systems require additional types which do not originate from the syntax of the language. Typical examples are top and bottom types in type hierarchies[13]. For example, Java's type system has a special type for `null` values at the bottom of its type hierarchy, which cannot be used as a type in Java programs. Spoofax allows to define constructors for additional types in signatures:

[13] A *top* type is a supertype of every other type, a *bottom* type is a subtype of every other type.

```
signature constructors
  FunType: List(Type) * Type -> FunType
```

This defines a binary constructor `FunType` for function types. The first subterm of a function type is a list of parameter types (`List(Type)`). The second subterm is the return type. We can employ the so defined function type in the typing rules for function definitions and calls:

```
  type-of:
    Function(name, param*, type) -> FunType(type*, type)
    with
      type* := <map(type-of)> param*

  type-of:
    Call(name, arg*) -> type
    where
      <index-lookup-type> name => FunType(type*, type)
```

■ *Type Constraints* Like any other constraint, type constraints are specified in Spoofax by rewrite rules which rewrite language constructs to errors, warnings, or notes. For example, we can define a constraint on additions:

```
  constraint-error:
    exp -> (exp, $[Operator + cannot be applied to arguments [<pprint> type1], [<pprint> type2].])
    where
      !exp  => Add(exp1, exp2) ;
      <not(type-of)> exp ;
      type1 := <type-of> exp1 ;
      type2 := <type-of> exp2
```

Basically, an expression is non-well-typed or *ill-typed*, if it cannot be rewritten to a type. But reporting an error on all ill-typed expressions will make it hard to discover the root cause of the error, since every expression with an ill-typed subexpression is also ill-typed. That is why we also check the subexpressions of the addition to be well-typed. The types of the subexpressions are then used to construct a meaningful error message.

■ *Type Compatibility* Whether two types are compatible is again defined by rewrite rules. These rules rewrite a pair of types to the second element of the pair, if the first one is compatible to it. In the simplest case, both types are the same:

```
is-compatible-to: (type, type) -> type
```

A type might also be compatible to a type, to which its supertype is compatible:

```
is-compatible-to:
  (subtype, type) -> type
  where
    supertype := <supertype> subtype ;
    <is-compatible-to> (supertype, type)
```

For this case, the subtype relation is defined by a rewrite rule, which rewrites a type to its supertype:

```
supertype: IntType() -> FloatType()
```

This approach only works for type systems where each type has at most one supertype. When a type system allows for multiple supertypes, we have to use lists of supertypes and need to adapt the rule for `is-compatible-to` accordingly:

```
supertypes: IntType() -> [ FloatType() ]

is-compatible-to:
  (subtype, type) -> type
  where
    supertype* := <supertypes> subtype ;
    <fetch-elem(is-compatible-to(|type))> supertype*
```

Here, `fetch-elem` tries to find an element in a list of supertypes, which is compatible to `type`. It uses a variant of `is-compatible-to` in order to deal with a list of types. This variant does not rewrite a pair of types

but only the first type. The second type is passed as a parameter to the rewrite rule. It can be defined in terms of the variant for pairs:

```
is-compatible-to(|type2): type1 ->  <is-compatible-to> (type1, type2)
```

The compability of types can easily be extended to compability of lists of types:

```
is-compatible-to:
  (type1*, type2*) -> type*
  where
    type* := <zip(is-compatible-to)> (type1*, type2*)
```

A list `type1*` of types is compatible to another list `type2*` of types, if each type in `type1*` is compatible to the corresponding type in `type2*`. Thereby, `zip` pairs up the types from both lists, rewrites each of these pairs by applying `is-compatible-to` to them, and collects the results in a new list `type*`.

With the extension on lists, we can define a constraint for function calls, which ensures that the types of the actual arguments are compatible to the types of the formal parameters:

```
constraint-error:
  Call(name, arg*) -> (arg*, $[Function [name] cannot be applied to arguments [<pprint> arg-type*].])
  where
    fun-type  := <index-lookup-type> name ;
    !fun-type => FunType(para-type*, type) ;
    arg-type* := <map(type-of)> arg* ;
    <not(is-compatible-to)> (arg-type*, par-type*)
```

5
Transformation and Generation

Transformation and generation have in common that another artifact is created from a program. This is in contrast to interpretation, which executes programs directly, without creating intermediate artifacts. Transformation refers to the case where the created artifact is an AST, and code generation refers to the case where textual concrete syntax is created. In some systems, the two are unified into a common approach. The Xtext, MPS and Spoofax examples illustrate this point.

Transformation of models is an essential step in working with DSLs. We typically distinguish between two different cases: if models are transformed into other models we call this *model transformation*. If models are transformed into text (usually programming language source code, XML or other configuration files) we refer to *code generation*. However, as we will see in the examples below, depending on the approach and tooling used, this distinction is not always easy to make, and the boundary becomes blurred.

A fundamentally different approach to processing models is *interpretation*. While in case of transformation and generation the model is migrated to artifacts expressed in a different language, in case of interpretation no such migration happens. Instead, an interpreter traverses a model and *directly* performs actions depending on the contents of the AS. Strictly speaking, we have already seen examples of interpretation in the sections on constraints and type systems: constraint and type checks can be seen as an interpreter where the actions performed as the tree is traversed are checks of various kinds. However, the term interpretation is typically only used for cases where the actions actually *execute* the model. Execution refers to performing the actions that are associated with the language concepts as defined by the execution semantics of the concepts. We discuss interpretation in the next chapter[1].

[1] We elaborate on the trade-offs between transformation and generation vs. interpretation in the chapter on language design .

5.1 *Overview of the approaches*

Classical code generation traverses a program's AST and outputs programming language source code (or other text). In this context, a clear distinction is made between models and source code. Models are represented as an AST expressed with some preferred AST formalism (or meta meta model); an API exists for the transformation developer to interact with the AST. In contrast, the generated source code is treated as text, i.e. a sequence of characters. The tool of choice for transforming an AST into text are template languages. They support the syntactic mixing of model traversal code and to-be-generated text. The two are separated using some escape character. Since the generated code is treated merely as text, there is no language awareness (and corresponding tool support) for the target language while editing templates. Xtend[2], the language used for code generation in Xtext is an example of this approach.

[2] Xtend is also sometimes referred to as Xtend2, since it has evolved from the old oAW Xtend language. In this chapter we use Xtend to refer to Xtend2. It can be found at http://www.eclipse.org/xtend/

Classical model transformation is the other extreme in that it works with ASTs only and does not consider the concrete syntax of the either the source or the target languages. The source AST is transformed using the source language AS API and a suitable traversal language. As the tree is traversed, the API of the target language AS is used to assemble the target model. For this to work smoothly, most specialized transformation languages assume that the source and target models are build with the same AST formalism (e.g. EMF Ecore). Model transformation languages typically provide support for efficiently navigating source models, and for creating instances of AS of the target language (tree construction). Examples for this approach once again include Xtext's Xtend2 as well as QVT Operational[3] and ATL[4]. MPS can also be used in this way. A slightly different approach establishes relationships between the source and target models instead of "imperatively" constructing a target tree as the source is traversed. While this is often less intuitive to write down, the approach has the advantage that it supports transformation in both directions, and also supports model diff. QVT relational[5] is an example of this approach.

[3] http://en.wikipedia.org/wiki/QVT

[4] http://www.eclipse.org/atl/

[5] http://en.wikipedia.org/wiki/QVT

In addition to the two classical cases described above, there are also hybrid approaches that blur the boundaries between these two clear cut extremes. They are based on the support for language modularization and composition in the sense that the template language and the target language can be composed. As a consequence, the tooling is aware of the syntactic structure and the static semantics of the template language *and* the target language. Both MPS and Spoofax/SDF support this approach to various extents.

In MPS, program is projected and every editing operation directly modifies the AST, while using a typically textual-looking notation as

the "user interface". Template code and target-language code can be represented as nested ASTs, each using its own textual syntax. MPS uses a slightly different approach based on a concept called annotations: As we have elaborated previously, projectional editors can store arbitrary information in an AST. Specifically, it can store information that does *not* correspond to the AS as defined by the language we want to represent ASTs of. MPS code generation templates exploit this approach: template code is fundamentally an instance of the target language. This "example model" is then annotated with template annotations that define how the example model relates to the source model, and which example nodes must be replaced by (further transformed) nodes from the source model. This way, any language can be "templatized" without changing the language definition itself. The MPS example below will elaborate on this approach.

Spoofax, with its Stratego transformation language uses a similar approach based on parser technology. As we have already seen, the underlying grammar formalism supports flexible composition of grammars. So the template language and the target language can be composed, retaining tool support for both of these languages. Execution of the template actually directly constructs an AST of the target language, using the concrete syntax of the target language to specify its structure. The Spoofax example will provide details.

5.2 *Xtext Example*

Since Xtext is based on EMF, generators can be built using any tool that can generate code from EMF models, including Acceleo[6], Jet[7] and Xtend (mentioned above). Xtend is a Java-like general-purpose language that removes some of Java's noise (it has type inference, property access syntax, operator overloading) and adds syntactic sugar (extension methods, multiple-dispatch and closures). Xtend comes with an interpreter and a compiler, the latter generating Java source. Xtend is built with Xtext, so it comes with a powerful Xtext-based IDE. One particularly interesting language feature in the context of code generators are Xtend's *template expressions*. Inside these expressions, a complete template language is available (similar to the old oAW Xpand). It also provides automatic whitespace management[8] The functional abstractions and higher-order functions provided by Xtend make it also very well suited for navigating and querying models. In the rest of this section we will use Xtend for writing code generators and model transformations.

[6] http://www.acceleo.org/pages/home/en

[7] http://www.eclipse.org/modeling/m2t/?project=jet

[8] Indentation of template code is traditionally a challenge, because it is not clear whether whitespace in template code is intended to go into the target file, or is just used for indenting the template itself.

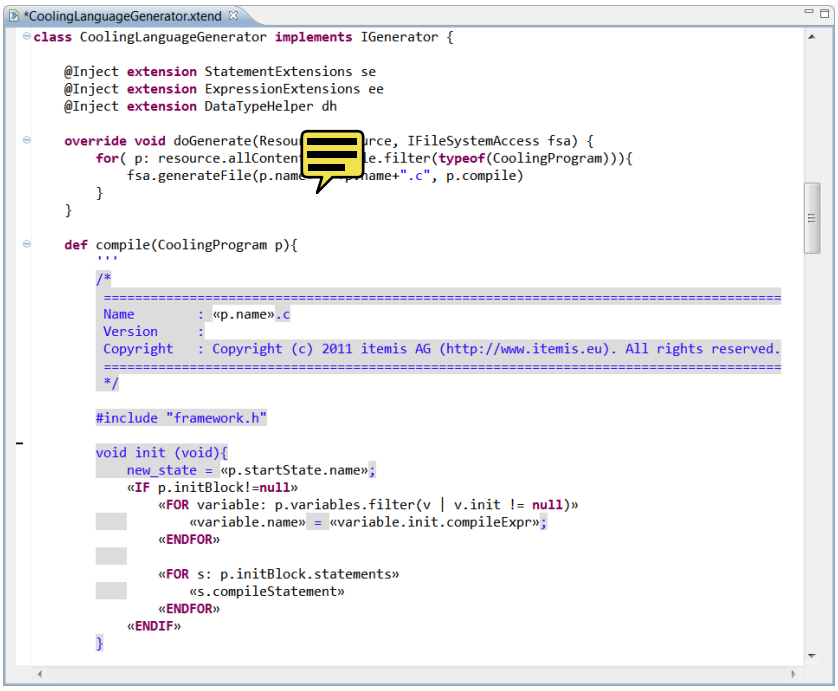


Figure 5.1: The top level structure of a generator written in Xtext is a class that implements the `IGenerator` interface, which requires the `doGenerate` method. Inside generator methods, template expressions (using the triple single quotes) are typically used. Inside those, guillemets (the small double angle brackets) are used to switch between to-be-generated code (grey background) and template control code. Note also the grey whitespace in the `init` function. Grey whitespace is whitespace that will end up in the generated code. White whitespace is used for indentation of template code. Xtend figures out which is which automatically.

5.2.1 *Generator*

We will take a look at generating the C code that implements cooling programs. Fig. 5.1 shows a screenshot of a typical generator. The generator is an Xtend class that implements `IGenerator`, which requires the `doGenerate` method to be defined. The method is called for each model file that has changed[9] (represented by the `resource` argument), and it has to output the corresponding generated code via the `fsa` (file system access) object[10].

When generating code from models there are two distinct cases. In the first case, the majority of the generated code is fixed; only some well-defined part of the code depend on the input model. In this case a template language is the right tool, because template control code can be "injected" into code that looks similar to what shall be generated. On the other hand there are fine grained structures, such as expressions. Since those are basically trees, using template languages for these parts of programs seems unnatural and implies a lot of syntactic noise. A more functional approach is useful. We will illustrate both cases as part of this example.

We start with the high-level structure of the C code generated for a cooling program. The following piece of code illustrates Xtend's power to navigate a model as well as the template syntax for text generation.

[9] This is achieved by a Builder that comes with Xtext. Alternatively, Xtend generators can also be run from the command line, from another Java program or from and and maven.

[10] Like any other Xtext language aspect, the generator has to be registered in the runtime module. Once this is done, the generator is automatically called for each changed resource associated with the respective language.

```
def compile(CoolingProgram program) {
    '''
    <<FOR appl : program.modules.map(m|m.module).filter(typeof(Appliance))>>
        <<FOR c : appl.contents>>
            #define <<c.name>> <<c.index>>
        <<ENDFOR>>
    <<ENDFOR>>

    // more ...
    '''
}
```

The `FOR` loop follows the `modules` collection of the `program`, follows the `module` reference of each one, and then selects all `Appliances` from the resulting collection. The nested loop then iterates over the contents of each `appliance` and generates a `#define`. Notice how the the first two and the last two lines are enclosed in the guillemets. Since we are in template expression mode (''' ... ''') the guillemets escape to template control code. The `#define` is *not* in guillemets, so it is generated into the target file. After the `#define` we generate the name of the respective content element and then its `index`. From within templates, the properties and references of model elements (such as the `name` or the `module` or the `contents`) can simply be accessed using the dot operator. `map` and `filter` are collection methods defined by the Xtend standard library. We also have to generate an `enum` for the states in the cooling program.

```
typedef enum states {
    null_state,
    <<FOR s : program.concreteStates SEPARATOR ",">>
        <<s.name>>
    <<ENDFOR>>
};
```

Here we embed a `FOR` loop inside the `enum` text. Note how we use the `SEPARATOR` keyword to put a comma *between* two subsequent states. In the `FOR` loop we call `concreteStates` on the cooling program. However, if you looked at the grammar or the meta model, you will see that no `concreteStates` property is defined there. Instead, we call an extension method; since it has no arguments, it looks like property access. The method is defined further down in the `CoolingLanguageGenerator` class and looks as follows:

```
def concreteStates(CoolingProgram p) {
    p.states.filter(s | !(s instanceof BackgroundState) && !(s instanceof ShadowState))
}
```

The following code is part of generating the code for a state transition. It first executes the exit actions of the current state, performs the state

change (`current_state = new_state;`) and then performs the entry actions of the new state (not shown):

```
if (new_state != current_state) {
    <<IF program.concreteStatesWitExitActions.size > 0>>
        // execute exit action for state if necessary
        switch (current_state) {
            <<FOR s: p.concreteStatesWitExitActions>>
                case <<s.name>>:
                    <<FOR st: s.exitStatements>>
                        <<st.compileStatement>>
                    <<ENDFOR>>
                    break;
            <<ENDFOR>>
            default:
                break;
    }
    <<ENDIF>>

    // The state change
    current_state = new_state;

    // similar as above, but for entry actions
}
```

The code first uses an `IF` statement to check whether the program has any states with exit actions (via an extension method call). Only if we have such states is the subsequent `switch` statement generated. The `switch` switches over the `current_state`, and then adds a `case` for each state with exit actions[11]. Inside the case we iterate over all the `exitStatements` and call `compileStatement` for each of them.

`compileStatement` is a multimethod. It is polymorphically overloaded based on its argument[12]. For each statement in the cooling language, represented by a subclass of `Statement`, there is an implementation of this method. The next piece of code shows some example implementations.

```
class StatementExtensions {

    def dispatch compileStatement(Statement s){
        // raise error if the overload for the abstract class is called
    }

    def dispatch compileStatement(AssignmentStatement s){
        s.left.compileExpr +" = " + s.right.compileExpr +";"
    }

    def dispatch compileStatement(IfStatement s){
        '''
        if( <<s.expr.compileExpr>> ){
            <<FOR st : s.statements>>
                <<st.compileStatement>>
            <<ENDFOR>>
        }<<IF s.elseStatements.size > 0>> else {
            <<FOR st : s.elseStatements>>
                <<st.compileStatement>>
            <<ENDFOR>>
        }<<ENDIF>>
        '''
    }

    // more ...
}
```

[11] The `s.name` expression in the `case` is actually a reference to the enum literal generated earlier for the particular state. From the perspective of Xtend, we simply generate text: it is not obvious from the template that the name corresponds to an enum literal. Potential structural or type errors are only revealed upon compilation of the generated code.

[12] Note that Java can only perform a polymorphic dispatch based on the `this` pointer. Xtend can dispatch polymorphically over the arguments of methods marked as `dispatch`.

The implementation of the overloaded methods simply returns the text string that represents the C implementation for the respective language construct. The two examples shown are simple because the language construct in the DSL closely resembles the C code in the first place. Notice how the implementation for the `IfStatement` uses a template string, whereas the one for `AssignmentStatement` uses normal string concatenation.

The `compileStatement` methods are implemented in the `StatementExtensions` class. However, from within the `CoolingLanguageGenerator` they are called using method syntax (`st.compileStatement`). This works because they are injected as extensions using the following statement:

```
@Inject extension StatementExtensions
```

Expressions are handled in the same way as statements. The injected class `ExpressionExtensions` defines a set of overloaded `dispatch` methods for `Expression` and all its subtypes.

```
def dispatch String compileExpr (Equals e){
    e.left.compileExpr + " == " + e.right.compileExpr
}

def dispatch String compileExpr (Greater e){
    e.left.compileExpr + " > " + e.right.compileExpr
}

def dispatch String compileExpr (Plus e){
    e.left.compileExpr + " + " + e.right.compileExpr
}

def dispatch String compileExpr (NotExpression e){
    "!(" + e.expr.compileExpr + ")"
}

def dispatch String compileExpr (TrueExpr e){
    "TRUE"
}

def dispatch String compileExpr (ParenExpr pe){
    "(" + pe.expr.compileExpr + ")"
}

def dispatch compileExpr (NumberLiteral nl){
    nl.value
}
```

Since expressions are trees, a `compileExpr` method typically calls `compileExpr` recursively on the children of the expression, if it has any.

5.2.2 *Model-to-Model Transformation*

For model-to-model transformations, the same argument can be made as for code generation: since Xtext is based on EMF, any EMF-based

model-to-model transformation engine can be used with Xtext models. Examples include ATL, QVT-O, QVT-R and Xtend[13].

Essentially, model-to-model transformations are similar to code generators in the sense that they navigate over the model. But instead of producing a text string as the result, they produce another object graph. So the general structure of a transformation is similar. In fact, the two can be mixed. Let us go back to the first code example of the generator:

[13] Of course you could use any JVM-based compatible programming language, including Java itself. However, Java is really not very well suited because of its clumsy support for model navigation and object instantiation. Scala and Groovy are much more interesting in this respect

```
def compile(CoolingProgram program) {
    val transformedProgram = program.transform
    '''
    <<FOR appl : transformedProgram.modules.map(m|m.module).filter(typeof(Appliance))>>
        <<FOR c : appl.contents>>
            #define <<c.name>> <<c.index>>
        <<ENDFOR>>
    <<ENDFOR>>

      // more ...
      '''
}
```

We have added a call to a function `transform` at the beginning of the transformation process. This program creates a new `CoolingProgram` from the original one, and the code generator then uses the `transformedProgram` as the object from which it generates code. In effect, we have added a "preprocessor" model-to-model transformation to the generator. As discussed in the design section , this is one of the most common uses of model-to-model transformations.

The transformation described below enriches the existing model. It creates a new state (`EMERGENCY_STOP`), creates a new event (`emergency_button_pressed`) and then adds a new transition to each existing state that checks if the new event occurred, and if so, transitions to the new state. Essentially, this adds emergency stop behavior to any existing state machine. Let us look at the implementation:

```
class Transformation {

    @Inject extension CoolingBuilder

    CoolingLanguageFactory factory = CoolingLanguageFactory::eINSTANCE

    def CoolingProgram transform(CoolingProgram p ) {
        p.states += emergencyState
        p.events += emergencyEvent
        for ( s: p.states.filter(typeof(CustomState)).filter(s|s != emergencyState) ) {
            s.events += s.eventHandler [
                symbolRef [
                    emergencyEvent()
                ]
                changeStateStatement(emergencyState())
            ]
        }
        return p;
    }

    def create result: factory.createCustomState emergencyState() {
```

```
        result.name = "EMERGENCY_STOP"
    }

    def create result: factory.createCustomEvent emergencyEvent() {
        result.name = "emergency_stop_button_pressed"
    }

}
```

Let us start with the two `create` methods. These methods create new objects, as the `create` prefix suggests[14]. However, simply creating objects can be done with a regular method as well:

[14] The `factory` is the way to create model elements in EMF. It is generated as part of the EMF code generator

```
def emergencyState() {
    val result = factory.createCustomState
    result.name = "EMERGENCY_STOP"
    result
}
```

What is different in `create` methods is that they can be called several times, and they still only ever create one object (per combination of arguments). The result of the first invocation is cached, and all subsequent invocations return the object *created during the first invocation*. Such a behavior is very useful in transformations, because it removes the need to keep track of already created objects. For example, in the `transform` method, we have to establish references to the state created by `emergencyState` and the event created by `emergencyEvent`. To do that, we simply call the same `create` extension again. Since it returns the *same* object as during the first call in the first two lines of `transform`, this actually establishes references to those already created objects[15].

We can now look at the implementation of `transform` itself. It starts out by adding the `emergencyState` and the `emergencyEvent` to the program (these are the first calls to the respective functions, so the objects are actually created at this point). We then iterate over all `CustomStates` except the emergency state we've just created. Notice how we just call the `emergencyState` function again; It returns the same object! We then use a builder to add the following code to each of the existing states.

[15] This is a major difference between text generation and model transformation. In text, two textual occurrences of a symbol are the same thing (in some sense, text strings are value objects). In model transformation the identity of elements does matter. It is not the same if we create *one* new state and then reference it, or if we create five new states. So a good transformation language helps keep track of identities. The `create` methods are a very nice implementation of this principle.

```
on emergency_button_pressed {
    state EMERGENCY_STOP
}
```

This code could be constructed by procedurally calling the respective factory methods:

```
val eh = factory.createEventHandler
val sr = factory.createSymbolRef
sr.symbol = emergencyEvent
val css = factory.createChangeStateStatement
css.targetState = emergencyState
eh.statements += css
s.events += eh
```

The notation used in the actual implementation is more concise and resembles the tree structure of the code much more closely. It uses the well-known *builder* notation. Builders are implemented in Xtend with a clever combination of closures and implicit arguments and a number of functions implemented in the `CoolingBuilder` class[16]. Since this class is imported with the `@Inject extension` construct, the methods become available "just so":

[16] Of course, if you add the line count and effort for implementing the builder, then using this alternative over the plain procedural one might not look so interesting. However, if you just create these builder functions once, and then create many different transformations, this approach makes a lot of sense.

```
class CoolingBuilder {

    CoolingLanguageFactory factory = CoolingLanguageFactory::eINSTANCE

    def eventHandler( CustomState it, (EventHandler)=>void handler ) {
        val res = factory.createEventHandler
        res
    }

    def symbolRef( EventHandler it, (SymbolRef)=>void symref ) {
        val res = factory.createSymbolRef
        it.events += res
    }

    def symbol( SymbolRef it, CustomEvent event ) {
        it.symbol = event
    }

    def changeStateStatement( EventHandler it, CustomState target ) {
        val res = factory.createChangeStateStatement
        it.statements += res
        res.targetState = target
    }
}
```

5.3 *MPS Example*

As we have seen above, the distinction between code generators and model-to-model transformations is much less clear in MPS. While there is a specialized language for ASCII text generation it is really just used "at the end" of the transformation chain as a language like Java or C is generated to text so it can be passed to existing compilers. Fig. 5.2 shows the `textgen` component for mbeddr C's `IfStatement`. MPS' text generation language basically appends text to a buffer. We won't discuss this aspect of MPS any further, since MPS textgen is basically a wrapper language around a `StringBuffer`. However, this is perfectly adequate for the task at hand, since it is only used in the last stage

of generation where the AST is essentially structurally identical to the generated text[17].

```
text gen component for concept IfStatement {
  (node, context, buffer)->void {
    if (node.condition.isInstanceOf(TrueLiteral)) {
      append ${node.thenPart};
    } else {
      append {if ( } ${node.condition} { ) };
      append ${node.thenPart};
      foreach eip in node.elseIfs {
        append ${eip};
      }
      if (node.elsePart != null) {
        append { else };
        append ${node.elsePart};
      }
    }
  }
}
```

Figure 15.2: The AST to text generator for an `if` statement. It first checks if the condition happens to be a `true` literal, in which case the `if` statement is optimized away and only the `thenPart` is output. Otherwise we generate an `if` statement, the condition in parentheses, and then the `thenPart`. We then iterate over all the `elseIfs`; an `elseIf` has its own textgen, and we delegate to that one. We finally output the code for the `else` part.

[17] If you want to generate text that is structurally different, then the textgen language is a bit of a pain to use; in this case, the MPS philosophy recommends you build a suitable intermediate language (such as for XML, or even for a particular schema).

DSLs and language extensions typically use model-to-model transformations to "generate" code expressed in a low level programming language. In general, writing transformation in MPS involves two ingredients. Templates define the actual transformation. Mapping configurations define which template to run when and where. Templates are valid sentences of the target language. So-called macros are used to express dependencies on and queries over the input model. For example, when the guard condition (a C expression) should be generated into an `if` statement in the target model, you first write an `if` statement with a dummy condition into the template. The following would work: `if (true) {}`. Then the nodes that should be replaced by the transformation with nodes from the input model are annotated with macros. In our example, this would look like this: `if (COPY_SRC[true]){}`. Inside the `COPY_SRC` macro you put an expression that describes which elements from the input model should replace the dummy node `true`: `node.guard;` would use the guard condition of the input node (expected to be of type `Transition` here). When the transformation is executed, the `true` node will be replaced by what the macro expression returns — in this case, the guard of the input transition. We will explain this process in detail below.

■ *Translating the State Machine* State machines live inside modules. Just like `structs` they can be instantiated. The following code shows an example. Notice the two global variables `c1` and `c2`, which are instances of the same state machine `Counter`.

```
module Statemachine from cdesignpaper.statemachine imports nothing {

  statemachine Counter {
    in events
      start()
      step(int[0..10] size)
    out events
      started()
      resetted()
      incremented(int[0..10] newVal)
    local variables
      int[0..10] currentVal = 0
      int[0..10] LIMIT = 10
    states ( initial = start )
      state start {
        on start [ ] -> countState { send started(); }
      }
      state countState {
        on step [currentVal + size > LIMIT] -> start { send resetted(); }
        on step [currentVal + size <= LIMIT] -> countState {
          currentVal = currentVal + size;
          send incremented(currentVal);
        }
        on start [ ] -> start { send resetted(); }
      }
  }

  var Counter c1;
  var Counter c2;

  void aFunction() {
    trigger(c1, start);
  }
}
```

State Machines are translated to the following lower level C entities:

- an `enum` for the states (a literal for each state)
- an `enum` for the events (a literal for each event)
- a `struct` declaration that contains an attribute for the current state, as well as attributes for the local variables declared in the state machine
- and finally, a function that implements the behavior of the state machine using a `switch` statement. The function takes two arguments: one named `instance` typed with the `struct` mentioned in the previous item, and one named `event` that is typed to the event `enum` mentioned above. The function checks whether the instance's current state can handle the event passed in, evaluates the guard, and if the guard is `true`, executes exit and entry actions and updates the current state.

This high level structure is clearly discernable from the two main templates shown in Fig. 5.3 and Fig. 5.7.
The MPS transformation engine works in phases. Each phase transforms models expressed in some languages to other models expressed

```
template weave_StatemachineTypesStuffIntoModule
input    Statemachine
content node:
module dummy imports nothing {
  <TF [ exported enum $[statemachineInEvents] { $LOOP$[$[anEvent]; ] } ] TF>
  <TF [ exported enum $[statemachineStates] { $LOOP$[$[aState]; ] } ] TF>
  <TF [ exported struct $[statemachineData] {                       ] TF>
        [   ->$[statemachineStates] __currentState;                 ]
        [   $LOOP$[$COPY_SRC$[int8_t] $[smLocalVar]; ]              ]
      [ };                                                          ]
}
```

Figure 5.3: The MPS generator that inserts two `enum` definitions and a `struct` into the module which contains the `StateMachine`.

in the same or other languages. Model element for which no transformation rules are specified are simply copied. Reduction rules are used to intercept program elements and transform them as generation progresses through the phases. Fig. 5.4 shows how this affects state machines. A reduction rule is defined that maps state machines to the various elements we mentioned above. Notice how the surrounding module remains unchanged, because no reduction rule is defined for it.

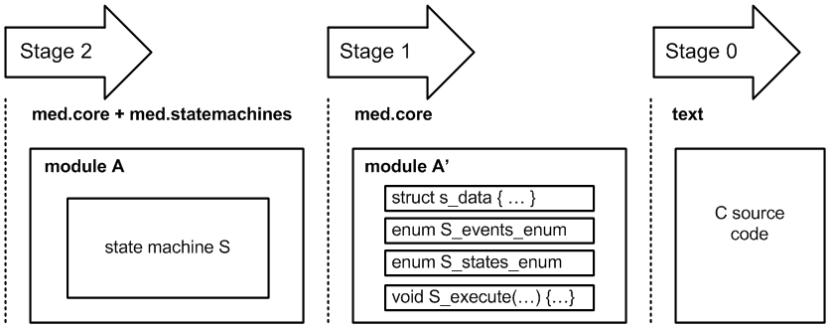


Figure 5.4: State machines are transformed (via a model-to-model transformation, if you will) into two `enums`, a `struct` and a function. These are then transformed to text via the regular `com.mbeddr.core` textgen.

Let us look in more detail at the template in Fig. 5.3. It reduce a `Statemachine`, the input node, to two `enums` and a `struct`. We use template fragments (marked with `<TF ... TF>`) to highlight those parts of the template that should actually be used to replace the input node as the transformation executes. The surrounding `ImplementationModule` (`module dummy`) is scaffolding: it is only needed because `enums` and `structs` *must* live in `ImplementationModules` in a valid instance of the mbeddr C language.

We have to create an `enum` literal for each state and each event. To

achieve this, we iterate over all states (and events, respectively). This is expressed with the LOOP macros in the template (Fig. 5.3). The expression that determines what we iterate over is entered in the Inspector; Fig. 5.5 shows the code for iterating over the statesmm-10. For the literals of the events enum we use a similar expression (node.events;).

[-10] Note that the only really interesting part of Fig. 5.5 is the body of the anonymous function (node.states;), which is why in the future we will only show this part.

```
comment        : <none>
mapping label  : <no label>
mapped nodes   : (node, genContext, operationContext)->sequence<node<>> {
                   node.states;
                 }
```

Figure 5.5: The inspector is used to provide the implementation details for the macros used in the templates. This one belongs to a LOOP macro, so we provide an expression that returns a collection, over which the LOOP macro iterates.

The LOOP macro iterates over collections and then creates an instance of the concept it is attached to for every iteration. In case of the two enums, the LOOP macro is attached to an EnumLiteral, so we create an EnumLiteral for each event and state we iterate over. However, these various EnumLiterals all have to have different names. In fact, the name of each literal should be the name of the state/event for which it is created. We can use a property macro, denoted by the $ sign, to acheive this. A property macro is used to replace values of properties[18]. In this case we use it to replace the name property of the generated EnumLiteral. Here is the implementation expression of the property macro (also entered in the inspector):

[18] The node macros used above (COPY_SRC) replace whole nodes as opposed to primitive *properties* of nodes.

```
node.cEnumLiteralName();
```

In the code above, cEnumLiteralName is a behavior method. It is defined as part of the behavior aspect of the State concept. It concatenates the name of the parent Statemachine with the string __state_ and the name of the current state. We use property macros in a similar way to define the names of the two generated enums themselves.

```
concept behavior State {

  public string cEnumLiteralName() {
    return this.parent : Statemachine.name + "__state_" + this.name;
  }

}
```

The first of the struct attributes is also interesting. It is used to store the current state. It has to be typed to the state enum that is generated from this particular state machine. The type of the attribute is an EnumType; EnumTypes extend Type and reference the EnumDeclaration

whose type they represent. How can we establish the reference to the correct `EnumDeclaration`? We use a reference macro (`->$`) to retarget the reference. Fig. 5.6 shows the macro's expression.

```
comment : <none>
referent : (node, outputNode, genContext, operationContext)->join(node<EnumDeclaration> | string) {
             node.cStatesEnumName();
           }
```

Figure 5.6: A reference macro has to return either the target node, or the name of the target node. The name is then resolved using the scoping rules for the particular reference concept.

Note how a reference macro expects either the target node (here: an `EnumDeclaration`) as the return value, or a `string`. That `string` would be the name of the target element. Our implementation returns the name of the states `enum` generated in the same template. MPS then uses the target language's scoping rules to find and link to the correct target element[19].

[19] So this is not simply a global name lookup! Since MPS knows the reference is an `EnumLiteralRef` expression, the scope of the that concept is used. As long as the name is unique within the scope, this is completely deterministic. Alternatively, the actual node can be identified and returned from the reference macro using mapping labels. However, using names is much more convenient and works also for cross-model references, where mapping labels don't work.

```
template generateSwitchCase
input    Statemachine
content node:
module dummy imports nothing {
  enum events { anEvent; }
  enum states { aState; }
  struct statemachineData {
    states __currentState;
  };
  void statemachineFunction(statemachineData* instance, events event) {
    <TF switch ( instance->__currentState ) {                                   TF>
      $LOOP$ case ->$[aState]: {
               switch ( event ) {
                 $LOOP$ case ->$[anEvent]: {
                          $LOOP$ if ( $COPY_SRC$[true] ) {
                                   $COPY_SRCL$[int8_t exitActions; ]
                                   $COPY_SRCL$[int8_t transActions; ]
                                   instance->__currentState = ->$[aState];
                                   $COPY_SRCL$[int8_t entryActions; ]
                                   return;
                                 } if
                          break;
                        }
               } switch
               break;
             }
    } switch

  } statemachineFunction (function)
}
```

Figure 5.7: The transformation template for generating the `switch`-based implementation of a `StateMachine`. See the text for details.

Let us now address the second main template, Fig. 5.7, which generates the execute function. Looking at the template fragment markers (`<TF ... TF>`) reveals that we only generate the `switch` statement

with this template, *not* the function that contains it. The reason is that we need to be able to embed the state machine `switch` into other function-like concepts as well (e.g. component operations), so we have separated the generation of the function from the generation of the actual state machine behavior.

The `switch` expression is interesting. It switches over the current state of the current state machine instance. That instance is represented by the `instance` parameter passed into the function. It has a `__currentState` field. Notice how the function that contains the `switch` statement *in the template* has to have the `instance` argument, and how its type, the `struct`, has to have the `__currentState` attribute *in the template*. If the respective elements were not there in the template, we couldn't write the template code! Since there is a convention that in the *resulting* function the argument will also be called `instance`, and the attribute will also be called `__currentState`, we don't have to use a reference macro to retarget the two.

Inside the `switch` we then `LOOP` over all the states of the state machine and generate a `case`, using the state's corresponding `enum` literal. Inside the `case`, we embed another `switch` that switches over the `event` argument. Inside this inner `switch` we iterate over all transitions that are triggered by the event we currently iterate over:

```
node<State> state = (node<State>) node.adapter.getUserObject("outer");
state.transitions.where({~it => it.trigger.event == node; });
```

We then generate an `if` statement that checks at runtime whether the guard condition for this transition is `true`. We copy in the guard condition using the `COPY_SRC` macro attached to the `true` dummy node. The `COPY_SRC` macro copies the original node, but it also applies additional reduction rules for this node (and all its descendants), if there are any. For example, in a guard condition, you can refer to event arguments. The reference to `size` in the `step` transition is an example.

```
statemachine Counter {
  in events
    step(int[0..10] size)
    ...
  states ( initial = start )
    ...
    state countState {
      on step [currentVal + size > LIMIT] -> start { send resetted(); }
    }
}
```

Event arguments are mapped to the resulting C function via a `void*` array. A reference to an event argument (`EventArgRef`) hence has to be reduced to accessing the `n`-th element in the array (where `n` is the

index of the event argument in the list of arguments). Fig. 5.8 shows the reduction rule. The reduction rule creates code that looks like this (for an `int` event attribute): `*((int*)arguments[0])`. It accesses the array, casts the element to a pointer to the type of the argument, and then dereferences everything.

```
concept    EventArgRef
inheritors false
condition  <always>
    --> content node:
        module dummy imports nothing {
          void dummy2(void*[ ] arguments) {
                <TF [ (*(($COPY_SRC$[int8_t]*) (arguments[$[1]]))) ] TF>; }
        }
```

Figure 5.8: The reduction rule for references to event arguments (to be used inside guard conditions of transitions).

Inside the `if` statement, we have to generate the code that has to be executed if a transition fires. We first copy in all the exit actions of the current state. Once again, the `int8_t exitActions;` is just an arbitrary dummy statement that will be replaced by a the statements in the exit actions (`COPY_SRCL` replaces a node with a *list* of nodes). The respective expression is this:

```
(node, genContext, operationContext)->sequence<node<>> {
  if (node.parent : State.exitAction != null) {
    return node.parent : State.exitAction.statements;
  }
  new sequence<node<>>(empty);
}
```

We then do the same for the transition actions of the current transition, set the `instance->__currentState` to the target state of the transition using a reference macro (`node.targetState.cEnumLiteralName();`), and then we handle the entry actions of the target state. Finally we return, because at most one transition can fire as a consequence of calling the state machine execute function.

As the last example I want to show how the `TriggerSMStatement` is translated. It "injects" an event into a state machine, and is used as follows:

```
var Counter c1;

void aFunction() {
  trigger(c1, start);
}
```

It has to be translated to a call to the generated state machine execute function that we have discussed at length above. For simplicity, we explain a version of the `TriggerSMStatement` that does not include event arguments. Fig. 5.9 shows the template.

```
content node:
module dummy imports nothing {
  enum eventEnum { e1; e2; }
  struct instanceData {
    << ... >>
  };
  void smExecutionFunction(instanceData* instance, eventEnum event) {

  } smExecutionFunction (function)
  var instanceData theStatemachine;
  void someMethod() {
    <TF [ { ->$[smExecutionFunction](&$COPY_SRC$[theStatemachine], ->$[e1]); } ] TF>
  } someMethod (function)
}
```

Figure 5.9: The reduction rule for a `trigger` statement. It is transformed to a call to the function that implements the behavior of the state machine that is referenced in the first argument of the `trigger` statement.

We use a dummy function `someMethod` so we can embed a function call — because we have to generate a function call to the execute function generated from the state machine. Only the function call is surrounded with the template fragment markers (so only it will be generated). The function we call *in the template* is the `smExecuteFunction` in the template. It has the same signature of the real, generated state machine execute function. We use a reference macro to retarget the `function` reference in the function call. It uses the following expression, which returns the name of the function generated for the state machine referenced in the `statemachine` expression of the trigger statement:

```
node.statemachine.type : StatemachineType.machine.cFunctionName();
```

Note how the first argument to the `trigger` statement can be *any* expression (local variable reference, global variable reference, a function call). However, we know (and enforce via the type system) that the expression's type must be a `StatemachineType`, which has a reference to the `Statemachine` whose instance the expression represents. So we can cast, access the machine, and get the name of the execute function generated from that state machine[20].

The second argument of the `trigger` statement is a reference to the event we want to trigger. We can use another reference macro to find the `enum` literal generated for this event. The macro code is straight forward:

[20] This is an example of where we use the type system in the code generator — not just for checking a program for type correctness.

```
node.event.cEnumLiteralName();
```

5.4 *Spoofax Example*

In Spoofax, model-to-model transformations and code generation are both specified by rewrite rules[21]. This allows for the seamless integration of model-to-model transformation steps into the code generation process; the clear distinction between model-to-model transformation and code generation vanishes.

[21] Rewrite rules have been introduced in .

■ *Code Generation by String Interpolation* Pure code generation from abstract syntax trees to text can be realised by rewriting to strings. The following simple rules rewrite types to their corresponding representation in Java:

```
to-java: IntType()     -> "int"
to-java: BoolType()    -> "boolean"
to-java: StringType()  -> "String"
to-java: EntType(name) -> name
```

For entity types, we use their name as the Java representation. Typically, more complex rules are recursive and use string interpolation[22] to construct strings from fixed and variable parts. For example, the following two rewrite rules generate Java code for entities and their properties:

[22] We used string interpolation already before to compose error messages .

```
to-java:
  Entity(x, ps) ->
  $[ class [x] {
        [ps']
     }
  ]
  with
    ps' := <map(to-java)> ps

to-java:
    Property(x, t) ->
    $[ private [t'] [x];

       public [t'] get_[x] {
           return [x];
       }

       public void set_[x] ([t'] [x]) {
           this.[x] = [x];
       }
    ]
    with
      t' := <to-java> t
```

String interpolation takes place inside `$[...]` brackets and allows to combine fixed text with variables that are bound to strings. Variables

can be inserted using brackets `[...]` without a dollar sign[23]. Instead of variables, we can also use directly the results from other rewrite rules that yield strings or lists of strings:

[23] You can also use any other kind of bracket: `{...}`, `<...>`, and `(...)` are allowed as well.

```
to-java:
  Entity(x, ps) ->
  $[ class [x] {
         [<map(to-java)> ps]
     }
   ]
```

Indentation is important both for the readability of rewrite rules and of the generated code: the indentation leading up to the `$[...]` brackets is removed, but any other indentation beyond the bracket level is preserved in the generated output. This way, we can indent the generated code, but also our rewrite rules. Applying `to-java` to the initial `shopping` entity, will yield the following Java code:

```
class Item {

    private String name;

    public String get_name {
        return name;
    }

    public void set_name (String name) {
        this.name = name;
    }

    private boolean checked;

    public boolean get_checked {
        return checked;
    }

    public void set_checked (boolean checked) {
        this.checked = checked;
    }

    private Num order;

    public Num get_order {
        return order;
    }

    public void set_order (Num order) {
        this.order = order;
    }
}
```

When we prefer camelcase in method names, we need to slightly change our code generation rules, replacing `get_[x]` and `set_[x]` by `get[<to-upper-first>x]` and `set[<to-upper-first>x]`. We also need to specify the following rewrite rule, which turns a string into a list of characters, upper-cases the first character, and turns the characters back into a string:

Option	Description
`(openeditor)`	Opens the generated file in an editor.
`(realtime)`	Re-generates the file as the source is edited.
`(meta)`	Excludes the transformation from the deployed plugin.
`(cursor)`	Transforms always the tree node at the cursor.

Figure 5.10: Options for builders in Spoofax.

```
to-upper-first: s -> s'
  where
    [first|chars] := <explode-string> s ;
    upper         := <to-upper> first ;
    s'            := <implode-string> [upper|chars]
```

■ *Editor Integration* To integrate the code generation into our editor, we first have to define the following rewrite rule:

```
  generate-java:
    (selected, position, ast, path, project-path) -> (filename, result)
    with
      filename := <guarantee-extension(|"java")> path;
      result   := <to-java> selected
```

Generation happens on a per-file basis. The rule follows Spoofax' convention for editor integration. On the left-hand side, it matches the current `selection` in the editor, its `position` in the abstract syntax tree, the abstract syntax tree itself, the `path` of the source file in the editor, and the path of the project this file belongs to. On the right-hand side, it results in the name of the generated file and its content as a string. The file name is derived from the source file's path, while the file content is generated from the current selection. If the generation should consider the whole abstract syntax regardless of the selection, we can replace the last line by `result := <to-java> ast`.

Once we have defined this rule, we can register it as a `builder` in `editor/Lang-Builders.esv`. Here, we add the following rule:

```
builder: "Generate Java code (for selection)" = generate-java (openeditor) (realtime)
```

This defines a label for our transformation, which is added to the editor's *Transform* menu. Additional options can be used to customize the behaviour of the transformation: `(openeditor)` tells Spoofax to open the generated file in an editor, while `(realtime)` enforces an automatic re-generation after changes to the source in the editor. Further options are shown in Fig. 5.10.

■ *Code Generation by Model Transformation* Rewrite rules with string interpolation support a template-based approach to code generation.

Thus, they share two typical problems of template languages. First, they are not *syntax safe*, that is, they do not guarantee the syntactical correctness of the generated code: we might accidently generate Java code which can not be parsed by a Java compiler. Such errors can only be detected by testing the code generator. Second, they inhibit subsequent transformation steps. For example, we might want to optimize the generated Java code, generate Java Bytecode from it, and finally optimize the generate Java Bytecode. At each step, we would first need to parse the generated code from the previous step before we can apply the actual transformation.

Both problems can be avoided by generating abstract syntax trees instead of concrete syntax, i.e. by using model-to-model transformations instead of code (text) generation. This can be achieved by constructing terms on the left-hand side of rewrite rules:

```
to-java: IntType()    -> IntBaseType()
to-java: BoolType()   -> BooleanBaseType()
to-java: StringType() -> ClassType("java.lang.String")
to-java: EntType(t)   -> ClassType(t)

to-java:
  Entity(x, ps) -> Class([], x, ps')
     ps' := <mapconcat(to-java)> ps

to-java:
  Property(x, t) -> [field, getter, setter]
  with
    t'     := <to-java> t ;
    field  := Field([Private()], t', x) ;
    getter := Method([Public()], t', $[get_[x]], [], [Return(VarRef(x))]) ;
    setter := Method([Public()], Void(), $[set_[x]], [Param(t', x)], [assign]) ;
    assign := Assign(FieldRef(This(), x), VarRef(x))
```

■ *Concrete Object Syntax* When we generate abstract syntax trees instead of concrete syntax, we can easily compose transformation steps into a transformation chain by using the output of transformation n as the input for transformation $n + 1$. But this chain will still result in abstract syntax trees. To turn them back into text, it has to be pretty printed (or serialized). Spoofax generates a language-specific rewrite rule `pp-<LanguageName>-string` in `lib/editor-common.generated` which rewrites an abstract syntax tree into a string according to a pretty-printer definition. Spoofax generates a default pretty-printer definition in `syntax/<LanguageName>.generated.pp` from the syntax definition of a language. We will discuss these pretty-printer definitions and how they can be customized in the next chapter.

■ *Concrete Object Syntax* Both template-based and term-based approaches to code generation have distinctive benefits. While template-based generation with string interpolation allows for concrete syntax in code generation rules, abstract syntax tree generation guarantees

syntactical correctness of the generated code and enables transformation chains. To combine the benefits of both approaches, Spoofax can parse user-defined concrete syntax quotations at compile-time, checking their syntax and replacing them with equivalent abstract syntax fragments. For example, a Java return statement can be expressed as `|[ return |[x]|; ]|`, rather than the abstract syntax form `Return(VarRef(x))`. Here, `|[...]|` surrounds Java syntax. It quotes Java fragments inside Stratego code. Furthermore, `|[x]|` refers to a Stratego variable `x`, matching the expression in the return statement. In this case, `|[...]|` is an antiquote, switching back to Stratego syntax in a Java fragment.

To enable this functionality, we have to customize Stratego, Spoofax' transformation language. This requires four steps. First, we need to combine Stratego's syntax definition with the syntax definitions of the source and target languages. There, it is important to keep the sorts of the languages disjunct. This can be achieved by renaming sorts in an imported module, which we do in the following example for the `Java` and the `Stratego` module:

```
module Stratego-Mobl-Java
imports Mobl
imports Java     [ ID   => JavaId ]
imports Stratego [ Id   => StrategoId
                   Var  => StrategoVar
                   Term => StrategoTerm ]
```

Second, we need to define quotations, which will enclose concrete syntax fragments of the target language in Stratego rewrite rules. We add a production rule for every sort of concrete syntax fragments that we like to use in our rewrite rules:

```
exports context-free syntax

  "|[" Module "]|"         -> StrategoTerm {"ToTerm"}
  "|[" Import "]|"         -> StrategoTerm {"ToTerm"}
  "|[" Entity "]|"         -> StrategoTerm {"ToTerm"}
  "|[" EntityBodyDecl "]|" -> StrategoTerm {"ToTerm"}

  "|[" JClass  "]|"   -> StrategoTerm {"ToTerm"}
  "|[" JField "]|"    -> StrategoTerm {"ToTerm"}
  "|[" JMethod "]|"   -> StrategoTerm {"ToTerm"}
  "|[" JFeature* "]|" -> StrategoTerm {"ToTerm"}
```

`StrategoTerm` is the sort for terms in a Stratego program. With these rules, we allow quoted Java fragments whereever an ordinary Stratego term is allowed. We further use `ToTerm` as a constructor in the abstract syntax tree. This way, Stratego can recognise places where we use concrete object syntax inside Stratego code. It will then lift the subtrees at these places into Stratego code. For example, the abstract syntax of

`|[ return |[x]|; ]|` would be `ToTerm(Return(...))`. Stratego lifts this to `NoAnnoList(Op("Return", [...]))`, which is the abstract syntax tree for the term `Return(x)`.

Third, we need to define antiquotations, which will enclose Stratego code in target language concrete syntax fragments. Here, we add a production rule for every sort where we like to inject Stratego code into concrete syntax fragments:

```
exports context-free syntax

  "|[" StrategoTerm "]|" -> JavaId    {"FromTerm"}
```

We use `FromTerm` as a constructor in the abstract syntax tree. Like `ToTerm`, Stratego uses this to recognise places where we switch between concrete object syntax and Stratego code. For example, the abstract syntax of `|[ return |[x]|; ]|` would be `ToTerm(Return(FromTerm(Var("x"))))`. Stratego lifts this to `NoAnnoList(Op("Return", [Var("x")]))`, which is the abstract syntax tree for the term `Return(x)`.

Finally, we need to create a `<filename>.meta` file for every transformation file `<filename>.str` with concrete syntax fragments. In this file, we tell Spoofax to use our customized Stratego syntax definition:

```
Meta([ Syntax("Stratego-Mobl-Java") ])
```

Now, we can use concrete syntax fragments in our rewrite rules:

```
  to-java:
    |[ entity |[x]| { |[ps]| } ]| ->
    |[ class |[x]| { |[<mapconcat(to-java)> ps]| } ]|

to-java:
    |[ |[x]| : |[t]| ]| ->
    |[ private |[t']| |[x]|;

       public |[t']| |[x]| { return |[x]|; }

       public void |[x]| (|[t']| |[x]|) { this.|[x]| = |[x]|; } ]|
    with
      t' := <to-java> t
```

Since Spoofax replaces concrete syntax fragments with equivalent abstract syntax fragments, indentation in the fragments is lost. But the generated code will still be indented by the pretty-printer.

Using concrete object syntax in Stratego code combines the benefits of string interpolation and code generation by model transformation. With string interpolation, we can use the syntax of the target language in code generation rules, which makes them easy to write. However, it is also easy to make syntactic errors, which are only detected when the

generated code is compiled. With code generation by model transformation, we can check if the generated abstract syntax tree corresponds to the grammar of the target language. Actually we can check each transformation rule and detect errors early. With concrete object syntax, we can now use the syntax of the target language in code generation. This syntax is checked by the parser which is derived from the combined grammars of the target language and Stratego.

This comes at the price of adding quotations and antiquotation rules manually. These rules might be generated from a declarative, more concise embedding definition in the future. However, we cannot expect full automation here, since choices for the syntactic sorts involved in the embedding and of quotation and antiquotation symbols require an understanding of Stratego, the target language, and the transformation we want to write. These choices have to be made carefully in order to avoid ambiguities.

In general, there is room for more improvements of the embedding of the target language into Stratego. When the target language comes with a Spoofax editor, we want to get editor services like code completion, hover help, and content folding also in the embedded editor. Until now, only syntax highlighting is supported, using the Stratego coloring rules. Keywords of the target language will be highlighted like Stratego keywords and embedded code fragments will get a grey background color.

6
Building Interpreters

Interpreters are programs that execute DSL programs by directly traversing the DSL program and performing the semantic actions associated with the respective program elements. The chapter contains examples for interpeters with Xtext, MPS and Spoofax.

Interpreters are basically programs that read a model, traverse the AST and perform actions corresponding to the semantics of the language constructs whose instances appear in the AST. How an interpreter implementation looks depends a lot on the programming language used for implementing it. Also, the complexity of the interpreter directly reflects the complexity of the language processes[1]. The following list explains some typical ingredients that go into building interpreters for functional and procedural languages. It assumes a programming language that can polymorphically invoke functions or methods.

[1] For example, building an interpreter for a pure expression language with a functional programming language is almost trivial. In contrast, the interpreters for languages that support parallelism can be much more challenging.

■ *Expressions* For program elements that can be evaluated to values, i.e., expressions, there is typically a function `eval` that is defined for the various expression concepts in the language, i.e. it is polymorphically overridden for subconcepts of `Expression`. Since nested expressions are almost always represented as nested trees in the AST, the `eval` function calls itself with the program elements it owns, and then performs some semantic action on the result. Consider an expression `3 * 2 + 5`. Since the `+` is at the root of the AST, `eval(Plus)` would be called (by some outside entity). It is implemented to add the values obtained by evaluating its arguments. So it calls `eval(Multi)` and `eval(5)`. Evaluating a number literal is trivial, since it simply returns the number itself. `eval(Multi)` would call `eval(3)` and `eval(2)`, multiplying their results and returning the result of the multiplication as its own result, allowing plus to finish its calculation.

Technically, `eval` could be implemented as a method of the AST classes in an object-oriented language. However, this is tyically not done since the interpreter should be kept separate from the AST classes, for example because there may be several interpreters or because the interpreter is developed by other people than the AST.

■ *Statements* Program elements that don't produce a value only make sense in programming languages that have side effects. In other words, execution of such a language concept produces some effect either on global data in the program (re-assignable variables, object state) or on the environment of the program (sending network data or rendering a UI). Such program elements are typically called `Statements`. Statements are typically not recursively nested in a tree, but rather arranged in a list, called a `StatementList`. To execute those, there is typically a function `execute` that is overloaded for all of the different statement types[2]. Note that statements often contain expressions and more statement lists (as in `if (a > 3) { print a; a=0; } else { a=1;}`), so an implementation of `execute` may call `eval` and perform some action based on the result (such as deciding whether to execute the `then`-part of the `else`-part of the `if` statement). Executing the `then`-part and the `else`-part simply boils down to called `execute` on the respective statement lists.

[2] It is also overloaded for `StatementList` which iterates over all statements and calls `execute` for each one.

■ *Environments* Languages that have can express assignment to variables (or modify any other global state) require an environment during execution to remember the values for the variables at each point during program execution. Consider `int a = 1; a = a + 1;`. In this example, the `a` in `a + 1` is a variable reference. When evaluating this reference, the system must "remember" that it had assigned `1` to that variable in the previous statement. The interpreter must keep some kind of global hash table, known as the *environment*, to keep track of symbols and their values, so it can look them up when evaluating a reference to that symbol. Many (though not all) languages that support assigning variables allow reassignment to the same variable (as we do in `a = a + 1;`). In this case, the environment must be updatable. Notice that in `a = a + 1` both mentions of `a` are references to the same variable, and both `a` and `a+1` are expressions. However, only `a` (and not `a+1`) can be assigned to: writing `a * 1 = 10 * a;` would be invalid. The notion of an `lvalue` is introduced to describe this. lvalues can be used "on the left side" of an assignment. Variable references are typically lvalues (if they don't point to a `const` variable). Complex expressions usually are not, unless they evaluate to something that is in turn an lvalue (an example of this is would be `*(someFunc(arg1, arg2)) = 12;`, in C, assuming that `someFunc` returns a pointer to an integer).

■ *Call Stacks* The ability to call other entities (functions, procedures, methods) introduces further complexity, especially regarding parameter and return value passing, and the values of local variables. Assume a function `int add(int a, int b) { return a + b; }`. When this

function is called via `add(2, 3);` the actual arguments 2 and 3 have to be bound to the formal arguments `a` and `b`. An environment must be established for the execution of `add` that keeps track of these assignments. If functions can also access global state (i.e. symbols that are not explicitly passed in via arguments), then this environment must delegate to the global environment in case a referenced symbol cannot be found in the local environment. Supporting recursively callable entities (as in `int fac(int i) { return i == 0 ? 1 : fac(i-1); }`) requires that for each recursive call to `fac` a new environment is created, with a binding for the formal variables. However, the original environment must be "remembered" because it is needed to complete the execution of the outer `fac` after a recursively called `fac` returns. This is achieved using a stack of environments. A new environment is pushed onto the stack as a function is called (recursively), and the stack is popped, returning to the previous environment, as a called function returns. The return value, which is often expressed using some kind of `return` statement, is usually placed into the inner environment using a special symbol or name (such as `__ret__`). It can then be picked up from there as the inner environment is popped.

6.1 *Building an Interpreter with Xtext*

The example discussed in this section is built using the Xtext interpreter framework that ships with the Xtext typesystem framework discussed earlier: http://code.google.com/a/eclipselabs.org/p/xtext-typesystem/

This example describes an interpreter for the cooling language. It is used to allow DSL users to "play" with the cooling programs before or instead of generating C code. The interpreter can execute test cases (and report success or failure) as well as simulate the program interactively. Since no code generation and no real target hardware is involved, the turn-around time is much shorter and the required infrastructure is trivial — just the IDE. The execution engine, as the interpreter is called here, has to handle the following language aspects:

- The DSL supports expressions and statements, for example in the entry and exit actions of states. These have to be supported in the way described above.
- The top level structure of a cooling program is a state machine. So the interpreter has to deal with states, events and transitions.
- The language supports deferred execution (i.e. perform a set of statements at a later time), so the interpreter has to keep track of deferred parts of the program
- The language supports writing tests for cooling programs incl. mock behavior for hardware elements. A set of constructs exists to express

this mock behavior, specifically, ramps to change temperatures over time. These background tasks must be handled by the interpreter as well.

■ *Expressions and Statements* We start our description of the execution engine inside out, by looking at the interpreter for expressions and statements first. As mentioned above, for interpreting expressions, there is typically an overloaded `eval` operation, that contains the implementation of expression evaluation for each subtype of a generic `Expression` concept. However, Java doesn't have polymorphically overloaded member methods[3]. We compensate this by generating a dispatcher that calls a *different* method for each subtype of `Expression`[4]. The generation of this dispatcher is integrated with Xtext via an Xtext workflow fragment, i.e. the dispatcher is generated during the overall Xtext code generation process. The fragment is configured with the abstract meta classes for expressions and statements. The following code shows the fragment configuration:

[3] Java only supports polymorphic dispatch on the `this` pointer, but not on method arguments

[4] If the interpreter had been built with Xtend instead, we would not have had to generate the dispatcher for the `StatementExecutor` or the `ExpressionEvaluator` since Xtend provides polymorphic dispatch on method arguments. However, the fundamental logic and structure of the interpreter would have been similar.

```
fragment = de.itemis.interpreter.generator.InterpreterGenerator {
    expressionRootClassName = "Expression"
    statementRootClassName = "Statement"
}
```

This fragment generates an abstract class that acts as the basis for the interpreter for the particular set of statements and expressions. As the following piece of code shows, the class contains an `eval` method that uses `instanceof` checks to dispatch to a method specific to the subclass and thereby emulating polymorphically overloaded methods. The specific methods throw an exception and are expected to be overridden by a manually written subclass that contains the actual interpreter logic for the particular language concepts. The class also uses a logging framework (based on the `LogEntry` class) that can be used to create a tree shaped trace of expression evaluation, which, short of building an actual debugger, is very useful for debugging and understanding the execution of the interpreter.

```
public abstract class AbstractCoolingLanguageExpressionEvaluator
                    extends AbstractExpressionEvaluator {

    public AbstractCoolingLanguageExpressionEvaluator( ExecutionContext ctx ) {
        super(ctx);
    }

    public Object eval( EObject expr, LogEntry parentLog )
                 throws InterpreterException {
        LogEntry localLog = parentLog.child(LogEntry.Kind.eval, expr,
                 "evaluating "+expr.eClass().getName());
        if ( expr instanceof Equals ) {
            return evalEquals( (Equals)expr, localLog );
        }
```

```
        if ( expr instanceof Unequals ) {
            return evalUnequals( (Unequals)expr, localLog );
        }
        if ( expr instanceof Greater ) {
            return evalGreater( (Greater)expr, localLog );
        }
        // the others...
    }

    protected Object evalEquals( Equals expr, LogEntry log )
              throws InterpreterException {
        throw new MethodNotImplementedException(expr, "method evalEquals not implemented");
    }

    protected Object evalUnequals( Unequals expr, LogEntry log )
              throws InterpreterException {
        throw new MethodNotImplementedException(expr, "method evalUnequals not implemented");
    }

    protected Object evalGreater( Greater expr, LogEntry log )
              throws InterpreterException {
        throw new MethodNotImplementedException(expr, "method evalGreater not implemented");
    }

}
```

A similar class is generated for the statements. Instead of `eval`, the method is called `execute` and it does not return a value. In every other respect the `StatementExecutor` is similar to the `ExpressionEvaluator`.

Let us now take a look at some example method implementations. The following code shows the implementation of `evalNumberLiteral` which evaluates number literals such as `2` or `2.3` or `-10.2`. To recap, the following grammar is used for defining number literals:

```
Atomic returns Expression:
    ...
    ({NumberLiteral} value=DECIMAL_NUMBER);

terminal DECIMAL_NUMBER:
    ("-")? ('0'..'9')* ('.' ('0'..'9')+)?;
```

Before we delve into the details of the interpreter code below, it is worth mentioning that the "global data" held by the execution engine is stored and passed around using an instance of `EngineExecutionContext`. For example, it contains the environment that keeps track of symbol values, and it also has access to the type system implementation class for the language. The `EngineExecutionContext` is available through the `eec()` method in the `StatementExecutor` and `ExpressionEvaluator`.

With this in mind, the implementation of `evalNumberLiteral` should be easily understandable. We first retrieve the actual value from the `NumberLiteral` object, and we find out that type of the number literal using the `typeof` function in the type system. The type system basically inspects whether the value contains a dot or not and returns either a `DoubleType` or `IntType`. Based on this distinction, `evalNumberLiteral` returns either a Java `Double` or `Integer` as the

value of the NumberLiteral. In addition, it creates log entries that document these decisions.

```
    protected Object evalNumberLiteral(NumberLiteral expr, LogEntry log) {
        String v = ((NumberLiteral) expr).getValue();
        EObject type = eec().typesystem.typeof(expr, new TypeCalculationTrace());
        if (type instanceof DoubleType) {
            log.child(Kind.debug, expr, "value is a double, " + v);
            return Double.valueOf(v);
        } else if (type instanceof IntType) {
            log.child(Kind.debug, expr, "value is a int, " + v);
            return Integer.valueOf(v);
        }
        return null;
    }
```

The evaluator for NumberLiteral was simple because number literals are leaves in the AST and have no children, so no recursive invocations of eval are required. This is different for the LogicalAnd, which has two children in the left and right properties. The following code shows the implementation of evalLogicalAnd. The first two statements recursively call the evaluator, for the left and right properties, respectively. They use a utility method called evalCheckNullLog which automatically creates a log entry for this recursive call and stops the interpreter if the value passed in is null (which would mean the AST is somehow broken). Once we have evaluated the two children we can simply return a conjunction of the two.

```
protected Object evalLogicalAnd(LogicalAnd expr, LogEntry log) {
    boolean leftVal = ((Boolean)evalCheckNullLog( expr.getLeft(), log ))
                      .booleanValue();
    boolean rightVal = ((Boole[illegible]CheckNullLog( expr.getRight(), log ))
                      .boolean[illegible];
    return leftVal && rightVal;
}
```

So far, we haven't used the environment, since we haven't worked with variables and their (changing) values. Let's now look and how variable assignment is handled. We first look at the AssignmentStatement, which is implemented in the StatementExecutor:

```
protected void executeAssignmentStatement(AssignmentStatement s, LogEntry log){
    Object l = s.getLeft();
    Object r = evalCheckNullLog(s.getRight(), log);
    SymbolRef sr = (SymbolRef) l;
    SymbolDeclaration symbol = sr.getSymbol();
    eec().environment.put(symbol, r);
    log.child(Kind.debug, s, "setting " + symbol.getName() + " to " + r);
}
```

The first two lines get the left argument as well as the value of the right argument. Note how only the right value is evaluated: the left

argument is a symbol reference (made sure through a constraint, since only `SymbolRefs` are lvalues in this language). We then retrieve the symbol referenced by the symbol reference and create a mapping from the symbol to the value in the environment, effectively "assigning" the value to the symbol during the execution of the interpreter.

The implementation of the `ExpressionEvaluator` for a symbol reference (if it is used not as an lvalue) is shown in the following code. We use the same environment to look up the value for the symbol. We then check if the value is `null` (i.e. nothing has been assigned to the symbol as yet). In this case we return the default value for the respective type and log a warning[5]. Otherwise we return the value.

[5] This is specialized functionality in the cooling language; in most other languages, we would probably just return `null`, since nothing seems to have been assigned to the symbol yet

```
protected Object evalSymbolRef(SymbolRef expr, LogEntry log) {
    SymbolDeclaration s = expr.getSymbol();
    Object val = eec().environment.get(s);
    if (val == null) {
        EObject type = eec().typesystem.typeof(expr, new TypeCalculationTrace());
        Object neutral = intDoubleNeutralValue(type);
        log.child(Kind.debug, expr,
           "looking up value; nothing found, using neutral value: " + neutral);
        return neutral;
    } else {
        log.child(Kind.debug, expr, "looking up value: " + val);
        return val;
    }
}
```

The cooling language does not support function calls, so we demonstrate function calls with a similar language that supports them. In that language, function calls are expressed as symbol references that have argument lists. Below is the grammar. Constraints make sure that argument lists are only used if the referenced symbol is actually a `FunctionDeclaration`.

```
FunctionDeclaration returns Symbol:
    {FunctionDeclaration} "function" type=Type name=ID "("
        (params+=Parameter  ("," params+=Parameter)* )?  ")" "{"
        (statements+=Statement)*
    "}";

Atomic returns Expression:
    ...
    {SymbolRef} symbol=[Symbol|QID]
        ("(" (actuals+=Expr)? ("," actuals+=Expr)* ")")?;
```

The following is the code for the evaluation function for the symbol reference. It must distinguish between references to variables and to functions[6].

[6] This is once again a consequence of the fact that all references to any symbol are handled via the `SymbolRef` class. We have discussed this in .

```
protected Object evalSymbolRef(SymbolRef expr, LogEntry log) {
    Symbol symbol = expr.getSymbol();
    if ( symbol instanceof VarDecl ) {
        return log( symbol, ctx.environment.getCheckNull(symbol), log);
    }
```

```
    if ( symbol instanceof FunctionDeclaration ) {
        FunctionDeclaration fd = (FunctionDeclaration) symbol;
        return callAndReturnWithPositionalArgs("calling "+fd.getName(),
                fd.getParams(), expr.getActuals(), fd.getElements(),
                RETURN_SYMBOL, log);
    }
    throw new InterpreterException(expr,
        "interpreter failed; cannot resolve symbol reference "
        +expr.eClass().getName()); }
```

The code for handling the `FunctionDeclaration` uses a predefined utility method `callAndReturnWithPositionalArgs`. It accepts as arguments the list of formal arguments of the called function, the list of actual arguments (expressions) passed in at the call site, the list of statements in the function body, a symbol that should be used for the return value in the environment as well as the obligatory log. The utility method is implemented as follows:

```
protected Object callAndReturnWithPositionalArgs(String name,
        EList<? extends EObject> formals, EList<? extends EObject> actuals,
        EList<? extends EObject> bodyStatements, Object returnSymbol,
        LogEntry log) {
    ctx.environment.push(name);
    for( int i=0; i<actuals.size(); i++ ) {
        EObject actual = actuals.get(i);
        EObject formal = formals.get(i);
        ctx.environment.put(formal, evalCheckNullLog(actual, log));
    }
    ctx.getExecutor().execute( bodyStatements, log );
    Object res = ctx.environment.get(returnSymbol);
    ctx.environment.pop();
    return res;
}
```

Remember from the above discussion about environments and function calls, that each invocation of a function has to get its own environment to handle the local variables for the particular invocation. We can see this in the first line of the implementation above: we first creates a new environment and push it onto the call stack. Then the implementation iterates over the actual arguments, evaluates each of them and "assigns" them to the formals by creating an association between the formal argument symbol and the actual argument value in the new environment. It then uses the `StatementExecutor` to execute all the statements in the body of the function. Notice that as the executed function deals with its own variables and function calls, it uses the *new* environment created, pushed onto the stack and populated by this method! When the execution of the body has finished, we retrieve the return value from the environment. The `return` statement in the function has put it there under a name we have prescribed, the `returnSymbol`, so we know how to find it in the environment. Finally, we pop the environment, restoring the caller's state of the world and return the return value as the resulting value of the the function call

expression.

■ *States, Events and the Main program* Changing a state (state as in state machine, not as in program state) from within a cooling program is done by executing a `ChangeStateStatement`, which simply references the state that should be entered. Here is the interpreter code in `StatementExecutor`:

```
protected void executeChangeStateStatement(ChangeStateStatement s, LogEntry l) {
    engine.enterState(s.getTargetState(), log);
}

public void enterState(State targetState, LogEntry logger )
        throws TestFailedException, InterpreterException, TestStoppedException {
    logger.child(Kind.info, targetState, "entering state "+targetState.getName());
    context.currentState = targetState;
    executor.execute(ss.getEntryStatements(), logger);
    throw new NewStateEntered();
}
```

`executeChangeStateStatement` calls back to an engine method that handles the state change (since this is a more global operation than executing statements, it is handled by the engine class itself). The method simply sets the current state to the target state passed into the method (the current state is kept track of in the execution context). It then executes the set of entry statements of the new state. After this it throws an exception `NewStateEntered` which stops the current execution step. The overall engine is step driven, i.e. an external "timer" triggers distinct execution steps of the engine. A state change always terminates the current step. The main method `step()` triggered by the external timer can be considered the main program of the interpreter. It looks as follows:

```
public int step(LogEntry logger) {
    try {
        context.currentStep++;
        executor.execute(getCurrentState().getEachTimeStatements(), stepLogger);
        executeAsyncStuff(logger);
        if ( !context.eventQueue.isEmpty() ) {
            CustomEvent event = context.eventQueue.remove(0);
            LogEntry evLog = logger.child(Kind.info, null,
                "processing event from queue: "+event.getName());
            processEventFromQueue( event, evLog );
            return context.currentStep;
        }
        processSignalHandlers(stepLogger);
    } catch ( NewStateEntered se ) {
    }
    return context.currentStep;
}
```

It first increments a counter that keeps track of how many steps haven been executed since the interpreter had been started up. It then executes the `each time` statements of the current state. This is a statement

list defined by a state that needs to be re-executed in each step while the system is in the respective state. It then executes asynchronous tasks. We'll explain those below. Next it checks if an event is in the event queue. If so, it removes the first event from the queue and executes it. After processing an event the step is always terminated. Lastly, if there was no event to be processed, we process signal handlers (the `check` statements in the cooling programs).

Processing events simply checks if the current state declares and event handler that can deal with the currently processed event. If so, it executes the statement list associated with this event handler.

```
private void processEventFromQueue(CustomEvent event, LogEntry logger) {
    for ( EventHandler eh: getCurrentState().getEventHandlers()) {
        if ( reactsOn( eh, event ) ) {
            executor.execute(eh.getStatements(), logger);
        }
    }
}
```

The DSL also supports executing code asynchronously, i.e. after a specified number of steps. The grammar looks as follows:

```
PerformAsyncStatement:
    "perform" "after" time=Expr "{"
        (statements+=Statement)*
    "}";
```

The following method interprets the `PerformAsyncStatements`:

```
protected void executePerformAsyncStatement(PerformAsyncStatement s,
                LogEntry log) throws InterpreterException {
    int inSteps = ((Integer)evalCheckNullLog(s.getTime(), log)).intValue();
    eec().asyncElements.add(new AsyncPerform(eec().currentStep + inSteps,
        "perform async", s, s.getStatements()));
}
```

It registers the statement list associated with the `PerformAsyncStatement` in the list of async elements in the execution context. The call to `executeAsyncStuff` at the beginning of the `step` method described above checks whether the time has come and executes those statements:

```
private void executeAsyncStuff(LogEntry logger) {
    List<AsyncElement> stuffToRun = new ArrayList<AsyncElement>();
    for (AsyncElement e: context.asyncElements) {
        if ( e.executeNow(context.currentStep) ) {
            stuffToRun.add(e);
        }
    }
    for (AsyncElement e : stuffToRun) {
        context.asyncElements.remove(e);
        e.execute(context, logger.child(Kind.info, null, "Async "+e));
    }
}
```

6.2 *An interpreter in MPS*

Building an interpreter in MPS is essentially similar to building an interpreter in Xtext and EMF. All concept would apply in the same way, instead of `EObjects` you would work with the `node<>` types that are available on MPS to deal with ASTs. However, since MPS' BaseLanguage is itself built with MPS, it can be extended. So instead of using a generator to generate the dispatcher that calls the `eval` methods for the expression classes, suitable modular language extensions can be defined in the first place.

For example, BaseLanguage could be extended with support for polymorphic dispatch (similar to what [illegible]2 does). An alternative solution involves a dispatch statement, a kind of "pimped switch". Fig. 6.1 shows an example.

```
public static Double eval(node<Expression> ex, final node<FunctionUnitTest> test) {
  ErrorMarkers.remove(ex);
  return dispatch <Double> (ex)                                                    ;
           MulExpression         -> eval($.leftExpression, test) * eval($.rightExpression, test)
           DivExpressionFraction -> eval($.numerator, test) / eval($.denominator, test)
           IntegerConstant       -> new Double($.value)
           FloatingPointConstant -> Double.valueOf($.value)
           Exp                   -> Math.pow(eval($.base, test), eval($.exp, test))
           SymbolReference       -> getValue($, test)
           default: 0.0
}
```

Figure 6.1: An extension to MPS' BaseLanguage that makes writing interpreters simpler. The `dispatch` statement has the neat feature that, on the right side of the `->`, the `$` reference to the `ex` expression is already downcast to the type mentioned on the left of the `->`.

The dispatch statement tests if the argument `ex` is an instance of the type referenced in the cases. If so, the code on the right side of the arrow is executed. Notice the special expression `$` used on the right side of the arrow. It refers to the argument `ex`, but it is already downcast to the type on the left of the case's arrow. This way, annoying downcasts can be avoided.

Note that this extension is modular in the sense that the definition of BaseLanguage was not changed. Instead, an additional language module was defined that *extends* BaseLanguage. This module can be used as part of the program that contains the interpreter, making the `dispatch` statement available there[7]. Also, the `$` expression is

[7] We discuss language modularization and composition is .

restricted to only be usable on the right side of the ->. This way the overall base language namespace is kept clean.

6.3 An Interpreter in Spoofax

State-based interpreters can be specified with rewrite rules in Spoofax, realising transitions between execution states. This requires:

- A representation of *states*. The simplest way to represent states are terms, but we can also define a new DSL for representing the states in concrete syntax.
- An *initialisation transformation* from a program in the DSL to the initial state of the interpreter.
- A *step transformation* from an actual state of the interpreter to the next state of the interpreter.

In the remainder of the section, we develop an interpreter for a subset of Mobl. We start with a simple interpreter for expressions, which we then extend to handle statements.

6.3.1 An Interpreter for Expressions

If we want to evaluate simple arithmetic expressions without variables, the expression itself is the state of the interpreter[8]. Thus, no extra term signatures are needed and the initialisation transformation is given by identity. For the step transformation, we can define rewrite rules for the different expression kinds:

[8] Remember how in the design chapter we discussed building debuggers for purely functional languages, and in particular, expression languages. We argues that a debugger is trivial, because there is no real "flow" of the program; instead, the expression can be debugged by simply showing the values of all intermediate expressions in a tree-like for. We exploit the same "flowless" nature of pure expression languages when building this interpreter.

```
eval: Add(Int(x), Int(y)) -> Int(z) where z := <add> (x, y)
eval: Mul(Int(x), Int(y)) -> Int(z) where z := <mul> (x, y)

eval: Not(True())  -> False()
eval: Not(False()) -> True()

eval: And(True(), True())   -> True()
eval: And(True(), False())  -> False()
eval: And(False(), True())  -> False()
eval: And(False(), False()) -> False()

eval: LazyAnd(True(), True()) -> True()
eval: LazyAnd(False(), _)     -> False()
eval: LazyAnd(_, False(), _)  -> False()
```

We can orchestrate these rules in two different styles. For a small-step interpreter, we only apply one rule in each step:

```
eval-one: exp -> <oncebu(eval)> exp
```

Here, `oncebu` tries to apply `eval` at one position in the tree, starting from the leaves. The `bu` in `oncebu` stands for bottom-up. We could also use `oncetd`, traversing the tree top-down. However, evaluations are likely to happen at the bottom of the tree, why `oncebu` is the better choice. The result of `eval-one` will be a slightly simpler expression, which might need further evaluation.

In contrast, we can directly apply as many rules as possible, trying to evaluate the whole expression. This will give us a big-step interpreter:

```
eval-all: exp -> <bottomup(try(eval))> exp
```

Here, `bottomup` tries to apply `eval` at every node, starting at the leaves. The result of `eval-all` will be the final result of the expression.

6.3.2 *An Interpreter for Statements*

If we want to evaluate statements, we need states which capture the value of variables and the list of statements which needs to be evaluated. We can define these states with a signature for terms:

```
signature constructors

     : ID * IntValue  -> VarValue
     : ID * BoolValue -> VarValue

State: List(VarValue) * List(Statement) -> State
```

The first two rules define binary tuples which combine a variable name (`ID`) and a value (either `IntValue` or `BoolValue`). The last rule defines a binary constructor `State` which combines a list of variable values with a list of statements. We first have to adapt the evaluation of expressions to handle variable references in expressions.

```
eval(|varvals): exp -> <eval> exp
eval(|varvals): VarRef(var) -> <lookup> (var, varvals)

eval-one(|s): exp -> <oncebu(eval(|s))> exp
eval-all(|s): exp -> <bottomup(try(eval(|s)))> exp
```

The first two rules take the actual list of variable values of the interpreter (`varvals`) as a parameter. The first rule integrates the old evaluation rules, which do not need any state information. The second rule looks up the current value `val` of the variable `var` in the list of variable

values. The last two rules define small-step and big-step interpreters of expressions, just as before.

We can now define evaluation rules for statements. These rules rewrite the current state into a new state:

```
eval:
  (varvals, [Init(type, var, exp)|stmts]) -> (varvals', stmts)
  where
    val      := <eval-all(|varvals)> exp;
    varvals' := <update> ((var, val), varvals)

eval:
  (varvals, [Assign(VarRef(var), exp)|stmts]) -> (varvals', stmts)
  where
    val      := <eval-all(|varvals)> exp;
    varvals' := <update> ((var, val), varvals)

eval:
  (varval, [Block(stmts1)|stmts2]) -> (varvals, <conc> (stmts1, stmts2))
```

The first rule handles variable declarations. It evaluates the expression on the right-hand side to a value, updates the list of variable values and removes the statement from the list of statements. The second rule handling assignments is quite similar. The third rule handles block statements, by concatenating the statements from a block with the remaining statements. The following rule handle an `if` statement:

```
eval:
  (varvals, [If(exp, then, else)|stmts]) -> (varvals, [stmt|stmts])
  where
    val := <eval-all(|varvals)> exp;
    if !val => True() then
      stmt := then
    else
      !val => False();
      stmt := else
    end
```

First, it evaluates the condition. Dependent of the result, it chooses the next statement to evaluate. When the result is `True()`, the statement from the `then` branch is chosen. Otherwise the result has to be `False()` and the statement from the `else` branch is chosen. If the result is neither `True()` nor `False()`, the rule will fail. The following rule handles `while` loops:

```
eval:
  (varvals, [While(exp, body)|stmts]) -> (varvals, stmts')
  where
    val := <eval-all(|varvals)> exp;
    if !val => True() then
      stmts' := [body, While(exp, body)|stmts]
    else
      !val => False();
      stmts' := stmts
    end
```

Again, the condition is evaluated first. If it evaluates to `True()`, the list of statements is updated to the body of the loop, the `while` loop again, followed by the remaining statements. If it evaluates to `False()`, only the remaining statements need to be evaluated.

The `eval` rules define already a small-step interpreter, going from one evaluation state to the next. We can define a big-step interpreter by adding a driver, which repeats the evaluation until it reaches a final state:

```
eval-all: state -> <repeat(eval)> state
```

6.3.3 *More Advanced Interpreters*

We can extend the interpreter to handle function calls and objects in a similar way we did for statements. First, we always have to think about the states of the extended interpreter. Functions will require a call stack, objects will require a heap. Next, we need to consider how the old rules can deal with the new states. Adjustments might be needed. For example, when we support objects, the heap needs to be passed to expressions. Expressions which create objects will change the heap, so we cannot only pass it, but have to propagate the changes back to the caller.

6.3.4 *IDE Integration*

We can integrate interpreters as builders into the IDE. For big-step interpreters, we can simply calculate the overall execution result and show it to the user. For small-step interpreters, we can use the initialisation transformation in the builder. This will create an initial state for the interpreter. When we define a concrete syntax for these states, they can be shown in an editor. The transition transformation can then be integrated as a refactoring on states, changing the current state to the next one. This way, the user can control the execution, undo steps, or even modify the current state.

7
IDE Services

In this chapter we discuss various services provided by the IDE. This includes code completion, syntax coloring, pretty printing, go-to-definition and find references, refactoring, outline views, folding, diff and merge, tooltips and visualization (debugging is handled separately in Chapter 9). We provide examples for each of them with either MPS, Spoofax or Xtext.

In this chapter we illustrate typical services provided by the IDE that are *not* automatically derived from the language definition itself and additional configuration or programming is required. Note that we are not going to show every service with every example tool[1].

[1]. So, if we don't show how X works in some tool, this does *not* mean that you cannot do X with this particular tool.

7.1 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* 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. 7.1).

This can be achieved by overriding the content assist provider for the particular reference. To customize code completion, you have to implement a `complete...` method in the `ProposalProvider` for

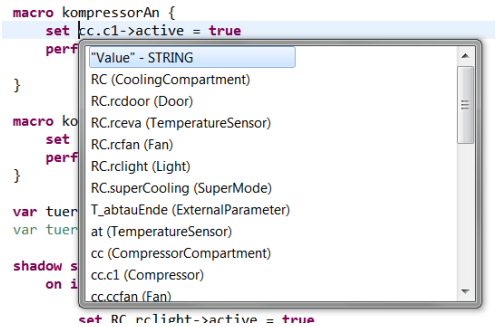


Figure 7.1: Code completion in the cooling language is customized to specify what kind of symbol a particular reference target is.

your language. The method name has to correspond to rule/property whose code completion menu you want to customize. 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 method. 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));
}
```

Once we have the scope we can iterate over all its contents (i.e. the target elements). Note how the scope does not directly contain the target `EObjects`, but rather `IEObjectDescriptions`. This is because the code completion is resolved against the index, a data structure maintained by Xtext that contains all referencable elements. This approach has the advantage that the target resource, i.e. the file that contains the target element, does not have to be loaded just to be able to reference into it.

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`)[2]. The last line then calls the `accept`

[2] Note that we could use a rich string to add some nice formatting to the string.

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 in case the particular code completion menu item is selected.

The contents of the code completion menu for references can be customized in MPS as well. It is instructive to take a look at this in addition to Xtext for two reasons. The first one is brev  onsider the following code:

```
link {function}
  referent set handler:<none>
  search scope:
    ...
  presentation :
    (parameterNode, visible, smartReference, inEditor, model, scope, referenceNode,
              linkTarget, enclosingNode, operationContext)->string {
        parameterNode.signatureInfo();
  }
```

To customize the contents of the code completion menu, one simply has to provide the expression that calculates the text in the `presentation` section of the scope provider. In this example 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 this reference. In the example of the function call, it projects the name of the called function (plus the actual arguments).

■ *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 values[3].

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. 7.2 shows the necessary code.
Every editor cell can have a cell menu (the menu you see when you

[3] Note that if we use this approach to provide code completion for primitive properties this does not affect the constraint check (in contrast to references, where a scope affects the code completion menu and the constraint checks). Users can always type something that is *not* in the code completion menu. A separate constraint check may have to be written.

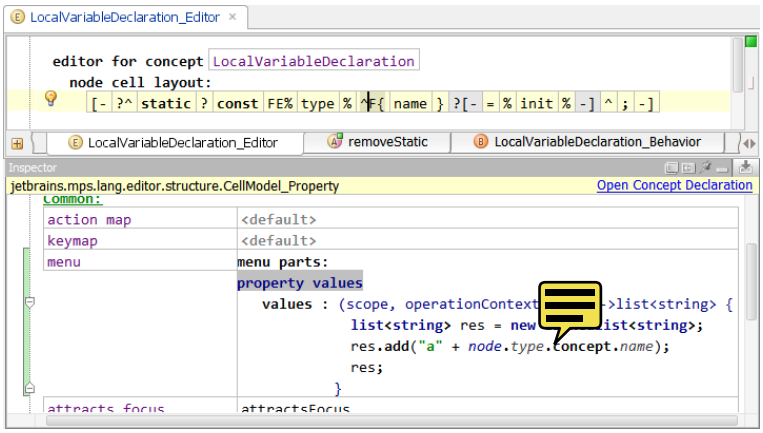


Figure 7.2: A cell menu for the `name` property of a `LocalVariableDeclaration`. In the editor definition (top window) we select the cell that renders the `name`. In the inspector we can then define additional properties for the selected cell. In this case we contribute an additional cell menu that provides the suggested names.

press `Ctrl-Space`). The cell menu consists of several parts. Each part contributes a set of menu entries. In the example in Fig. 7.2, 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 local variable's type's name, prefixed by an `a`).

■ *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-then-else` 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. 7.3 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). 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. 7.4 shows the code for a cell menu.

7.2 *Syntax Coloring*

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

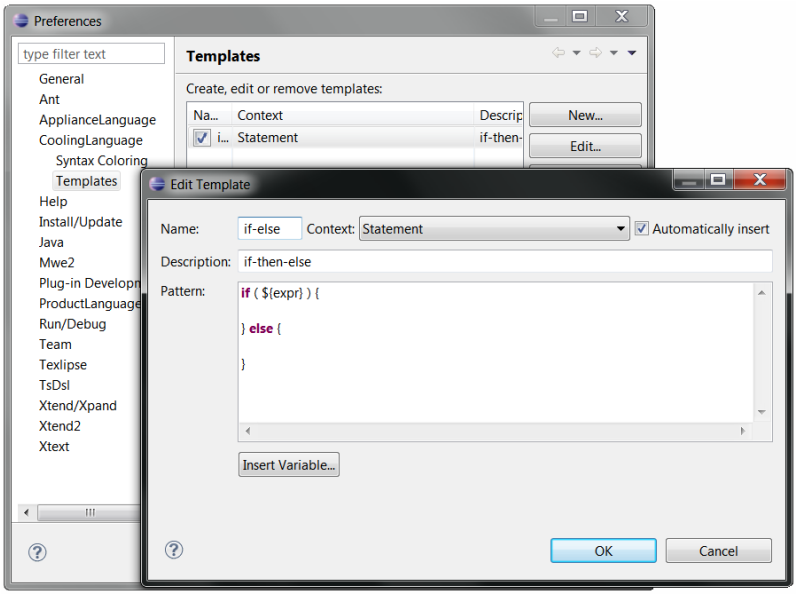


Figure 7.3: Template definitions contain a name (the text shown in the code completion menu), a description as well as the context and the pattern. The context refers to a grammar rule. The template will show up in the code completion menu at all locations where that grammar rule would be valid as well. The pattern is the actual text that will be inserted into the editor if the template is selected. It can contain so-called variables. Once inserted, the user can use `Tab` to step through the variables and replace them with text. In the example, we define the condition expression as a variable.

```
menu parts:
replace node (custom node concept)
   replace with : Statement
generic item
   matching text : if-then-else
   handler       : (node, model, scope, operationContext)->void {
                     node.replace with(<if (true) {
                       <no statements>
                     } else {
                       <no statements>
                     }>);
                   }
```

Figure 7.4: A cell menu to insert the if-then-else statement. Note how we contribute *two* menu parts. The first one inserts the default code completion contents for `Statement`. The second one provides an if/then/else statement under the menu text `if-then-else`. Notice how we can use a quotation (concrete syntax expression) in the cell menu code. Because of MPS's support for language composition, the editor even provides code completion etc. *for the contents of the quotation* in the editor for the cell menu.

for example. These keywords are readily 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 grey instead of black.

■ *An example with MPS* Let us first look at syntax coloring in MPS, starting with purely syntactic highlighting. Fig. 7.5 shows a collage of several ingredients: at the top we see the editor for `GlobalVariableDeclaration`. `GlobalVariableDeclaration` implements the interface `IModuleContent`. `IModuleContents` can be exported, 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` (it is also embedded into the editor of all other concepts that implement `IModuleContent`). The editor component simply defines a keyword `exported` that is rendered in dark

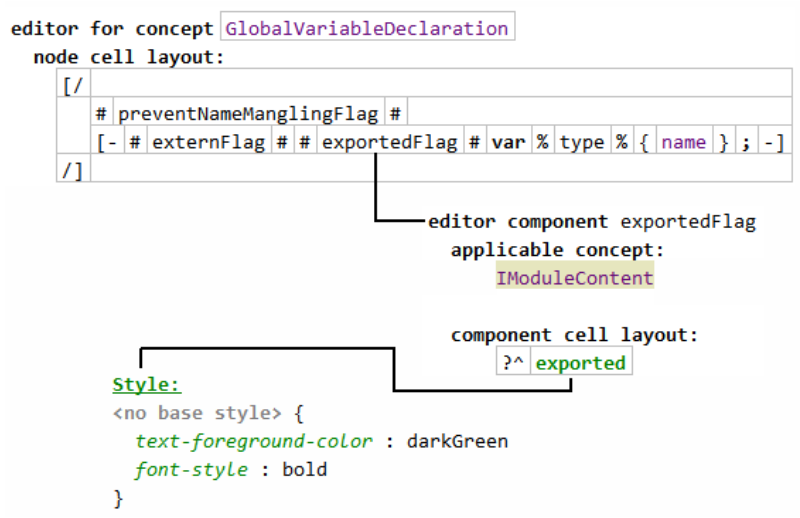


Figure 7.5: Syntax coloring is achieved by simply associating one or more style properties with the elements at hand. In this case we assign a `darkGreen` text foreground color as well as a `bold` font style.

green and in bold font. This can be achieved by simply specifying the respective style properties for the editor cell[4].

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. 7.6 renders a the `state` keyword of a `State` in a *Statemachine* grey if that particular state has no incoming transitions.

[4] Groups of style definitions can also be modularized into style sheets and reused for several cells.

```
Style:
<no base style> {
  text-foreground-color :
     (node, editorContext)->Color {
       boolean hasIncoming = node.ancestor<concept = Statemachine>.
           descendants<concept = Transition>.any({~it =>
             it.targetState == node; });
       if (hasIncoming) {
         return Color.black;
       } else {
         return Color.gray;
       }
     }
}
```

Figure 7.6: A style query that renders the associated cell in grey if the `state` (to which the cell belongs) has no incoming transitions. We first find out if the state has incoming transitions by finding the `Statemachine` ancestor of the state, finding all the `Transitions` in the subtree under the `Statemachine`, and then checking if one exists whose `targetState` is the current state (`node`). We then use the result of this query to color the cell appropriately.

■ *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 a 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. 7.7).

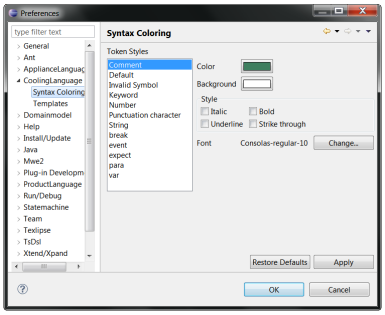


Figure 7.7: Preferences dialog that allows users to change the styles registered with the framework from a highlighting configuration.

We now have to associate the style with program syntax. The semantic highlighting calculator for the target language is used to this end[5]. It requires a method `provideHighlightingFor` 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(),
                                     CLHighlightingConf[illegible]on.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* symbol (variable, event, hardware element) with a `SymbolRef`, so we now have to check whether we reference a `Variable` or not[6]. If we have successfully identified a reference to a 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 colored. The

[5] Even though it is called *semantic* highlighting calculator, it is used for syntactic and semantic highlighting. It simply associates concrete syntax nodes with styles; it does not matter how it establishes the association (statically or based on the structure of the AST).

[6] This is the place where we could perform any other structural or semantic analysis (such as the check for no incoming transitions) as well.

concrete syntax tree in Xtext is a complete representation of the parse result, including keywords, symbols and whitespace[7]. 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 before. Notice how we color the *complete* reference. Since it is only one text string anyway, this is just as well. If we had more structured concrete syntax (as in `state someState {}`), and we only wanted to highlight parts of it (e.g. the `state` keyword), we'd have to do some further analysis on the `ICompositeNode` to find out the actual concrete syntax node for the keyword.

[7] It is represented as a [illegible] model as well so we can access it [illegible] the usual means.

■ *An example with Spoofax* In Spoofax, syntax coloring can be specified declaritively on the lexical and on the syntactic level. Both specifications need to be part of the editor specification, preferably in `<LanguageName>-Colorer.esv`. 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 the `int` type in declarations with a dark green:

```
module DSLbook-Colorer

imports DSLbook-Colorer.generated

colorer

  Type.IntType: darkgreen
```

Here, `Type` is a sort from the syntax definition, while `IntType` 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 with a 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 with a dark green, independent of their constructor. Similarly, we can use a wildcard for sorts. For example, `_.IntType` will include all nodes with a constructor `IntType`, independent of their syntactic sort.

In the current example, predefined types like `int` and entity types are all colored in a dark green, but only the predefined types will appear in a 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.

7.3 *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.

■ *Customizing the Target with Xtext* Let us first take a 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[8]. 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)
- and 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.

[8] Hyperlinking gets its name from the fact that, as you mouse over an element while keeping the `Ctrl` key depressed you see the respective element turn blue and underlined. You can the click on to it to follow the hyperlink

To add hyperlinks to a language concepts, Xtext provides the `IHyperlinkHelper`. It can be implemented by language developers to customize hyperlinking behavior. It requires one method, `createHyperlinksTo` to be

implemented. Typically, language developers will inerit from one of the existing base classes, such as the `TypeAwareHyperlinkHelper`. 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 im 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* a an interface (so other components can call the interface's operation) or it 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 so-called finders to achieve this. Fig. 7.8 shows the resulting Find Usages dialog for an `Interface` after we have added two finders to the language: one for components providing the interface and one for components requiring the interface.

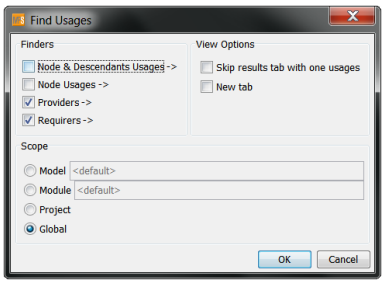


Figure 7.8: In the Find Usages dialog for `Interfaces`. The two additional Finders in the top left box are contributed by the language.

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

```
simple finder findProviders for concept Interface
  description: Providers

  find(node, scope)->void {
    nlist<> nodes = execute NodeUsages ( node , <same scope> );
    foreach n in nodes {
      if (n.isInstanceOf(ProvidedPort)) {
        add result n.parent ;
      }
    }
  }

  getCategory(node)->string {
    "Providers";
  }
```

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 simply check if the

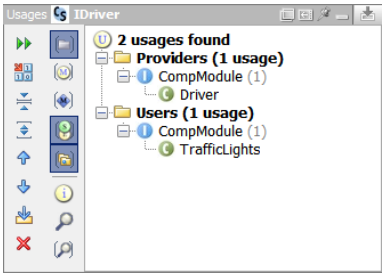


Figure 7.9: The result dialog of running Find Usages with our customized finders. Note the `Providers` and `Users` categories; these correspongf to the strings returned from `getCategory` in the two finders.

referencing element is a `ProvidedPort`, and if so, we add the parent of the port, i.e. a `Component`, to the result[9]. Finally, `getCategory` returns a string that is use to structure the result. Fig. 7.9 shows an example result.

[9] Note how we make use of extensions to the MPS BaseLanguage to concisely specify finders: `execute` and `add result` are only available in the finder specification language.

■ *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, preferably in `<LanguageName>-References.esv`. 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 customized[10]. On the right-hand side of the colon, the rule names a rewrite rule which implements the hyperlinking:

[10] As in colorer specifications, we can use _ as a wildcard for syntactic sorts and constructors.

```
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 type. Like all rewrite rules implementing hyperlinking functionality, it needs to follow a Spoofax-defined signature: on the left-hand side, it matches a tuple consisting of the `link`, the `position` of this node in the abstract syntax tree, the tree itself (`ast`), the `path` of the current file, and the path of the current Eclipse project (`project-path`). On the right-hand side, it provides the target of the hyperlink.

7.4 *Pretty Printing*

Pretty printing refers to the reverse activity from parsing[11]. 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.

[11] It is also known as *serialization*.

So when and where is a formatter useful? There is the obvious use case: users somehow screw 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 next paragrap) or by a graphical editor that operates in parallel to a text editor on the same AST.

■ *Pretty Printing in MPS* is a non-issue. The editor always pretty prints as part of the projection[12].

[12] On the flipside, MPS users do not have the option of changing the layout or formatting of a program, since the projection rules implement the one true way of formatting. Of course, this can be considered a plus or a minus, depending on the context.

■ *Pretty Printing in Spoofax* Spoofax generates a language-specific rewrite rule `pp-<LanguageName>-string` in `lib/editor-common.generated` which rewrites an abstract syntax tree into a string according to a pretty-printer definition. Spoofax generates a default pretty-printer definition in `syntax/<LanguageName>.generated.pp` from the syntax definition of a language. For example, Spoofax generates the following pretty-printer definition from the Mobl syntax definition, expressed in the Box language:

```
 @PhdThesis{deJonge:2003:TROTBR,
  author =       "Jonge, Merijn de",
  title =        "To Reuse or To Be Reused: Techniques for Component Composition and Construction",
  school =       "Faculty of Natural Sciences, Mathematics, and
                  Computer Science, University of Amsterdam",
  year =         2003,
  month =        jan
}
chapter 4
```

```
[
   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,
   IntType                  -- 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
]
```

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`

Figure 7.10: Selector types in pretty-printing rules for nested constructs in Spoofax.

In this 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 language. Since font operators are only meaningful when pretty-printing to HTML or LaTeX, we do not dive into the details here.

Numbers on the right-hand side can be used to combine boxes from the subtrees. Here, 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. Fig. 7.10 shows all nesting constructs in syntax definitions and their corresponding types in pretty-printing rules.
Additionally, user-defined pretty-printing rules can be defined in `syntax/<LanguageName>.pp`. 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]
```

The `H` box operator layouts sub-boxes horizontally. 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. Similarly, the `V` box operator places sub-boxes vertically. 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. For indenting boxes in a vertical combination, the spacing option `is` can be specified. All boxes

except the first will be indented accordingly.

■ *Pretty Printing in Xtext* In Xtext, whitespace in the grammar is irrelevant. In other words, Xtext cannot infer the "correct" use of whitespace from the grammar. Consequently, the use of whitespace has to be specified explicitly. This is done in a language's `Formatter`. Formatters use a Java API to specify whitespace policies for a grammar. Let us 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 it to format it to look 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. It is the basis for an internal Java DSL for expressing formatting rules. Let us 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. And 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 expresses a wrap before the closing curly brace as well as a decrement in the indentation level[13].

[13] As you can see, specifying the formatting for a complete grammar can become a lot of code! In my opinion, there are two approaches to improve this: one is reasonable defaults or global configurations. Curly braces, for example, are typically formatted the same way. Second, a more efficient way of specifying the formatting should be provided. Annotations in the grammar, or a DSL for specifying the formatting (such as the Box language used by [illegible]x) should go a long way.

7.5 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 violation[14].

[14] Notice that a quick fix only makes sense for problems that have one or more "obvious" fixes. This is not true for all problems.

■ *Quick fixes in Xtext* Xtext supports quick fixes for constraint violations. Quick fixes can either be implemented using the concrete syntax (i.e. via text replacement) or via the abtract syntax (i.e. via a model modification and subsequent serizalization. As an example, consider the following constraint defined in the cooling language's `CoolingLanguageJavaValidator`:

```
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 );
    }
}
```

Based on our discussion of constraint checks (in), this code should be fairly self-explaing. 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, implemented in the `CoolingLanguageQuickfixProvider`[15]. Notice how in the `@Fix` annotation we refer to the same constant that was used in the constraint check.

[15] This code resides in the UI part of the language, since, in contrast to the constraint check, it is relevant only in the editor.

```
@Fix(CoolingLanguageJavaValidator.VARIABLE_LOWER_CASE)
public void capitalizeName(final Issue issue, IssueResolutionAcceptor acceptor) {
    acceptor.accept(issue, "Deapitalize 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 though[16]. For these cases, one can use an instance of `ISemanticModification` instead:

[16] Imagine a problem that requires changes to the model in several places. Often it is easy to navigate to these places via the abstract syntax (following references, climbing up the tree), but finding the respective locations on the concrete syntax would be cumbersome and brittle

```
    @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 problem[17].

[17] Notice that after the problem is solved, the changed AST is serialized back into text. Depending on the scope of the change, a formatter has to be implemented for the language to make sure the resulting serialized text looks ok.

■ *Quick Fixes in MPS* Quick fixes work essentially the same way as in Xtext. Of couse 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 letter:

```
checking rule check_INameAllUpperCase {
  applicable for concept = INameAllUpperCase as a

  do {
    if (!(a.name.equals(a.name.toUpperCase()))) {
      warning "name should be all upper case" -> 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[18]. Quick fixes are executed by selecting them from the intentions menu (`Alt-Enter`).

[18] Note that the separation between core language and UI concerns is not as clear in MPS as it is in Xtext. This is mainly due to the fact that the language and the workbench/editor are inseparable in the first place

```
quick fix fixAllUpperCase

arguments:
  node<IIdentifierNamedConcept> node

description(node)->string { "Fix name"; }

execute(node)->void {
    node.name = node.name.toUpperCase();
}
```

■ *Model Synchronization via Quick Fixes* A particularly interesting feature of MPS' quick fixes is that they can be executed *automatically*. 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.

■ *Intentions* As discussed in , MPS also has intentions. These are essentially quick fixes that are not associated with an error. Instead, they can be invoked on any instance of the concept for which the intention is declared.

7.6 Refactoring

Refactoring concerns 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, structured refactorings are still useful.

■ *Renaming in Xtext* One of the most essential refactorings is renaming a program element. The reason why it is a refactoring (and not just typing a new name) is because all references to this element have to be updated. In textual languages, such references are by name, and if the name of the target element changes, so has the text of the reference. Xtext comes with a rename refactoring that handles this. Every language supports a rename refactoring automatically. The only thing the user has to remember is to not just type a new name, but instead invoke the Rename refactoring, for example with via `Ctrl-Alt-R`.

Note that in a projectional editor such as MPS renaming is not even a refactoring. A reference is established with the UUID of the target element. Renaming it does not lead to any structural change. And since the editor for the *referencing* element defines how to render the reference, it will just display the updated name in case it changes.

■ *Renaming in Spoofax* Like code generators, refactorings need to be specified in the editor specification (preferably in `<LanguageName>-Builders.esv`) and implemented with rewrite rules. For example, the following specification specifies a refactoring for renaming entities:

```
refactorings

  refactoring Decl.Entity : "Rename Entity" = rename-entity (cursor)
    shortcut : "org.eclipse.jdt.ui.edit.text.java.rename.element"
    input
      identifier : "new name" = ""
```

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. These options are the same as for code generators. 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 keybinding 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 follow a certain 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 pathes 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. Thereby, the new abstract syntax tree is retrieved by traversing the old on in a `topdown` fashion, `try`ing to apply rewrite rules `rename-entity-local`[19] These rules take the old and new entity name as parameters. They take care that declarations and references to entities are renamed. The first rule rewrites entity declarations, while

[19] `try(s)` tries to apply a strategy `s` to a term. Thereby, it never fails. If `s` succeeds, it will return the result of `s`. Otherwise, it will return the original term.

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 this one.

■ *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_t someFunction(int8_t v) {
  int8_t 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 could look like this:

```
int8_t someFunction(int8_t v) {
  int8_t product = v * FACTOR;
  int8_t y = somethingElse(product);
  if ( product > 20 ) {
    return 1;
  } else {
    return 0;
  }
}
```

The *Introduce Local Variable* refactoring performs just this change. MPS provides a DSL for specifying refactorings, based on which the implementation is ca. 20 lines of code. We'll go through it in steps. We start with the declaration of the refactoring itself.

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

The code above specifies the name of the refactoring (`introduce-LocalVariable`), the label used in the refactoring menu, the keystroke to execute it directly (`Ctrl-Alt-V`) as well as the target. The i.e. the language concept on which the refactoring can be executed. In our case, we want to refactor `Expressions`, but only if these expressions

are used in a `Statement` (we cannot refactor an expression if it is used, for example, as the `init` expression for a global constant). We 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 parameter. The `ask for` epression returns `false` if the user selects `Cancel` in the dialog that prompts the user for the name. The execution of the refactoring stops in this case.

We are now ready to implement the refactoring algorithm itself in the `refactor` block. We first declare two local variables that represent the expression on which we invoked the refactoring. We can get it from the `refactoringContext`[20]. We then grab the `Statement` under which this expression is located. Finally, we get the `index` of the `Statement` (`.index` returns the index of an element in its owning collection).

[20] In case the refactoring was declared to `allow multiple`, we can use `refactoringContext.nodes` to access all of the selected nodes.

```
node<Expression> targetExpr = refactoringContext.node;
node<Statement> targetStmt = targetExpr.ancestor<concept = 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 us *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<concept = Expression> {
      if (MatchingUtil.matchNodes(targetExpr, e)) {
        matchingExpressions.add(e);
      }
    }
  }
}
```

The next step is to actually introduce the new local variable. We simply create a new `LocalVariableDeclaration` using the tree 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 on which contains our target expression. We use the very handy `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 occurences 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 collection[21]:

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

[21] Note how the actual replacement is done with the `replace with` built-in function. It comes in very handy since we don't have to manually find out in which property or collection the expression lives in order to replace it.

All in all, building refactorings is straight forward 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.

7.7 *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`. To define the 
text, you simply override the `text` method for your element. For the icon, override the `image` method. Here are a couple of examples from the cooling langauage:

```
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";
    }
}
```

Notice how the label and the image are defined via methods, you can can change the text and the icon dynamically, based on some property of the model.

■ *Labels and Icons in MPS* Labels are defined by overriding the `getPresentation` behavior method on the respective concept. This way, the label can also be adjusted dynamically. The icon can be selected in the inspector (see Fig. 7.11) if we select a language concept. The icon is fixed and cannot be changed dynamically.

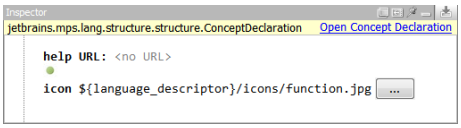


Figure 7.11: Assigning an icon to a language concept

7.8 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. 7.12.

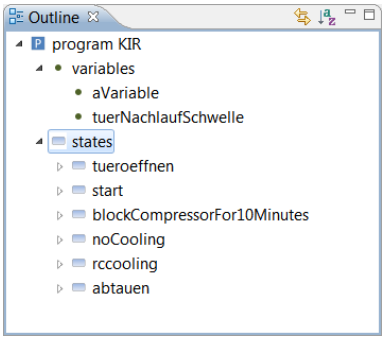


Figure 7.12: A customized outline view for cooling programs in Xtext

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 `_createChildren`:

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

```
    }
}
```

Inside the method, we first grab all the `CoolingPrograms` 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 node[22]. We then do the same for tests.

[22] The text and icon for the outline node is taken from the label provider discussed in the previous section.

Inside a program, we want to show variabes and states in separate sections, i.e. under separate intermediate nodes (see Fig. 7.12). 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);
    }
}
```

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, preferably in `<LanguageName>-Outliner.esv`. Abstract syntax tree nodes, which should appear in the outline, are selected based on their syntactic sort and constructor names. For example, the following outline specification will include all entity declarations:

```
module MoblLang-Outliner

imports MoblLang-Outliner.generated

outliner Entity Outliner

  Decl.Entity
```

Like in the specification of other editor services, we can use `_` as a wildcard for sorts and constructors. For example, `Decl._` will include imports and entities in the outline. Similarly, `_.Property` will include all nodes with a constructor `Property`, independent of their syntactic sort.

Spoofax analyses the syntax definition and tries to come up with a reasonable default outline specification, which is imported into `<LanguageName>-Outliner.esv`. We can then either extend the generated specification with our own rules, or discard it by removing the import statement.

■ *The Outline in MPS* MPS does not have a cutomizable 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 arbitrary additional views (called *tools* in MPS) to MPS. We discuss this briefly below[23].

[23] The MPS tutorial at `http://bit.ly/xU78ys` shows how to implement your own outline view.

7.9 Code Folding

Code folding refers to the small minuses in the gutter of an editor that let you collapse code regions (see Fig. 7.13). The editor shows an ellipsis (...) for the folded parts of the code. Clicking on the + or on the ellipsis restores the full code.

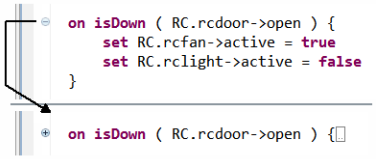


Figure 7.13: Code folding in Xtext. If you hover over the folded code, a popup shows the hidden code.

■ *Folding in Xtext* Xtext automatically provides folding for all language concepts that stretch over more than one line. To turn off this default behavior, you have to implement your own subclass of `DefaultFoldingRegionProvider` and overwrite `isHandled` in a suitable way. For example, to *not* provide folding for `CustomStates`, 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, preferably in `<LanguageName>-Folding.esv`. Very similar to the specification of outlines, folding is specified in terms of syntactic sort and constructor names:

```
module Mobl-Folding

folding Entity folding

  Module._
  Decl.Entity
  _.Function
```

Like for outlines, Spoofax analyses the syntax definition and tries to come up with a reasonable default specification, which is imported into `<LanguageName>-Folding.esv`. We can then either extend the generated specification with our own rules, disable particular specifications by adding a `(disabled)` annotation, or discard it completely by removing the import statement. A `(folded)` annotation tells Spoofax to fold a node by default in an opening editor, which is typically seen for import sections.

■ *Folding in MPS* In MPS, folding can be activated for any vertical collection. For example, in a state machine, each state contains a vertical list of transitions. Fig. 7.14 shows the definition of the `State` editor. It contains the `transitions` collection, which is an indent collection with the option `new-line-children` set. This arranges the transitions vertically.

To enable folding for this collection, we simple set the `uses folding` property for the collection to `true`. It can also be set to `query`, in which case code can be written that determines at runtime whether folding should be enabled or not. For example, folding could be enable if there are more than three transitions in the state. Once we've set the property to `true`, we have to provide a cell that is rendered in case the user selects to fold the code. This way the text shown as an ellipses can be customized beyond just showing three dots.
As Fig. 7.15 shows, we use a `read only model access` cell, which allows us to return an arbitrary string. In the example, we essentially output the number of "hidden" transitions.

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. 7.16 shows an list/tree of requirements.

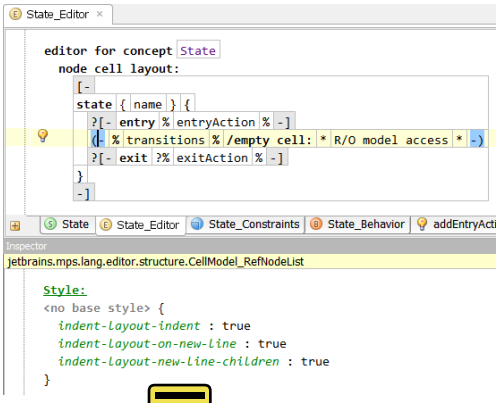


Figure 7.14: C lding in Xtext. If you hover over the folded code, a popup shows the hidden code.

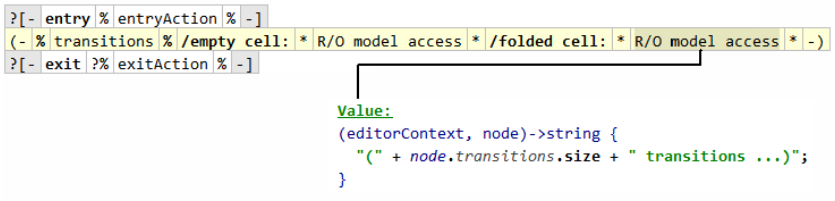


Figure 7.15: Folding enabled for states

After pressing `Ctrl-Shift-D` on a requirement, the editor shows the requirements details (Fig. 7.17). This effect of "expanding editors" is implemented by making the detail part optional in the sense that the projection rule only shows it conditionally. Fig. 7.18 shows the editor definition.

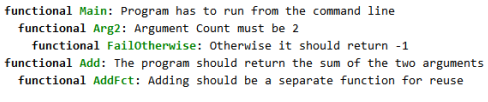


Figure 7.16: A list/tree of requirements

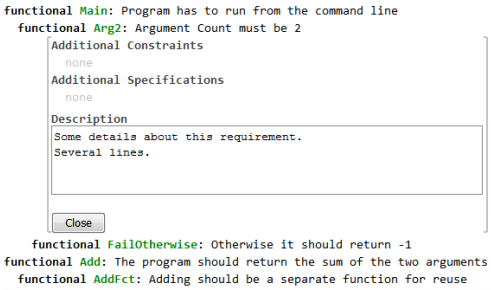


Figure 7.17: Optionally, the details about a requirement can be shown online in the editor.

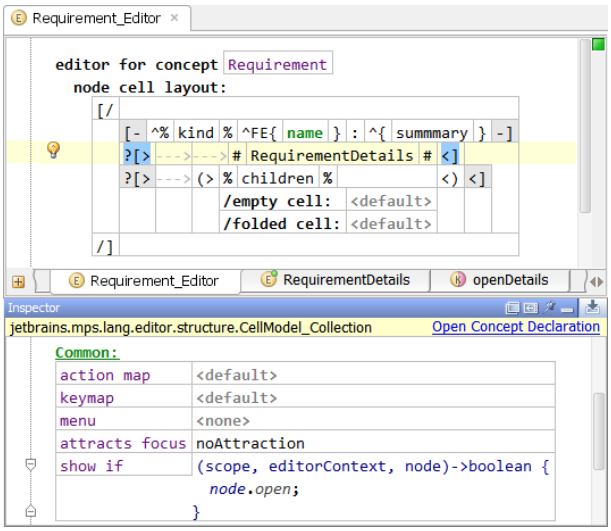


Figure 7.18: The part of the editor that includes the details pane is only projected if the `open` property is true. This property is toggled using `Ctrl-Shift-D`.

7.10 *Diff and Merge*

Highlighting the differences between versions of a program and the allowing the resolution of conflicts is important in the context of version control integration. For tools like Xtext that store models as ASCII text this is a non-issue: existing 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. Diff and merge has to be performed on the concrete projected syntax.

MPS provides this feature. MPS also annotates the editor with gutter annotations that highlight whether a part of the program has changed relative to the last checkout.

7.11 *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.

■ *Xtext* While Xtext does not support tooltips directly, the tooltip framework provided by JFace can be used.

■ *Spoofax* Spoofax supports tooltips directly. Tooltips are provided by rewrite rules, which need to be defined as hovers in the editor specification, preferable in `<LanguageName-References.esv>`:

```
hover _: editor-hover
```

This line tells Spoofax to use a rewrite rule `editor-hover` to retrieve tooltips for all kinds of abstract syntax tree nodes. 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 typable node:

```
editor-hover:
    (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: the 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. 7.19).

Looking at the editor definition for a `RequirementRef`, you can see that the actual editor (top in Fig. 7.20) shows only the name of the referenced element. The bottom part, the `inspected cell layout` projects the details about the referenced element.

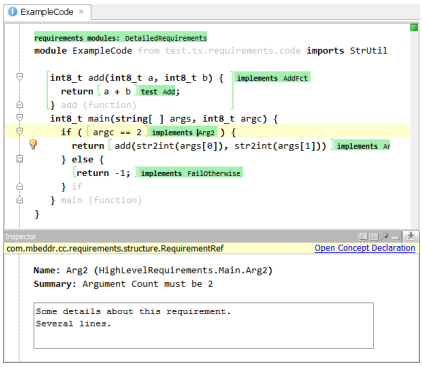


Figure 7.19: If a user selects a reference to a requirement in a requirements trance (`Arg2` in the example), the inspector shows information about the referenced requirement.

```
editor for concept RequirementRef
  node cell layout:
    ( % req % -> { name } )

  inspected cell layout:
    [/
      [- Name: ( % req % -> { name } ) ( * R/O model access * ) -]
      [- Summary: ( % req % -> { summmary } ) -]
      <constant>
      $swing component$
    /]
```

Figure 7.20: The editor definition for the `RequirementRef` projects details about the referenced element in the inspector. Notice the use of the `$swing component$` as a means to embed the Swing `JTextArea` that shows the prose description of the requirement.

7.12 *Visualizations*

To provide an overview over the structure of the programs, graphical representations are useful. Note that these are not necessarily a workaround for not having graphical editors. Visualizations can provide real added value.

■ *Xtext* In Xtext, Jan Koehnlein's Generic Graph View[24] can be used to render digrams 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, which means you can use Xbase expressions to traverse and query the model you want to visualize (in Fig. 7.21 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.

■ *MPS* In MPS we have integrated ZGRViewer[25], a Java-based renderer for GraphViz[26] dot files.

As part of the transformations, we map the model to 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.

[24] http://bit.ly/AluxnB

```
diagram EClassHierarchy type EClass {
  node EClassNode for this {
    label Name for this
    edge SuperType for each this.getESuperTypes() {
      => call EClassNode for this
    }
  }
}
```

```
stylesheet EClassHierarchy
  for EClassHierarchy

style EClassNode.SuperType {
  var arrow = new PolygonDecoration()
  arrow.setScale(10,10)
  arrow.backgroundColor = color(#ffffff)
  arrow.lineWidth = 2
  this.targetDecoration = arrow
}

style EClassNode.Name {
  font = font("Helvetica", 13,
    if (element.abstract) 3 else 1)
}
```

Figure 7.21: A model-to-graph mapping and a style definition expressed with the Generic Graph Viewer DSLs by Jan Koehnlein.

[25] http://bit.ly/qvJJ2

[26] http://bit.ly/2BqWbh

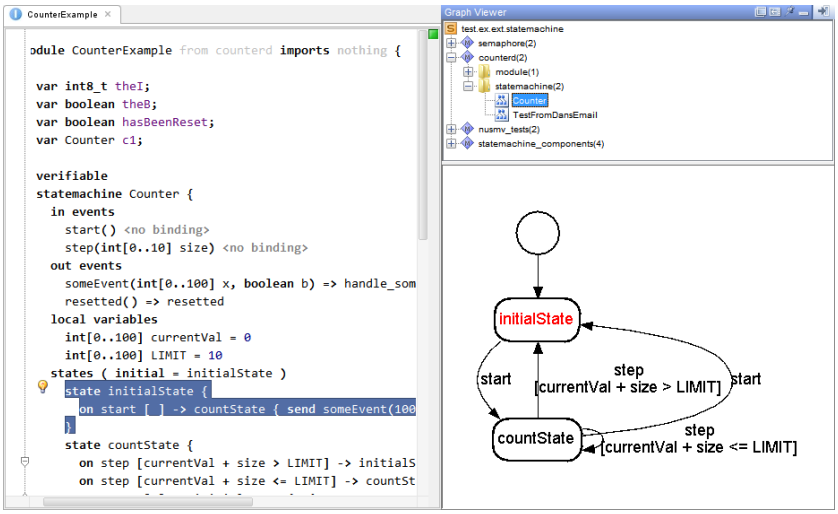


Figure 7.22: Clicking on a node in the graphview opens the respective program element in the MPS editor.

8
Testing DSLs

All the aspects of DSL implementation we have discussed so far need to be tested to keep them stable. In this chapter we address testing of the language syntax, the constraints, the semantics as well as some of the editor services based on examples with Xtext, MPS and Spoofax. We conclude the chapter with a brief look at "testing" a language for appropriateness relative to the domain.

DSL testing is a multi-faceted problem. DSL testing needs to address all the aspects of the DSL implementation we have discussed so far. In particular, this includes the syntax, the constraints and type system as well as the execution semantics (i.e. transformations or interpreters). Here are some examples:

- can the syntax cover all required sentences? Is the concrete syntax "correct"?
- do the scopes work correctly?
- do the constraints work? are all "wrong" programs actually detected, and is the right error message attached to the right program element?
- are the semantics correct? do transformations, generators and interpreters work correctly?
- can all programs relevant to the users actually be expressed? Does the language cover the complete domain?

An important ingredient to testing is that test execution can be automated via scripts so they can be run as part of automatic builds. All of the test strategies shown below can be executed from `ant` scripts. However, we don't describe in detail how this works for each of the tools.

8.1 Syntax Testing

Testing the syntax is simpxle in principle. Developers simply try to write all relevant programs and see if they can be expressed with the

language. If not, the concrete syntax is incomplete. We may also want to try to write "wrong" programs and check that the errors are detected, and that meaningful error messages are reported.

■ *An Example with Xtext* The following piece of code is the fundamental code that needs to be written in Xtext to test a DSL program using the Xtext testing utilities[1]. It is a JUnit 4 test case with special support for the Xtext infrastructure.

[1] http://code.google.com/a/eclipselabs.org/p/xtext-utils/wiki/Unit_Testing

```
@RunWith(XtextRunner.class)
@InjectWith(CoolingLanguageInjectorProvider.class)
public class InterpreterTests extends XtextTest {

    @Test
    public void testET0() throws Exception {
        testFileNoSerializer("interpreter/engine0.cool", "tests.appl", "stdparams.cool" );
    }

}
```

The single test method loads the `interpreter/engine0.cool` program, as well as two more files which contain elements referenced from `engine0.cool`. The `testFileNoSerializer` method loads the file, parses it, and checks constraints. If either parsing or constraint checking fails, the test fails. There is also a `testFile` method which, after loading and parsing the file, reserializes the AST to the text file, writes it back, and loads it again, the comparing the two ASTs. This way, the (potentially adapted) formatter is tested[2].

[2] For testing the formatting, a text comparison is the only ways to go, even though we argue against text comparison in general.

On a more fine grained level it is often useful to test partial sentences instead of complete sentences or programs. The following piece of Xtext example code tests the `CustomState` parser rule:

```
@Test
public void testStateParserRule() throws Exception {
    testParserRule("state s:", "CustomState" );
    testParserRule("state s: entry { do fach1->anOperation }", "CustomState" );
    testParserRule("state s: entry { do fach1->anOperation }", "State" );
}
```

The first line asserts that the string `state s:` can be parsed with the `CustomState` parser rule. The second line passes in a more complex state, one with a command in an entry action. Line three tries the same text with the `State` rule, which itself calls the `CustomState`. Notice that these tests really just test the parser. No linking or constraints checks are performed. This is why we can "call" `anOperation` on the `fach1` object, although `anOperation` is not defined as a callable operation anywhere.

■ *An Example with Spoofax* Spoofax supports writing tests for language definitions using a testing language. Consider the following

test suite:

```
module example
language MoblEntities

test empty module [[module foo]] parse succeeds
test missing layout (module name) [[modulefoo]] parse fails
```

The first two lines specify the name of the test suite and the language under test. The remaining lines specify a positive and a negative test cases concerning the language's syntax. Each test case consists of a name, the to-be-tested code fragment in double square brackets, and a specification that determines what kind of test should be performed (`parsing`) and what the expected outcome is (`succeeds` or`fails`). We can also specify the expected abstract syntax based on the ATerm textual notation:

```
test empty module (AST) [[module foo]] parse to Module("foo", [])
```

Instead of specifying a complete abstract syntax tree, we can only specify the interesting parts in a pattern. For example, if we only want to verify that the definition list of an empty module is indeed empty, we can use _ as a wildcard for the module name:

```
test empty module (AST) [[module foo]] parse to Module(_, [])
```

Abstract syntax patterns are particularly useful for testing operator precedence and associativity:

```
test multiply and add [[1 + 2 * 3]] parse to Add(_, Mul(_, _))
test add and multiply [[1 * 2 + 3]] parse to Add(Mul(_, _), _)
test add and add [[1 + 2 + 3]] parse to Add(Add(_, _), _)
```

Alternatively, we can specify an equivalent context-syntax fragment instead of an abstract syntax pattern:

```
test multiply and add [[1 + 2 * 3]] parse to [[1 + (2 * 3)]]
test add and multiply [[1 * 2 + 3]] parse to [[(1 * 2) + 3]]
test add and add [[1 + 2 + 3]] parse to [[(1 + 2) + 3]]
```

A test suite can be run from the *Transform* menu. This will open the *Spoofax Test Runner View* which provides information about failing and succeeding test cases in a test suite . Additionally, we can also get

instant feedback while editing a test suite. Tests can also be evaluated outside the IDE, for example as part of a continuous integration setup.

■ *Syntax Testing with MPS* Syntax testing in the strict sense is not useful or necessary with MPS, since it is not possible to "write text that does not parse". Invalid programs cannot even be entered. However, it is useful to write a set of programs which the language developer consideres relevant. While it is not possible to write syntactically invalid programs, the following scenario is possible (and useful to test): a user writes a program with the language in version 1. The language evolves to version 2, making that program invalid. In this case, the program contains unbound language concepts or "holes". By running the model checker (interactively or via `ant`), such problems can be detected. Fig. 8.1 shows an example.

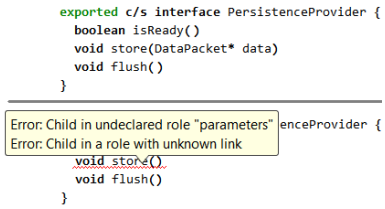


Figure 8.1: **Top:** an interface expressed in the mbeddr C components extension. **Bottom:** The same interface after we have removed the `parameters` collection in an `Operation`. The error reports that the model contains child nodes in a child collection that does not exist.

8.2 Constraints Testing

Especially for languages with complex constraints, such as those implied by type systems, testing of constraints is essential. The goal of constraints testing is to ensure that the correct error messages are annotated to the correct program elements, if those program elements have a type error.

■ *An Example with Xtext* A special API is necessary to be able to verify that a program which makes a particular constraint fail actually annotates the corresponding error message to the respective program element. This way, tests can then be written which assert that a given program has a specific set of error annotations.

The unit testing utilities mentioned above also support testing constraints. The utilities come with an internal Java DSL that supports checking for the presence of error annotations after parsing and constraint-checking a model file.

```
@Test
public void testTypesOfParams() throws Exception {
    testFileNoSerializer("typesystem/tst1.cool", "tests.appl", "stdparams.cool");
    assertConstraints( issues.sizeIs(3) );                              // 1
    assertConstraints( issues.forElement(Variable.class, "v1").         // 2
      theOneAndOnlyContains("incompatible type") );                     // 2
    assertConstraints( issues.under(Variable.class, "w1").              // 3
      errorsOnly().sizeIs(2).oneOfThemContains("incompatible type") ); // 3
}
```

We first load the model file that constains constraint errors (in this case, type system errors). Then we assert the total number of errors in the file to be three (line 1). This makes sure that the file does not contain

additional errors beyond those asserted in the rest of the test case. Next, in line 2, we check that the instance of `Variable` named `v1` has exactly one error annotation, and it has the text *incompatible type* in the error message. Finally, in line 3 we assert that there are exactly two errors anywhere under (i.e. in the subtree below) a `Variable` named `w1`, and one of these contains *incompatible type* in the error message. Using the fluent API style shown by these examples, it is easy to express errors and their locations in the program. If a test fails, a meaningful error message is output that supports localizing (potential) problems in the test. The following is the error message if no error message is found that contains the substring *incompatible type*:

```
junit.framework.AssertionFailedError: <no id> failed
  - failed oneOfThemContains: none of the issues
    contains substring 'incompatible type'
  at junit.framework.Assert.fail(Assert.java:47)
  at junit.framework.Assert.assertTrue(Assert.java:20)
  ...
```

A test may also fail earlier in the chain of filter expressions if, for example, there is no `Variable` named `v1` in the program. More output is provided in this case:

```
junit.framework.AssertionFailedError: <no id> failed
  - no elements of type
    com.bsh.pk.cooling.coolingLanguage.Variable named 'v1' found
  - failed oneOfThemContains: none of the issues
    contains substring 'incompatible type'
  at junit.framework.Assert.fail(Assert.java:47)
  ...
```

Scopes can be tested in the same way: we can write example programs where references point to valid targets (i.e. those in scope) and invalid targets (i.e. not in scope). Valid references may not have errors, invalid references must have errors.

■ *An Example with MPS* MPS comes with the `NodesTestCase` for testing constraints and type system rules (Fig. 8.2). It supports special annotations to express assertions on types and errors, directly in the program. For example, the third line of the nodes section in Fig. 8.2 reads `var double d3 = d` without annotations. This is a valid variable declaration in mbeddr C. After this has been written down, annotations can be added. They are rendered in green. Line three asserts that the type of the variable `d` is `double`, i.e. it tests that variable references assume the type of the referenced variable. In line four we assign a `double` to an `int`, which is illegal according to the typing rules. The error is detected, hence the red squiggly. We use another annotation to assert the presence of the error.

```
Test case testSubtyping
nodes
  ( [ <dnode  var double d = 10>                    ] )
    [ var double d2 = <dref  d>                     ]
    [ var double d3 = <node d has type double>      ]
    [ <node var int i = d has error>                ]

test methods
  test testReference {
    assert dref.var == dnode;
  }
```

Figure 8.2: Using MPS `NodesTestCase`, assertions about types and the presence of errors can be directly annotated on programs written in any language.

In addition to using these annotations to check models, developers can also write more detailed test cases about the structure or the types of programs. In the example we assert that the `var` reference of the node referred to as `dref` points to the node referred to as `dnode`. Note how labels (green, underlined) are used to add names to program elements so they can be referred to from test expressions. This approach can be used to test scopes. If two variables with the same name are defined (e.g. because one of them is defined in an outer block, and we assume that the inner variable shadows the outer variable of the same name) we can use this mechanism to check that a reference actually points to the inner variable. Fig. 8.3 shows an example.

```
Test case testShadowing
nodes
  ( [ module dummy imports nothing {  ] )
    [                                 ]
    [   void aFunction() {            ]
    [     <outerX  int8_t x = 0;>     ]
    [     if ( true ) {               ]
    [       <innerX  int8_t x = 0;>   ]
    [       <xRef  x>++;              ]
    [     } if                        ]
    [   } aFunction (function)        ]
    [ }                               ]

test methods
  test testShadowing {
    assert xRef.var == innerX;
  }
```

Figure 8.3: Labels and test methods can be used to check that scoping works. In this example, we check shadowing of variables.

■ *An Example with Spoofax* In Spoofax' testing language, we can write test cases which specify the number of errors and warnings in a code fragment:

```
test duplicate entities [[
  module foo
  entity X {}
  entity X {}
]] 1 error

test lower case entity name [[
  module foo
  entity x {}
]] 1 warning
```

Additionally, we can specify parts of the error or warning messages:

```
test duplicate entities [[
  module foo
  entity X {}
  entity X {}
]] 1 error /duplicate/
```

Like in Xtext and MPS, we can test scopes by means of correct and incorrect references. Alternatively, we can specify source and target of a link in a test case:

```
test property reference [[
  module foo
  entity X {
   [[p]]: int
   function f(q: int) {
     r: int = 0;
      return [[p]];
   }
  }

]] resolve #2 to #1

test parameter reference [[
  module foo
  entity X {
   p: int
   function f([[p]]: int) {
      r: int = 0;
      return [[p]];
   }
  }

]] resolve #2 to #1

test variable reference [[
  module foo
  entity X {
   p: int
   function f(q: int) {
      [[p]]: int = 0;
      return [[p]];
   }
  }

]] resolve #2 to #1
```

These cases use double square brackets to select parts of the program and specify the expected reference resolving in terms of these selections.

8.3 Semantics Testing

Fundamentally, testing the execution semantics of a program involves writing constraints against the *execution* of a program as it is executed. In the simplest case this can be done the following way:

- write a DSL program, based on an understanding what this program is expected to do
- generate the program into its executable representation
- *manually* write unit tests (in the target language) that assert the generated program's behavior based on the understanding of what the DSL program should do

Notice how we do *not* test the structure or syntax of the generated artifact. Instead we test it's meaning, which is exactly what we *want* to

test. An important variation of this appraoch is the following: instead of writing the unit tests manually in the target language, we can also write the tests in the DSL, assuming the DSL has syntax to express such tests[3]. Writing the test cases on DSL level results in more concise and readable tests.

[3] Many DSLs are explicitly extended to support this use case

Testing DSL programs by running tests expressed in the same language runs the risk of doubly-negating errors. If the generators for the tests and the core program in a compatible way, errors in either one may not be found. However, this problem can be alleviated by running a large enough number of tests.

The same approach can be used to test execution semantics based on an interpreter, although it may be a little bit more difficult to manually write the test cases in the target language; the interpreter must provide a means to "inspect" its execution so we can check if it is correct. If the tests are written in the DSL and the interpreter executes them along with the core program, the approach works well.

Strictly speaking, this approach tests the semantics of a *specific* program. As always in testing, we have to write many of these tests to make sure we have covered all of the possible executions paths through a generator or interpreter. If we do that, the set of tests implicitly tests the generator or interpreter — which is the goal we want to achieve in semantics testing.

If we have several execution backends, such as an interpreter *and* a compiler it must be ensured that both have the same semantics. This can be achieved by writing the tests in the DSL and then executing them in *both* backends. By executing enough tests, the semantic equivalence of the backends can be ensured.

■ *Testing an Interpreter with Xtext* The cooling language provides a way to express test cases for the cooling programs within the cooling language itself[4]. These tests are executed with an interpreter inside the IDE, and they can also be executed on the level of the C program, by generating the program *and* the test cases to C. Note that this approach cannot just be used for testing interpreters or generators, it can also be used to test whether a program written in the DSL works correclty. This is in fact why the interpreter and the test sub-language have been built in the first place: DSL users should be able to test the programs written in the DSL.

[4] Strictly speaking, tests are a separate viewpoint to keep them out of the actual programs.

The following code shows one of the simplest possible cooling programs, as well as a test case for that program:

```
cooling program EngineProgram0 for Einzonengeraet uses stdlib {

    var v: int
    event e1

    init { set v = 1 }

    start:
       entry { set v = v * 2 }
       on e1 { state s2 }

    state s2:
        entry { set v = 0 }
```

```
}

test EngineTest0 for EngineProgram0 {
    assert-currentstate-is ^start  // 1
    assert-value v is 2            // 2
    step                           // 3
    event e1                       // 4
    step                           // 5
    assert-currentstate-is s2      // 6
    assert-value v is 0            // 7
}
```

The test first asserts that, as soon as the program starts, it is in the `start` state (line 1). We then assert that `v` is 2. The only reasonable way how `v` can become 2 is that the code in the `init` block as well as the code in the entry action of the `start` start have been executed. Note that this arrangement even checks that the `init` block is executed before the entry action of the start state, since otherwise `v` would be 1! We then perform one step in the execution of the program in line 3. At this point nothing should happen, since no event was triggered. Then we trigger the event `e1` (line 4) and perform another `step` (line 5). After this step, the program must transition to the state `s2`, whose entry action sets `v` back to 0. We assert both of these in lines 6 and 7.

These tests can be run interactively from the IDE, in which case assertion failures are annotated as error marks on the program, or from within JUnit. The following piece of code shows how to run the tests from JUnit.

```
@RunWith(XtextRunner.class)
@InjectWith(CoolingLanguageInjectorProvider.class)
public class InterpreterTests extends PKInterpreterTestCase {

    @Test
    public void testET0() throws Exception {
        testFileNoSerializer("interpreter/engine0.cool", "tests.appl", "stdparams.cool" );
        runAllTestsInFile( (Model) getModelRoot());
    }
}
```

This is basically a JUnit test that inherits from a base class that helps with loading models and running the interpreter. We call the `runAllTestsInFile` method, passing in the model's root element. `runAllTestsInFile` is defined by the `PKInterpreterTestCase` base class, which in turn inherits from `XtextTest`, which we have seen before. The method iterates over all tests in the model and executes them by creating and running a `TestExecutionEngine`. The `TestExecutionEngine` is a wrapper around the interpreter for cooling programs that we have discussed before.

```
protected void runAllTestsInFile(Model m) {
```

```
    CLTypesystem ts = new CLTypesystem();
    EList<CoolingTest> tests = m.getTests();
    for (CoolingTest test : tests) {
        TestExecutionEngine e = new TestExecutionEngine(test, ts);
        final LogEntry logger = LogEntry.root("test execution");
        LogEntry.setMostRecentRoot(logger);
        e.runTest(logger);
    }
}
```

The cooling programs are generated to C for execution in the refrigerator. To make sure the generated C code has the same semantics as the interpreter, we simply generate C code from the test cases as well. This way, the same tests are executed against the generated C code. By ensuring that all of them work in the interpreter and the generator, we ensure that both behave the same way.

■ *Testing a Generator with MPS* The following is a test case expressed using the testing extension to mbeddr C. It contributes `test cases` to modules[5]. `testMultiply` is the actual test case. It calls the to-be-tested function `times2` several times with different arguments and then uses an `assert` to check for the expected value.

[5] Instead of using a separate viewpoint for expressing test cases, these are inlined into the same program in this case. However, the *language* for expressing test cases is a modular extension to C, to keep the core C clean.

```
module UnitTestDemo {

  int32_t main(int32_t argc, int8_t*[ ] argv) {
    return test testMultiply;
  }

  test case testMultiply {
    assert(0) times2(21) == 42;
    assert(1) times2(0) == 0;
    assert(2) times2(-10) == -20;
  }

  int8_t times2(int8_t a) {
    return 2 * a;
  }
}
```

Note that, while this unit testing extension can be used to test any C program, we use it a lot to test the generator. Consider the following example:

```
  assert(0) 4 * 3 + 2 == 14;
```

The result of `4 * 3 + 2` is only 14 if the correct operator precedences are observed (otherwise the result would be 20). One problem we had initially in mbeddr C was to make sure that the expression tree that was created while manually entering expressions like `4 * 3 + 2` is built correctly. If the tree were built in the wrong way, the generated code could end up as `4 * (3 + 2)` resulting in 20. So we've used

tests like these to implicltly test quite intricate aspects of our language implementation[6].

We have built much more elaborate support for testing various other extensions. It is illustrative to take a look at two of them. The next piece of code shows a test for a state machine:

[6] This is also the reason why the unit test extension was the first extension we've built for C. We needed it to test many other aspects of the language.

```
exported test case test1 {
  initsm(c1);
  assert(0) isInState<c1, initialState>;
  test statemachine c1 {
    start -> countState
    step(1) -> countState
    step(2) -> countState
    step(7) -> countState
    step(1) -> initialState
  }
}
```

`c1` is an instance of a state machine. After initializing it, we assert that it is in the `initialState`. We then use a special `test statemachine` statement, which consists of event/state pairs: after triggering the event (on the left side of the `->`) we expect the state machine to go into the state specified on the right side of the `->`. We could have achieve the same goal by using sequences of `trigger` and `assert` statements, but the syntax used here is much more concise.

The second example concerns mocking. A mock is a part of a program that can be used in place of the real one, to simulate some kind of environment of the unit under test. We use this with the components extension. The following is a test case that checks if the `client` uses the `PersistenceProvider` interface correctly. Let's start by taking a look at the interface:

```
interface PersistenceProvider {
  boolean isReady()
  void store(DataPacket* data)
  void flush()
}
```

The interface is expected to be used in the following way: clients first have to call `isReady`, and only if that method returns `true` are they supposed to call `store`, and then after any number of calls to `store`, they have to call `flush`. Let us assume now we want to check if a certain client component uses the interface correctly[7]. Assuming it provides an operation `run` that uses the persistence provider, we could write the following test:

[7] Our components language actually also supports protocol state machines which support the declarative specification of valid call sequences

```
exported test case runTest {
  client.run();
  // somehow check is behaved correctly
}
```

To check if the client behaves correctly, we can use a mock. Our mock specifies the *icoming* methods calls it expects to see during the test. We have provided a mocking extension to components to support this. Here is the mock:

```
exported mock component PersistenceMock {
  ports:
    provides PersistenceProvider pp
  expectations:
    total no. of calls is 4
    sequence {
      0: pp.isReady return false;
      1: pp.isReady return true;
      2: pp.store {
          0: parameter data: data != null
        }
      3: pp.flush
    }
}
```

The mock provides the `PersistenceProvider` interface, so any other component that `requires` this interface can use this component as the implementation. But instead of actually implementing the operations prescribed by `PersistenceProvider`, we specify the sequence of invocations we expect to see. We expect a total number of 4 invocations. The first one is expected to be to `isReady`. We return `false`, expecting that the client tries again later. If it does, we return `true` and expect the client to continue with persisting data. We can now validate the mock as part of the test case:

```
exported test case runTest {
  client.run();
  validate mock persistenceMock
}
```

In case the `persistenceMock` saw behavior different from the one specified above, the `validate mock` statement will fail — and with it, the whole test.

One particular challenge with this approach to semantics testing is that, if an assertion fails, you get some kind of `assertion XYZ failed at ABC` output from the running test case. To understand and fix the problem, you will have to navigate back to the `assert` statement in the DSL program. If you have many failed assertions or just generally a lot of test program output, this can be tedious and error prone. For example, the following piece of code shows the output from executing a C-based test case on the command line:

```
./TestHelperTest
$$runningTest: running test () @TestHelperTest:test_testCase1:0#767515563077315487
$$FAILED: ***FAILED*** (testID=0) @TestHelperTest:f:0#9125142491355884683
$$FAILED: ***FAILED*** (testID=1) @TestHelperTest:f:1#9125142491355901742
```

In mbeddr we have built a tool that simplifies finding the message source. You can paste arbitrary text that contains error messages into a text area (for example the example above) on the left in Fig. 8.4. Pressing the `Analyze` button will find the nodes that created a particular message[8]. You can then click on the node to select it in the editor

[8] This process is based on the unique node ID; this is the long number that follows the # in the message text.

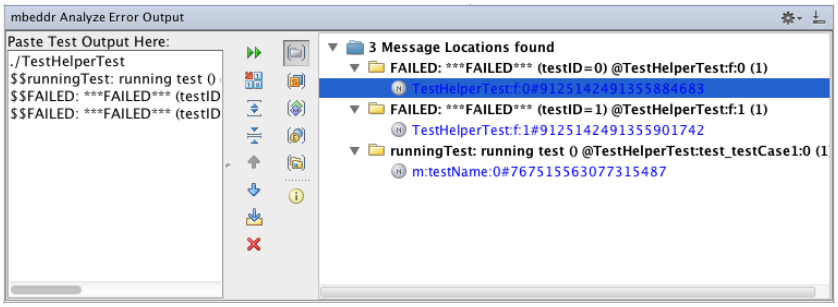


Figure 8.4: The mbeddr error output analyzer parses test output and supports navigating to the source of error messages in the MPS program.

■ *Testing Interpreters and Generators with Spoofax* Spoofax' testing language also supports testing transformations. We can rely on this, to test interpreters, assuming that the interpreter is implemented as a transformation from programs to program results. For example, the following tests address a transformation `eval-all` which interpretes expressions:

```
test evaluate addition [[1+2]] run eval-all to Int("3")
test evaluate multiplication [[3*4]] run eval-all to Int("12")
test complex evaluation [[1+2*(3+4)]] run eval-all to Int("15")
```

To test generators, we can rely on Spoofax' testing support for builders. Since we are not interested in testing in terms of the generated code, but in the behaviour of the generated code, we need a builder which generates code, runs the generated code, and finally yields the result of this run.

For example, the following tests address a builder `generate-and-run` which generates code from expressions, runs this code, and returns the result of the run as a string:

```
test generate addition [[1+2]] build generate-and-run to "3"
```

```
test generate multiplication [[3*4]] run generate-and-run to "12"
test generate evaluation 1 [[1+2*(3+4)]] run generate-and-run to "15"
```

■ *Structural Testing* What we suggested in the previous subsection tests the execution semantics of programs written in DSLs, and, if we have enough of these tests, it tests the correctness of the transformation, generator or interpreter. However, there is a significant limitation to this approach: it only works if the DSL actually specifies behavior! If the DSL only specifies structures and cannot be executed, the approach does not work[9]. In this case you have to perform a structural test. In principle, this is simple:

- You write an example model
- You generate it[10]
- And then you inspect the resulting model or test for the expected structures

[9] For testing execution semantics, this does not matter — if you cannot execute something because it has no behavior, then there is no execution semantics to test in the first place. But you also cannot use the approach to implicitly test generators or transformations then

[10] I have never seen an interpreter to process languages that only specify structures

Depending on the target formalism you can use regular expressions, XPath expressions or OCL-like expressions to achieve this`Remember that you should still automate this!`.

Note that you really should only use this if you cannot use semantics testing based on execution. Inspecting the generated C code for syntactic correctness, based on the input program, would be much more work. And if we evolve the generator to generate better (faster, smaller, more robust) code, tests based on the execution semantics will still work, while those that test the structure may fail because line numbers or variable names change.

Structural testing can also be useful to unit test model-to-model transformations[11]. Conside the example in Section 5.2.2. There, we inserted additional states and transitions into whatever input state machine our transformation processed. Testing this via execution invariably tests the model-to-model transformation as well as the generator (or interpreter). If we wanted to test the model-to-model transformation in isolation, we have to use structural testing, because the result of that transformation itself is not yet executable. The following piece of code could be used to check that, for a specific input program, the transformation works correctly:

[11] If a program is transformed to an executable representation in several steps, then the approach discussed above tests *all transformations in total*, so it is more like an integration test, and not a unit test. Depending on the complexity and the reuse potential of the the transformation steps, it may make sense to test them in isolation.

```
// run transformation
val tp = p.transform

// test result structurally
val states = tp.states.filter(typeof(CustomState))
assert( states.filter(s|s.name.equals("EMERGENCY_STOP")).size == 1 )

val emergencyState = states.findFirst(s|s.name.equals("EMERGENCY_STOP"))
```

```
states.findFirst(s|s.name.equals("noCooling")).eAllContents.
    filter(typeof(ChangeStateStatement)).
        exists(css|css.targetState == emergencyState)
```

This program first runs the transformation, and then grabs all `CustomStates` (those that are not start or stop states). We then assert that in those states there is exactly one with the name `EMERGENCY_STOP`, because we assume that the transformation has added this state. We then check that in the (one and only) `noCooling` state there's at least one `ChangeStateStatement` whose target state is the `emergencyState` we had retrieved above[12].

[12] Notice that we don't write an algorithmic check that closely resembles the transformation itself. Rather, we test a specific model for the presence of specific structures. For example, we explicitly look for a state called `noCooling` and check that this one has the correct `ChangeStateStatement`.

■ *Formal Verification* Formal verification is an alternative to testing in some cases. The fundamental difference between testing and verification is this: in testing, each test case specifies `one` particular execution scenario. To get reasonable or full coverage of the whole model or transformation, you have to write and execute a lot of tests. This can be a lot of work, and, more importantly, you may not think about certain (exceptional) scenarios, and hence you may not test them. Bugs may go unnoticed.

Verification checks `the whole program` at once. Various non-trivial algorithms are used to do that, and understanding these algorithms in detail is beyond the scope of this book. However, it is very useful to know these appraoches exist, especially since, over the last couple of years, they have become scalable enough to address real-world problems.

In this section we look at two examples: model checking and SAT solving.

■ *Model Checking State Machines* Model Checking is a verification technique for state machines. In this section we can only scratch the surface; to learn more about model checking, we recommend [13]. Here is how it works in principle:

[13]

- some functionality is expressed as a state machine
- you then specify *properties* of the state machine. Properties are expressions that have to `true` about the state machine
- you run the model checker with the state machine and the properties as input
- the output of the model checker either confirms that your properties hold, or it shows a counter example[14].

[14] It may also report Out Of Memory, in which case you have to reformulate or modularize your program and try again

Conceptually, the model checker performs an exhaustive search during the verification process. Obviously, the more complex your state

machine is, the more possibilities the checker has to address, a problem known as state space explosion. With finite memory, this limits scalability. In reality the model checker does *not* perform an exhaustive search; clever algorithms have been devised that are semantically equivalent to an exhaustive search, but don't actually perform one. This makes model checking scalable and fast enough for real-world problems, although there is still a limit in terms of input model complexity.

The interesting aspect of model checking is that the properties you specify are not just simple Boolean expressions such as *each state must have at least one outgoing transition, unless it is a stop state*. Such a check can be performed statically, by "just looking" at the model. The properties model checkers are more elaborate and are often typically expressed in (various flavours of) temporal logic. Here are some examples, expressed in plain English:

- *It is always true that after we have been in state X we will eventually be reaching state Y*. This is a so-called Fairness property. It ensures that the state machine does not get stuck in some state forever. For example, `state Y` may be the green light for pedestrians, and `state X` could be the green light for cars.
- *Wherever we are in the state machine, it is always possible to get into state X*. This is a Liveliness property. Consider a machine where you always want to be able to turn it off.
- *It is not ever possible to get into state X without having gone through state Y directly before*. This is a Safety property. Imagine a state machine where entering *state X* turns the pedestrian lights gree and entering `state X` turns the car lights red.

The important property of these temporal logic specifications is that quantifiers such as *always*, *whenever* and *there exists* are available. Using these one can specify *global truths* about the *execution* of a system[15].

Model checking does come with its challenges. The input language for specifying state machines as well as specifying the properties is not necessarily easy to work with. Interpreting the results of the model checker can be a challenge. And for some of the tools, the usability is really bad[16].

To make model checking more user friendly the mbeddr C language provides a nice syntax for state machines and then generates the corresponding representation in the input language of the model checker (we use the NuSMV[17] model checker). The replies of the model checker are also reinterpreted in the context of the higher level state machine. Tool integration is provided as well: users can select the context menu on a state machine and invoke the model checker.

[15] Three different flavours of languages exist to specify these properties: LTL, CTL and CTL+. The concrete syntax is beyond the scope of this book.

[16] the SPIN/Promela model checker comes to mind here!

[17] http://nusmv.fbk.eu/

The model checker input is generated, the model checker is executed, and the replies are rendered in a nice table in MPS. Finally, we have abstracted the property specification language by providing support for the most important idioms; these can be specified relatively easily. Also, a number of properties are automatically checked for each state machine.

Let us look at an example. The following code shows a state machine that represents a counter. We can send the `step` event into the state machine, and as a consequence, it increments the `currentVal` counter by the `size` parameter passed with the event. If the `currentVal` would become greater than `LIMIT`, the counter wraps around. We can also use the `start` event to reset the counter to 0.

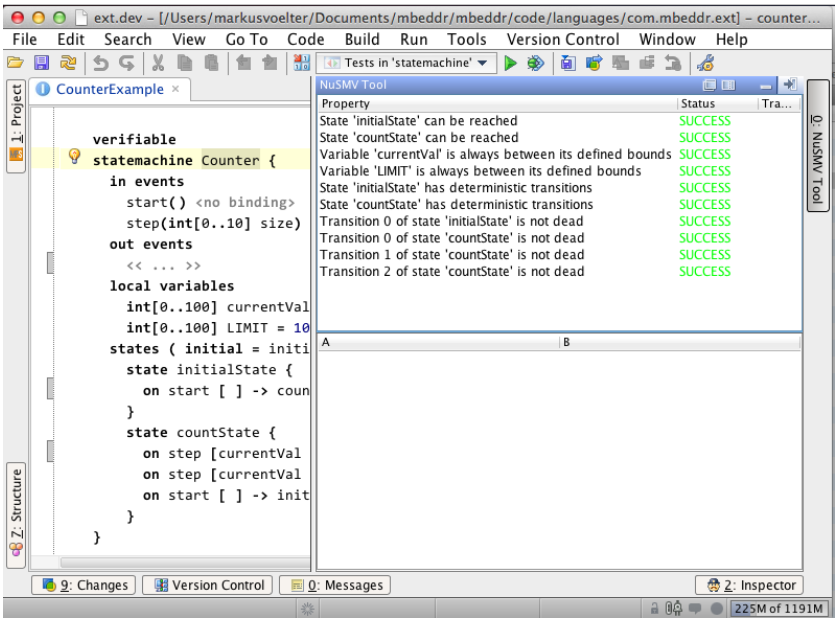
```
verified statemachine Counter {
  in events
    start()
    step(int[0..10] size)
  local variables
    int[0..100] currentVal = 0
    int[0..100] LIMIT = 10
  states ( initial = initialState )
    state initialState {
      on start [ ] -> countState {  }
    }
    state countState {
      on step [currentVal + size > LIMIT] -> initialState { }
      on step [currentVal + size <= LIMIT] -> countState { currentVal = currentVal + size; }
      on start [ ] -> initialState {  }
    }
}
```

Since this state machine is marked as `verifiable`, we can run the model checker from the context menu. Fig. 8.3 shows the result of running the model checker.
Here is a subset of the properties it has checked successfully (it performs these checks for all states/transitions by default):

```
State 'initialState' can be reached                          SUCCESS
Variable 'currentVal' is always between its defined bounds   SUCCESS
State 'countState' has deterministic transitions             SUCCESS
Transition 0 of state 'initialState' is not dead             SUCCESS
```

The first one reports that NuSMV has successfully proven that the `initialState` can be reached somehow. The second one reports that the variable `currentVal` stays within its bounds (notice how `currentVal` is a bounded integer). Line three reports that `countState` it never happens that in `countState`, more than one transition is ready to fire at any time. Finally, it reports that the first of the transitions is actually used at some point.

Let us provoke an error. We change the two transitions in `countState` to




```
on step [currentVal + size >= LIMIT] -> initialState {  }
on step [currentVal + size <= LIMIT] -> countState { currentVal = currentVal + size; }
```

You perhaps don't even recognize the difference — which is exactly why you would want to use a model checker! We have changed the > to a >= in the first transition. Running the model checker again, we get, among others,

```
State 'countState' contains nondeterministic transitions     FAIL    4
```

This means that there is a case where the two transitions are non-deterministic, i.e. both are possible based on the guard, and it not clear which one should be used. The 4 at the end means that the execution trace to this problem contains four steps. Clicking on the failed property check reveals the problematic execution trace:

```
State initialState
  LIMIT                        10
  currentVal                    0
State initialState
  in\_event: start start()
  LIMIT                        10
  currentVal                    0
State countState
  in\_event: step step(10)
```

```
  LIMIT                    10
  currentVal                0
State initialState
  LIMIT                    10
  currentVal               10
```

This is one (of potentially many) execution traces of this state machine that leads to the non-determinism: `currentVal` is 10, and because of the >=, both transitions could fire.

In addition to these default properties, it is also possible to specify custom properties. Instead of requiring users to be able to use the LTL or CTL specification languages and formulating the property on the level of the generated NuSMV input code (which is *much* longer than the state machine itself!), users can use convenient patterns, where they just have to fill in values. Here are two examples:

```
verification conditions
  never LIMIT != 10
  always eventually reachable initialState
```

The first one allows to express that we want the model checker to prove that any Boolean condition will never happen. In our example, we check that the `LIMIT` really is a constant and is never (accidentally) changed. The second one specifies that wherever we are in the execution of the state machine, it is still possible (after an arbitrary number of steps) to reach the `initialState`. Both properties hold for the example state machine.

■ *SAT Solving* SAT solving, which is short for satisfiability solving, concerns the satisfiability of sets of Boolean equations. Users specify a set of Boolean equations and the solver tries to assign truth values to the free variables so as to satisfy all specified equations[18]. SAT solving is an NP-complete problem, so there is no analytic approach: exhaustive search (implemented, of course, in much more clever ways) is the way to address these problems.

[18] There are also solvers that work with arithmetic expressions in addition to Booleans.

For example, SAT solving can be used to address the following problem. In the mbeddr C language, we support decision tables. A decision table has a set of Boolean conditions as row headers as well as a set of Boolean conditions in the column headers, as well as arbitrary values in the content cells. Fig. 8.5 shows an example.
SAT solving can be used to check whether all cases are handled. It can detect if combinations of the relevant variables exist for which no combination of row header and column header expressions match; in this case, the decision table would not return any value.

SAT solvers have some of the same challenges as model checkers regarding scalability, a low level and limited input language and the

```
module DecisionTableExample from cdesignpaper.gswitch imports nothing {

  enum mode { MANUAL; AUTO; FAIL; }

  mode nextMode(mode mode, int8_t speed) {
    return mode, FAIL
```

	mode == MANUAL	mode == AUTO
speed < 30	MANUAL	AUTO
speed > 30	MANUAL	MANUAL

```
  } nextMode (function)

}
```

Figure 8.5: An example decision table in mbeddr C. SAT solving is used to check it for consistency and completeness.

challenge of interpreting and understanding the output of a solver. Hence we use the same approach to solve the problem: from higher level models (such as the decision table) we generate the input to the solver, run it, and then report the result in the context of the high-level language.

■ *Model Checking and Transformations* A problem with model verification approaches in general is that they verify only the model. They can detect inconsistencies or property violations as a consequence of flaws in the program expressed with a DSL. However, even if we find no flaws in the model on DSL level, the *generator* may still introduce problems. In other words, the behavior of the generated code may be different from the (proven correct) behavior expressed in the model. There are three ways to address this:

- You can test your generator manually using the strategies suggested in this chapter. Once you trust the generator based one a sufficiently large set of tests, you then only have to verify the models — since you know they will be translated correctly.
- Some tools, for example the UPAAL model checker[19], can also generate test cases. These are stimuli to the model, together with the expected reactions. You can generate those into your target language and then run them in your target language. This is essentially an automated version of the first approach.
- Finally, you can verify the generated code. For example, there are model checkers for C. You can then verify that the properties that hold on the DSL level also hold on the level of the generated code. This approach runs into scalability issues relatively quickly, since the state space of a C program is much larger than the state space of a well-crafted state machine[20]. However, you can use this approach to verify the generated code based on a sufficient set of relatively

[19] http://www.uppaal.com/

[20] Remember that we use formalisms such as state machines instead of low level code specifically to allow more meaningful validation.

small test cases, making sure that these cover all aspects of the generator. Once you've built trust in the generator in this way, you can resort to verifying just the DSL models (which scales better).

8.4 Testing Editor Services

Testing IDE services such as code completion (beyond scopes), quick fixes, refactorings or outline structure has some of the challenges of UI testing in general. There are three ways of approaching this:

- The language workbench may provide specific APIs to hook into UI aspects and script tests for those. For example, MPS supports the scripting of user interactions in the editor, and hence check, if the editor behaves correctly (see Fig. 8.6).
- You can use generic UI testing tools to simulate typing and clicking in the editor, and checking the resulting behavior.
- Finally, you can islolate the algorithmic aspects of the IDE behavior (e.g. in refactorings or quick fixes) into separate modules (classes) and then unit test those with the techniques discussed in the rest of this chapter, independent of the actual UI.

In practice, I try to use the third alternative as far as possible: for non-trivial IDE functionality in quick fixes and refactorings, I isolate the behavior and write unit tests. For simple things I don't do any automated tests. For the actual UI, I typically don't do any automated tests at all, for three reasons. (1) it is simply too cumbersome and not worth the trouble, (2) as we use the editor to try things out, we implicitly test the UI, and (3), language workbenches are frameworks where, if you get the functionality right (via unit tests), they provide generic UIs that can be expected to work.

8.5 Testing for language appropriateness

A DSL is only useful if it can express what it is supposed to express. A bit more formally, one can say that the coverage of the DSL relative to the target domain should be 100%. In practice, this questions is much more faceted, though:

- Do we actually understand completely the domain the DSL is intended to cover?

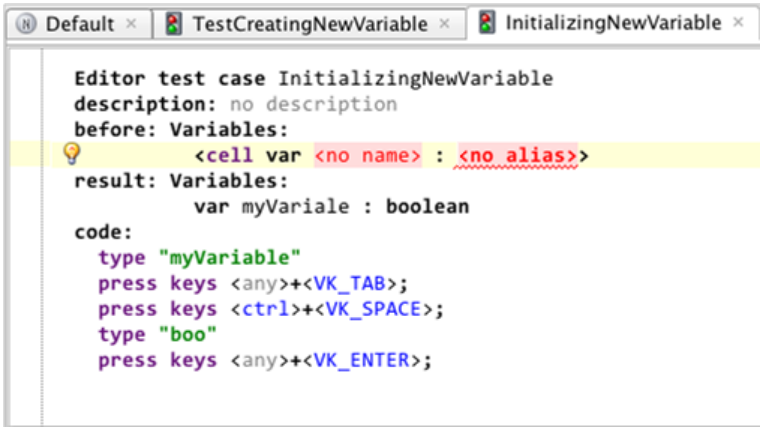


Figure 8.6: This test tests whether code completion works correctly. We start with an "empty" variable declaration in the `before` slot. It is marked with `cell`, an special annotation used in UI tests to mark the editor cell for that has the focus for the subsequent scripted behavior. In the `result` slot, we describe the state of the editor *after* the script code has been executed. The script code then simulates typing the word `myVariable`, pressing `TAB`, pressing `CTRL-SPACE`, typing `boo` (as a prefix of `boolean`) and pressing `ENTER`.

- Can the DSL cover this domain completely? What does "completely" even mean? Is it ok to have parts of the system written in L_{D-1} or do we have to express everything with the DSL?
- Even if the DSL covers the domain completely: are the abstractions chosen appropriate for the model purpose?
- Do the users of the DSL like the notation?

Answering these questions is impossible to automate. Manual reviews and validation relative to the (explicit or tacit) requirements for the DSL have to be performed. Getting these aspects right is the main reason why DSLs should be developed incrementally and iteratively.

9
Debugging DSLs

Debugging is relevant in two ways in the context of DSLs and language workbenches. First, the DSL developer may want to debug the definition of a DSL, including constraints, scopes or transformations and interpreters. Second, programs written in the DSL may have to be debuggable by the end user. We address both aspects in this chapter, illustrated with examples from the usual tools: MPS, Spoofax and Xtext.

9.1 Debugging the DSL Definition

Debugging the definition of the DSL boils down to a language workbench providing a debugger for the languages used or language definition. In the section we look at understanding and debugging the structure and concrete syntax, the definition of scopes, constraints and type systems as well as debugging interpreters and transformations.

9.1.1 Understanding and Debugging the Language Structure

In parser-based systems, the transformation from text to the AST performed by the parser[1] is itself a non-trivial process and has a potential for errors. Debugging the parsing process can be important.

[1] ... and the lexer, if there is one ...

■ *Xtext* Xtext uses ANTLR[2] under the hood. In other words, from the Xtext grammar, an ANTLR grammar is generated which performs the actual parsing. So understanding and debugging the Xtext parsing process means understanding and debugging the ANTLR parsing process.

[2] http://antlr.org

There are two ways to do this. First, since ANTLR generates a Java-based parser, you can debug the execution of ANTLR (as part of Xtext)

itself. Second, you can have Xtext generate a debug grammar, which contains no action code (so it does not populate the AST). However, it can be used to debug the parsing process with ANTLRWorks[3]. ANTLRWorks comes with an interactive debugger for ANTLR grammars.

[3] http://www.antlr.org/works

■ *MPS* In MPS there is no transformation from text to the AST since it is a projectional editor. However, there are still means to help better understand the structure of an existing program. For example, any program element can be inspected in the *Explorer*. Fig. 9.1 shows the information for a trivial C function:

```
int8_t add(int8_t x, int8_t y) {
  return x + y;
}
```

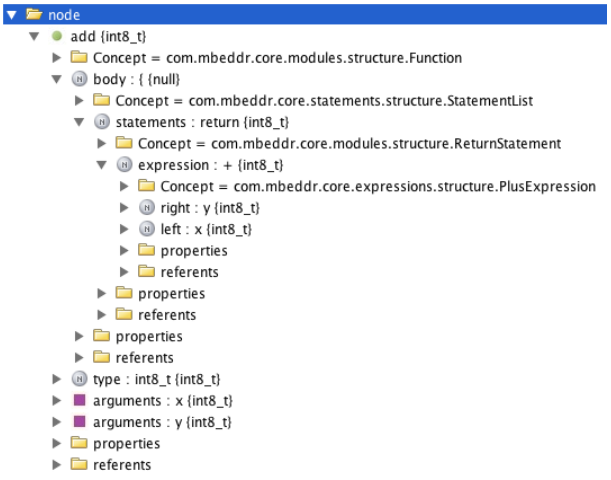


Figure 9.1: The MPS explorer shows the structure of a program as a tree. The explorer also shows the concept for each program element as well as the type, if an element has one.

MPS provides similar support for understanding the projection rules. For any program node MPS can show the cell structure as a tree. The tree contains detailed information about the cell hierarchy, the program element associated with each cell as well as the properties of the cell (height, width, etc.).

9.1.2 *Debugging Scopes, Constraints and Type Systems*

Depending on the level of sophistication of a particular language, a lot of non-trivial behavior can be contained in the code that determines scopes, checks constraints or computes types. In fact, in many languages, these are the most sophisticated aspects of language definition. Consequently, there is a need for debugging those.

■ *Xtext* In Xtext, all aspects of a language except the grammar

and the abstract syntax are defined via Java programs using Xtext APIs. This includes scopes, constraints and type system rules[4]. Consequently, all these aspects can be debugged by using a Java debugger. To do this, you can simply launch the Eclipse Application that contains the language and editor in debug mode and set breakpoints at the relevant locations[5].

[4] It also includes all IDE aspects.

[5] It is easy to criticize Xtext for the fact that it does not use DSLs for defining DSLs. However, in the context of debugging this is good, because no special debuggers are necessary.

■ *MPS* MPS comes with an equivalent facility in the sense that a second instance of MPS can be launched that runs "inside" the current one. This inner instance can be debugged from the outside one. This approach can be used for all those aspects of MPS-defined languages that are defined in terms of the BaseLanguage, MPS' version of Java. For example, scopes can be debugged this way. For example, in Fig. 9.2 we debug the scope for a `LocalVariableRef`.

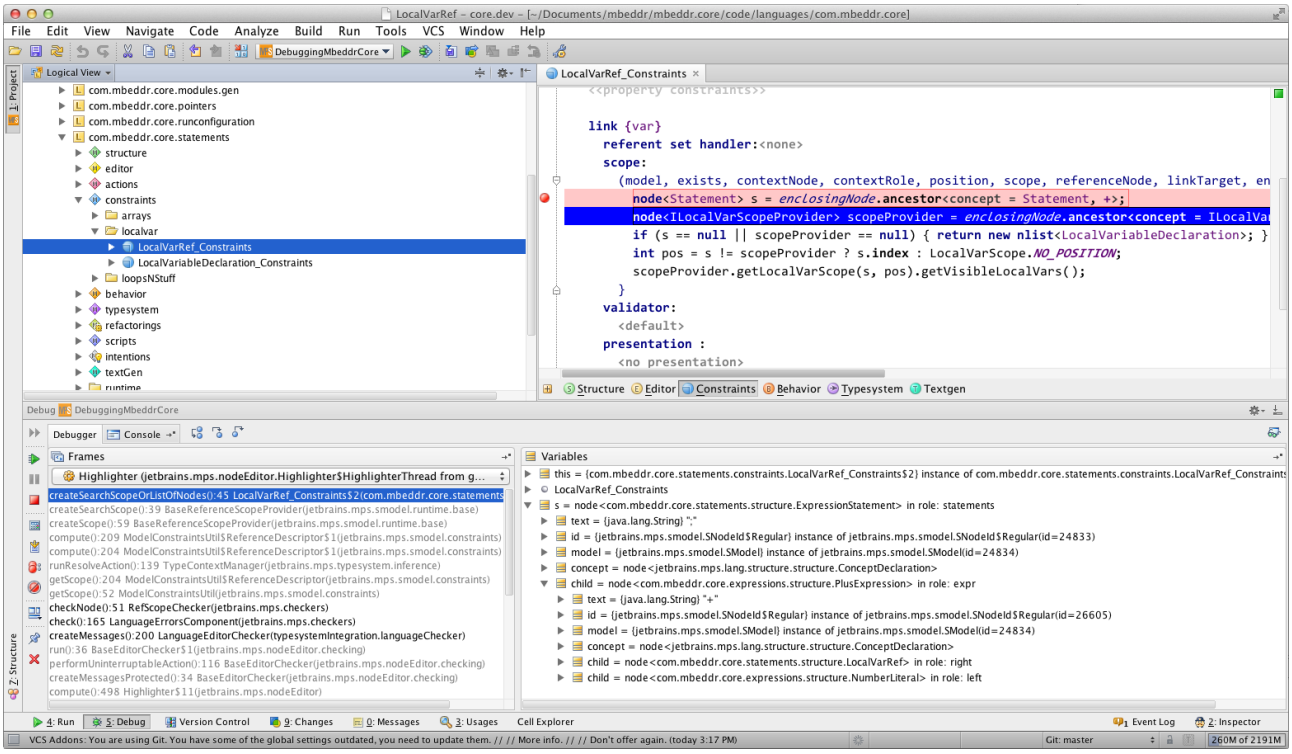


Figure 9.2: The debugger that can debug MPS while it "executes" a language is aware of all the relevant extensions to BaseLanguage. For example, in this screenshot we debug a scope constraint. Notice how in the `Variables` view program nodes (such as the `Statement s`) are shown on the abstraction level of the node, not in terms of its underlying Java data structure representation.

A related feature of MPS is the ability to analyze exception stack traces. MPS generates Java code from language definitions and then executes this Java code. If an exception occurs in language implementation code, this exception produces a Java stack trace. This stack trace can be pasted into a dialog in MPS. MPS then produces a version of the stack trace where the code locations in the stack trace (which are relative to the generated Java) have been translated to locations in the

DSL definition (expressed in Base Language). The locations can be clicked directly, opening the MPS editor at the respective location.

In the design chapter we discussed how declarative languages may come with a debugger that fits the particular declarative paradigm used by a particular declarative language. MPS comes with two facilities. First, pressing `Ctrl-Shift-T` on any program element will open a dialog that shows the type of the element. If the element has a type system error, that dialog also lets the user navigate to the rule that reported the error. The second facility is much more sophisticated. For any program node, MPS can show the so-called type system trace (Fig. 9.3 shows a simple example). Remember how the MPS type system is relies on a solver to solve the type system equations associated with program elements (specified by the language developer for the respective concepts). So each program has an associated set of type system equations. Those contain explicitly specified types as well as type variables. The solver tries to find type values for these variables such that all type system equations become true. The type system trace essentially visualizes the state of the solver including the values it assigns to type variables as well as which type system rules are applied to which program element.

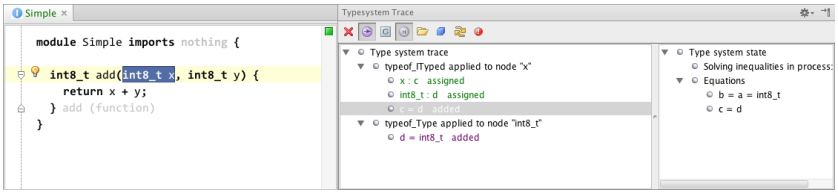


Figure 9.3: This example shows the solver state for the `Argument x`. It first applies the rule `typeof_ITyped` (`Argument` implements `ITyped`) which expresses that the type of the element (type variable `c` is the same as the element's `type` property (type variable `d`). It then applies the `typeof_Type` rule to the argument's type itself. This rule expresses that the type of a `Type` is a clone of itself. Consequently, the type variable `d` can be set to `int8_t`. In consequence this means that the type variable `c` (which represents the type of the `Argument`) is also `int8_t`. Note that this is a trivial example. Type system traces can become quite involved.

9.1.3 *Debugging Interpreters and Transformations*

Debugging an interpreter is simple: since an interpreter is just a program written in some programming language that processes and acts on the DSL program, debugging the interpreter simply uses the debugger for the language in which the interpreter is written[6].

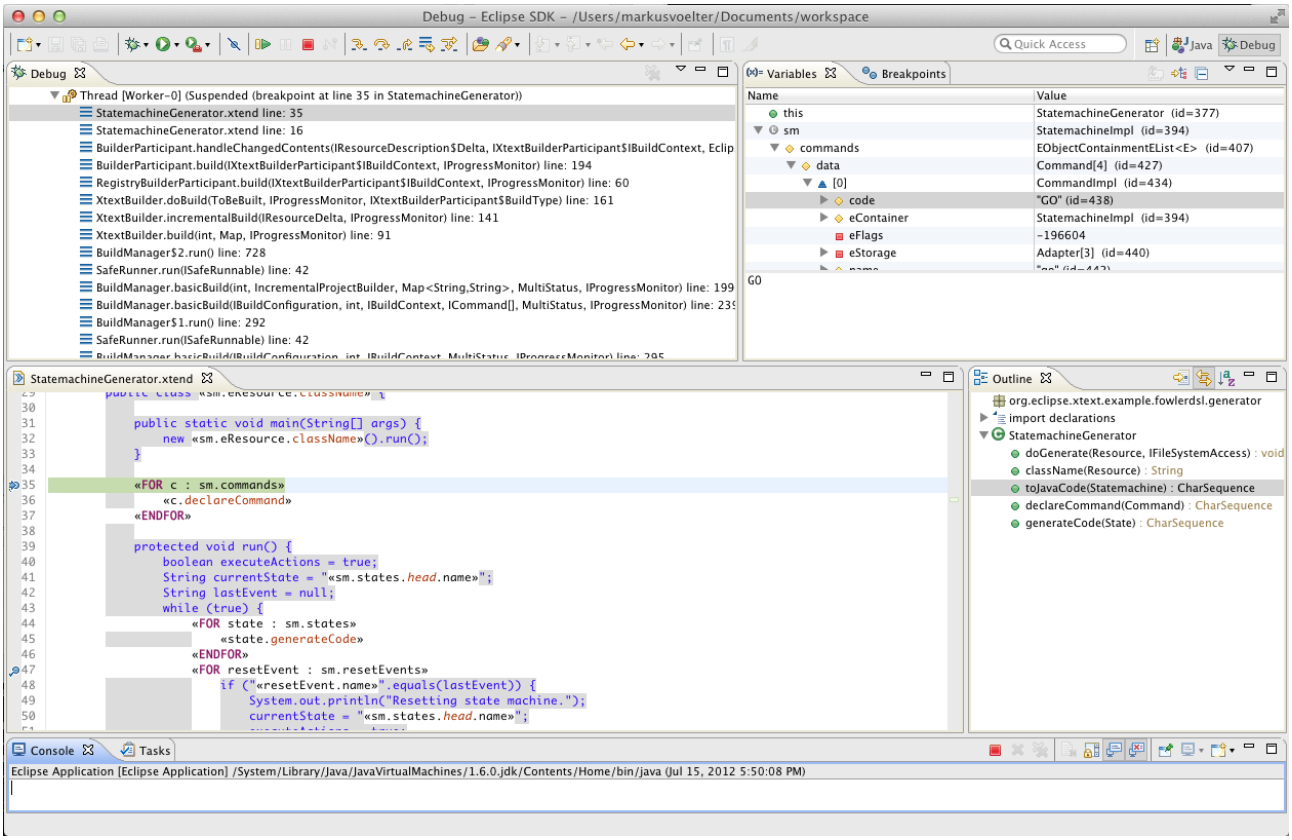
Debugging transformations and generators is typically not quite as trivial, for two reasons. First, transformations and generators are typically written in DSLs optimized for this task. So a specialized debugger is required. Second, if multi-step transformations are used, the intermediate models may have to be accessible, and it should be possible to trace a particular element through the multi-step transformation.

■ *Xtext* As we have said before, Xtext can be used together with any EMF-compatible code generator or transformation engine. However, since Xtext ships with Xtend, we look at debugging Xtend trans-

[6] This assumes that the interpreter is written in a programming language for which a debugger exists

formations. Model-to-model transformations and code generators in Xtend look very similar: both use Xtend to navigate over and query the model, based on the AST. The difference is that, as a side effect, model-to-model transformations create new model elements and code generators create strings — typically using the rich strings (aka template expressions).

Xtend supports debugging either on the level of the generated code, or on the Xtend source level. As Fig. 9.1.3 shows, even the template expressions can be debugged. The Variables view shows the EMF representation (i.e. the implementation) of program elements.



Xtend is a fundamentally an imperative/OO language, so the step-through metaphor for debuggers is appropriate. If Xtend is used for code generation or transformation, debugging a boils down to stepping through the code that builds the target model. I emphasize this because the next example uses a different appraoch.

■ *MPS* In MPS, working with several chained transformations is normal, so MPS provides support for debugging the transformation

process. This support includes two ingredients. The first one is showing the mapping partitioning. For a given model, MPS computes the order in which transformations apply and reports it to the user. This is useful to understand which transformations are executed in which order[7]. Let us investigate a simple example C program that contains a message definition and a `report` statement. The `report` statement is transformed to `printf` statements.

[7] ... and to debug transformation priorities!

```
module Simple imports nothing {

  message list messages {
    INFO aMessage() active: something happened
  }

  exported int32_t main(int32_t argc, int8_t*[] argv) {
    report(0) messages.aMessage() on/if;
    return 0;
  }
}
```

Below is the mapping configuration for this program:

```
[ 1 ]
com.mbeddr.core.modules.gen.generator.template.main.removeCommentedCode
[ 2 ]
com.mbeddr.core.util.generator.template.main.reportingPrintf
[ 3 ]
com.mbeddr.core.buildconfig.generator.template.main.desktop
com.mbeddr.core.modules.gen.generator.template.main.main
```

This particular model is generated in three phases. The first one removes commented code to make sure it does not show up in the resulting C text file. The second phase runs the generator that transforms `report` statements into `printf`s. Finally, the `desktop` generator generates a `make` file from the build configuration, and the last step generates the C text from the C tree.

By default, MPS runs all generators until everything is either discarded or transformed into text. While intermediate models exist, they are not shown to the user. For debugging purposes though, these intermediate, transient models can be kept around for inspection. Each of the phases is represented by one or more transient models. As an example, here is the program after the `report` statement has been transformed:

```
module Simple imports nothing {

  exported int32_t main(int32_t argc, int8_t*[] argv) {
    printf("$$ aMessage: something happened ");
    printf("@ Simple:main:0#240337946125104144 \n ");
    return 0;
  }
}
```

MPS also supports tracing an element through the intermediate models. Fig. 9.4 shows an example. Users can select a program element in the source, target or an intermediate model and trace it to the respective other ends of the transformation.

Figure 9.4: The generation trace functionality in MPS allows users to trace how a particular program element is transformed through a chain of transformations. The generation tracer also shows the transformation rules involved in the transformation.

Note how this approach to debugging transformations is very different from the Xtend example above: instead of stepping through the transformation code[8], MPS provides a *static* representation of the transformation in terms of the intermediate models and the element traces through them.

[8] As part of the general Debug-MPS-in-MPS functionality, MPS transformations can also be debugged in a more imperative fashion. This is useful, for example, to debug more complex logic used inside transformation templates.

9.2 *Debugging DSL Programs*

To find errors in DSL programs, we can either debug them on the level of the DSL program or in its L_{D-1} representation (i.e. in the generated code or the interpreter). Debugging on L_{D-1} is useful if you want to find problems in the execution engine, or, to some extent, if the language users are programmers and they have an intimate understanding of the L_{D-1} representation of the program. However, for many DSLs it is necessary to debug on the level of the DSL program either because the users are not familiar with the L_{D-1} representation[9], or because the L_{D-1} is so low-level and complex that is bears no obvious resemblance to the DSL program.

[9] This is true particularly for DSLs qtargeted at domain experts,

The way to build debuggers for DSLs of course depends on the DSL itself. For example, for DSLs that only describe structures, debugging does not make much sense in the first place. For DSLs that describe behavior, the debugging approach depends on the behavioral paradigm

used in the DSL. We have discussed this in . In this section we focus mostly on the imperative paradigm[10].

Building a debugger poses two challenges. The first one is the debugger UI: creating all the buttons and views for controlling the debugger and for showing variables and treads. The second challenge concerns the control of and data exchange with the to-be-debugged program. The first challenge is relatively simple to solve, since many IDE frameworks (including Eclipse and MPS) already come with debugger frameworks.

The second challenge can be a bit more tricky. If the DSL is executed by an interpreter, the situation is simple: the interpreter can be run and controlled directly from the debugger. It is easy to implement single-stepping and variable watches, for example, since the interpreter can directly provide the respective interfaces[11]. On the other hand, if the DSL program is transformed into code that is executed in some other environment outside of our control, it may even be impossible to build a debugger because there is no way to influence and inspect the running program. Alternatively, it may be necessary to build a variant of the code generator which generates a *debug version* of the program which contains specialized code to interact with the debugger. For example, values of variables may be stored in a special data structure inspectable by the debugger, and at each program location where the program may have to stop (in single step mode or as a consequence of a breakpoint) code is inserted that explicitly suspends the execution of the program, for example by `sleep`ing the current thread. However, such an approach is often limited and ugly — in the end, an execution infrastructure must provide debug support to enable robust debugging.

[10] An examples for the functional paradigm has been provided in , and the type system tracer described above is an example of a debugger for a declarative language.

[11] This is especially true if the interpreter is written in the same language as the IDE — no language integration issues have to be addressed in this case.

9.2.1 *Print Statements — a Poor Man's Debugger*

As the above discussion suggests, building full-blown debuggers may be a lot of work. It is worth exploring whether a simpler approach is good enough. The simplest such approach is to extend the DSL with language concepts that simply print interesting aspects of the executing program to the console or a log file. For example, the values of variables may be output this way.

The mbeddr `report` statement is an example of this approach. A `report` statement takes a message text plus a set of variables. It then outputs the message and the values of these variables. The target of the report statement can be changed. By default, it reports to the console. However, since certain target devices may not have any console, alternative transformations can be defined for report statements, that, for example, could output the data to an error memory or a serial line. A particularly interesting feature of `report` statements is that the trans-

formation that handles them knows where in the program the `report` statement is located. It can add this information to the output[12].

[12] The go-to-error-location functionality discussed in Fig. 8.4 is based on this appraoch.

An approach based on print *statements* is sometimes clumsy because it requires factoring out the to-be-printed expression. And it only works for an imperative language in the first place. For languages that make use of sophisticated expressions, a different approach is recommended. Consider the following example:

```
Collection[Type] argTypes = aClass.operations.arguments.type;
```

If you wanted to print the list of operations and arguments, you would have to change the program to something like this:

```
print("operations: " + aClass.operations);
print("arguments: " + aClass.operations.arguments);
Collection[Type] argTypes = aClass.operations.arguments.type;
```

A much simpler alternative uses *inlined* reporting expressions:

```
Collection[Type] argTypes = aClass.operations.print("operations:")
                            .arguments.print("arguments:").type;
```

To make this convenient to use, the `print` function has to return the object it is called on (the one before the dot), and it must be typed accordingly if a language with static type checking is used[13].

[13] The original openArchitectureWare Xtend did it this way.

9.2.2 *Automatic Program Tracing*

As languages and programs become more complex, an automated tracing of program execution may be useful. In this approach, all execution steps in a program are automatically traced and logged into a tree-like data structure. The refrigerator cooling language uses this approach. Here is an example program:

```
cooling program HelloWorld {
    var temp: int
    start:
        entry { state s1 }
    state s1:
        check temp < 10 { state s2 }
    state s2:
}
```

Upon startup, it enters the `start` state and immediately transitions to state `s1`. It remains in `s1` until the variable `temp` becomes less than 10. It then transitions to `s2`. Below is a test for this program that verifies this behavior:

```
test HelloWorldTest for HelloWorld {
    prolog {
        set temp = 30
    }
    step
    assert-currentstate-is s1
    step
    mock: set temp = 5
    step
    assert-currentstate-is s2
}
```

Fig. 9.5 shows the execution trace. It shows the execution of each statement and the evaluation of each expression. The log viewer is a table tree so the various execution steps can be selectively expanded and collapsed. To connect to the program, users can double-click on an entry to select the respective program element in the source node. By adding special comments to the source, the log can be structured further.

Interpreter Execution Log — Rerun

Time	Kind	Element	Message
▼16:48:07.451	info	TestProlog	executing prolog
▼16:48:07.451	debug	AssignmentStatement	executing AssignmentStatement
▼16:48:07.451	eval	NumberLiteral	evaluating NumberLiteral result: 30
16:48:07.451	debug	NumberLiteral	value is a char, 30
16:48:07.451	debug	AssignmentStatement	setting temp to 30
▼16:48:07.451	info	CoolingProgram HelloWorld	initializing Program
▼16:48:07.451	info	StartState start	executing startState
16:48:07.451	info	StartState start	entering state start
▼16:48:07.451	debug	ChangeStateStatement	executing ChangeStateStatement
16:48:07.452	debug	ChangeStateStatement	change state to s1
16:48:07.452	info	StartState start	leaving state start
16:48:07.452	info	CustomState s1	entering state s1
▼16:48:07.452	info	CoolingTest HelloWorld...	running test code
▼16:48:07.452	info	CoolingProgram HelloWorld	step 0
▼16:48:07.452	eval	Smaller	evaluating Smaller result: false
▼16:48:07.452	eval	SymbolRef	evaluating SymbolRef result: 30
16:48:07.452	debug	SymbolRef	looking up value: 30
▼16:48:07.452	eval	NumberLiteral	evaluating NumberLiteral result: 10
16:48:07.452	debug	NumberLiteral	value is a char, 10
16:48:07.452	debug	TestAssertState	executing TestAssertState
▼16:48:07.452	info	CoolingProgram HelloWorld	step 1
▼16:48:07.452	eval	Smaller	evaluating Smaller result: false
▼16:48:07.452	eval	SymbolRef	evaluating SymbolRef result: 30
16:48:07.452	debug	SymbolRef	looking up value: 30
▼16:48:07.452	eval	NumberLiteral	evaluating NumberLiteral result: 10
16:48:07.452	debug	NumberLiteral	value is a char, 10
▼16:48:07.452	debug	TestMockSingleLine	executing TestMockSingleLine
▼16:48:07.452	debug	AssignmentStatement	executing AssignmentStatement
▼16:48:07.452	eval	NumberLiteral	evaluating NumberLiteral result: 5
16:48:07.452	debug	NumberLiteral	value is a char, 5
16:48:07.452	debug	AssignmentStatement	setting temp to 5
▼16:48:07.452	info	CoolingProgram HelloWorld	step 2
▼16:48:07.452	eval	Smaller	evaluating Smaller result: true
▼16:48:07.452	eval	SymbolRef	evaluating SymbolRef result: 5
16:48:07.452	debug	SymbolRef	looking up value: 5
▼16:48:07.452	eval	NumberLiteral	evaluating NumberLiteral result: 10
16:48:07.452	debug	NumberLiteral	value is a char, 10
▼16:48:07.452	debug	ChangeStateStatement	executing ChangeStateStatement
16:48:07.452	debug	ChangeStateStatement	change state to s2
16:48:07.452	info	CustomState s1	leaving state s1
16:48:07.452	info	CustomState s2	entering state s2
16:48:07.452	debug	TestAssertState	executing TestAssertState
16:48:07.452	success	CoolingTest HelloWorld...	test successful.

Figure 9.5: The log viewer represents a program's execution as a tree. The first column contains a timestamp and the tree nesting structure. The second column contains the kind (severity) of the log message. A filter can be used to show only messages above a certain severity. The third column shows the language concept with which the trace step is associated (double-clicking on a row selects this element in the editor). Finally, the last column contains the information about the semantic action that was performed in the respective step.

The execution engine for the programs is an interpreter, which makes it particularly simple to collect the trace data. All interpreter methods that execute statements or evaluate expressions take a `LogEntry` object as an additional argument. The methods then add children to the current `LogEntry` that describe whatever the method did, and then passes the child to any other interpreter methods it calls. As an example, here is the implementation of the `AssignmentStatement`:

```
protected void executeAssignmentStatement(AssignmentStatement s, LogEntry log) {
    LogEntry c = log.child(Kind.info, context, "executing AssignmentStatement" );
    Object l = s.getLeft();
    Object r = eval(s.getRight(), c);
    eec().environment.put(symbol, r);
    c.child(Kind.debug, context, "setting " + symbol.getName() + " to " + r);
}
```

If instead of an interpreter a code generator were used, the same approach could essentially be used. Instead of embedding the tracing code in the interpreter, the code generator would generate code that would build the respective trace data structure in the executing program. Upon termination, the data structure could be dumped to an XML file and subsequently loaded by the IDE for inspection.

9.2.3 *Simulation as an Approximation for Debugging*

The interpreter for the Cooling programs mentioned above is of course not the final execution engine — C code is generated that is executed on the actual target refrigerator hardware. However, as we have discussed in the design part of the book (), we can make sure the generated code and the interpreter are semantically identical by running a sufficient (large) number of tests. If we do this, we can use the interpreter to test the programs for logical errors. The interpreter can also be used interactively, in which case it acts as a simulator for the executing program. Fig. **??** shows a screenshot of a running program.

If you look at Fig. **??** closely, you will see that it shows all variables in the program, the events in the queue, running tasks as well as the values of properties of hardware elements and the current state. It also provides a button to single-step the program, to run it continuously, and until it hits a breakpoint. In other words, although the simulator does not use the familiar[14] UI of a debugger, it actually is a debugger[15]!

If you already have the interpreter[16], expanding it into a simulator/debugger is relatively simple. Essentially only three things have to be done:

- First, the execution of the program must be controllable from the outside. This involves setting breakpoints, single-stepping through

[14] ... familiar to programmers, but not to the target audience!

[15] As we have said above, the fact that it runs in the interpreter instead of the generated code is not a problem if we ensure the two are semantically identical. Of course we cannot find bugs in the implementation (i.e. in the generator) this way. But to detect those, debugging on the level of the generated C code is more useful anyway.

[16] we discuss how to build on in

the program and stopping execution if a breakpoint is hit. In our example case, we do not single-step through statements, but only through steps[17]. Breakpoints are essentially Boolean flags associated with program elements: if the execution processes a statement that has the `breakpoint` flat set to `true`, execution stops.

- Second, we have implemented an Observer infrastructure for all parts of the program state that should be represented in the simulator UI. Whenever one of them changes (as a side effect of executing a statement in the program), an event is fired. The UI registers as an observer and updates the UI in accordance with the event.
- Third, values from the program state must be changeable from the outside. As a value in the UI (such as the temperature of a cooling compartment) is changed by the user, the value is updated in the state of the interpreter as well.

[17] Remember that the language is time-triggered anyway, so execution is basically a sequence of steps triggered by a timer. In the simulator/debugger, the timer is replaced with the user pressing the `Next Step` button for single stepping.

9.2.4 *Automatic Debugging for Xbase-based DSLs*

DSLs that use Xbase, the re[illegible]e expression language that ships with Xtext, get debugging mostly for free. This is because of the tight integration of Xbase with the JVM. While we describe this integration in more detail in . Here is the essence.

A DSL that uses Xbase typically defines its own structural and high-level behavioral aspects but uses Xbase for the fine-grained, expression-level behavior. For example, in a state machine DSL, states, events and transitions would be concepts defined by the DSL, but the guard conditions and the action code would reuse Xbase expressions. When mapping this DSL to Java[18], the following approach is used. The structural and high-level behavioral aspects are mapped to Java, but [illegible]y generating Java text, but by mapping the DSL AST to a Java AST. For the reused Xbase aspects (the finer-grained behavior) there already exists a to-Java generator (called the Xbase compiler) which we simply call from our generator.

[18] Xbase-based DSLs *must* be mapped to Java to benefit from Xbase in the way discussed in this section. If you generate code other than Java, Xbase cannot be used sensibly.

Essentially, we do not create a code generator, but a model-to-model transformation from the DSL AST to the Java AST. As part of the this transformation (performed by the so-called JVMModel inferrer), trace links between the DSL code and the created Java code are created. In other words, the relationship between the Java code and the DSL code is well known. This relationship is exploited in the debugging process.

Xbase-based DSLs use the Java debugger for debugging. In addition to showing the generated Java code, the debugger can also show the DSL code, based on the trace information collected by the JVMModel inferrer. In the same way, if a user sets a breakpoint in the DSL code, the trace information is used to determine where to set the breakpoint in the generated Java code.

9.2.5 *Debuggers for an Extensible Languages*

This section describes in some detail the architecture of an extensible debugger for an extensible language[19]. We illustrate the approach with an implementation based on mbeddr, an extensible version of C implemented with the MPS. We also show the debuggers for non-trivial extensions of C.

[19] Extensible languages have been defined and discussed in the design section . We show in detail in how this works with MPS

Requirements on the Debugger Debuggers for imperative languages support at least the following features: *Breakpoints* suspend execution on arbitrary statements; *Single-Step Execution* steps over statements, and into and out of functions or other callables; *Watches* show values of variables, arguments or other aspects of the program state. *Stack Frames* visualize the call hierarchy of functions or other callables.

When debugging a program that contains extensions, breakpoints, stepping, watches and call stacks *at the extension-level* differ from their counterparts *at the base-level*. The debugger has to perform the mapping from the base-level to the extension-level (Fig. 9.6). We distinguish between the *tree* representation of a program in MPS and the generated *text* that is used by the C compiler and the debugger backend. A program in the tree representation can he separated into parts expressed in the *base* language and parts expressed using extensions. We refer to the latter the *extension*-level or *DSL*-level (see Fig. 9.6).
An extensible *tree*-level debugger for mbeddr that supports debugging on the *base*-level and *extension*-level, addresses the following requirements:

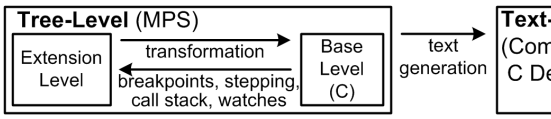


Figure 9.6: An extension-aware debugger maps the debug behavior from the base-level to the extension-level (an extension may also be mapped onto other extensions; we ignore this aspect in this section).

Modularity Language extensions in mbeddr are modular, so debugger extensions must be modular as well. No changes to the base language must be necessary in order to enable debugging for a language extension.

Framework Genericity In addition, new language extensions must not require changes *to the core debugger infrastructure* (not just the base language).

Simple Debugger Definition Creating language extensions is an integral part of using mbeddr. Hence, the development of a debugger for an extension should be simple and not require too much knowledge about the inner workings of the framework, or even the C debugger backend.

Limited Overhead As a consequence of embedded software development, we have to limit the additional, debugger-specific code generated into the binary. This would increase the size of the binary, potentially making debugging on a small target device infeasible.

Debugger Backend Independence Embedded software projects use different C debuggers depending on the target device. This prevents modifying the C debugger itself: changes would have to be reimplemented for every C debugger used.

■ *An example Extension.* We start out by developing a simple extension to the mbeddr C language[20]. The `foreach` statement can be used to conveniently iterate over C arrays. Users have to specify the array as well as its size, since in C an array cannot be queried for its size. Inside the `foreach` body, `it` acts as a reference to the current iteration's array element.

[20] We assume that you have read the basics about MPS language development in .

```
int8_t s = 0;
int8_t[] a = {1, 2, 3};
foreach (a sized 3) {
    s += it;
}
```

The code generated from this piece of extended C looks as follows. The `foreach` statement is expanded into a regular `for` statement and an additional variable `__it`:

```
int8_t s = 0;
int8_t[] a = {1, 2, 3};
for (int __c = 0; __c < 3; __c++) {
    int8_t __it = a[__c];
    s += __it;
}
```

To make the `foreach` extension modular, it lives in a separate language module named `ForeachLanguage`. The new language extends C, since we will refer to concepts defined in C (see Fig. 9.7).

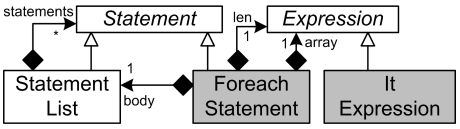


Figure 9.7: UML class diagram showing the structure of the `ForeachLanguage`. Concepts from the C base language are in white boxes, new concepts are grey.

■ *Developing the Language Extension.* In the new language, we define the `ForeachStatement`. To make it usable wherever C expects `Statements` (i.e. in functions), `ForeachStatement` extends C's `Statement`. `ForeachStatements` have three children (see Fig. 9.7): an `Expression` that represents the array, an `Expression` for the array length, and a `StatementList` for the body. `Expression` and `StatementList` are both defined in C.

The editor that defines the concrete syntax is shown in Fig. 9.8. It consists of a horizontal list of cells: the `foreach` keyword, the opening parenthesis, the embedded editor of the `array` child, the `sized` keyword, the embedded editor of the `len` expression, the closing parenthesis and the editor of the `body`.

As shown in the code snippet below, the `array` must be of type `ArrayType`, and the type of `len` must be `int64_t` or any of its shorter subtypes.

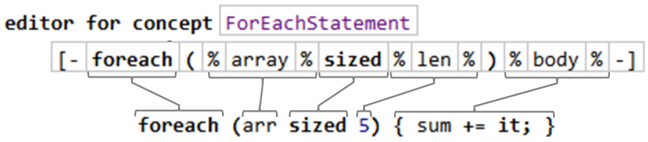


Figure 9.8: The editor definition of the `foreach` statement and its relationship to an example instance.

```
rule typeof_ForeachStatement for ForeachStatement as fes do {
    typeof(fes.len) :<=: <int64_t>; if
    (!(fes.array.type.isInstanceOf(ArrayType))) {
      error "array required" -> fes.array;
    }
}
```

The generator translates a `ForeachStatement` to a regular `for` statement that iterates over the elements with a counter variable `__c` (Fig. 9.9). Inside the `for` body, we create a variable `__it` that refers to the array element at position `__c`. We then copy in the other statements from the body of the `foreach`. Note that there is an existing generator for C that outputs C text which is subsequently compiled.

```
[concept    ForEachStatement
inheritors false
condition  <always>
 -->
content node:
void dummy() {
  int8_t[ ] x;
  <TF  for ($COPY_SRC$[int64_t] __c = 0; __c < $COPY_SRC$[10]; __c++) {  TF>
       $COPY_SRC$[int8_t] __it = $COPY_SRC$[x][__c];
       $COPY_SRCL$[int8_t x; ]
     }
}
```

Figure 9.9: The `foreach` generator template. A `ForeachStatement` is replaced by what is framed `<TF .. TF>` when the template is executed, the `dummy` function around it just provides context. The `COPY_SRC` and `COPY_SRCL` macros contain expressions (not shown) that determine with what the nodes in square brackets (e.g. `10`, `int8_t x;`) are replaced during a transformation.

The `ItExpression` extends C's `Expression` to make it usable where expressions are expected. The editor consists of a single cell with the keyword `it`. A constraint enforces that the `ItExpression` is only used inside the body of a `foreach`:

```
concepts constraints ItExpression {
  can be child
    (context, scope, parentNode, link, childConcept)->boolean {
      parentNode.ancestor<ForeachStatement, +>.isNotNull &&
      parentNode.ancestor<StatementList, +>.isNotNull;
    }
}
```

The type of `it` must be the base type of the `array` (e.g. `int` in case of `int[]`), as shown in the code below:

```
node<Type> basetype = typeof(it.ancestor<ForeachStatement>.array)
                        as ArrayType.baseType;
typeof(it) :==: basetype.copy;
```

The `foreach` generator already generated a local variable `__it` into the body of the `for` loop. We can thus translate an `ItExpression` into a `LocalVariableReference` that refers to `__it`.

The specification of the `foreach` debugger extension resides completely in the `ForeachLanguage`; this keeps the debugger definition for the extension local to the extenstion language.

To set a breakpoint on a concept, it must implement the `IBreakpoint-Support` marker interface. `Statement` already implements this interface, so `ForEachStatement` implicitly implements this interface as well.

Stepping behavior is implemented via `ISteppable`. `ForeachStatement` implements this interface indirectly via `Statement`, but we have to overwrite the methods that define the step over and step into behavior. Assume the debugger is suspended on a `foreach` and the user invokes *step over*. If the array is empty or we have finished iterating over it, a step over ends up on the statement that follows *after the whole* `foreach` statement. Otherwise we end up on the first line of the `foreach` body (`sum += it;`)[21].

[21] This is the first line of the mbeddr program, *not* the first line of the generated base program (which would be `int8_t __it = arr[__c];`).

The debugger cannot guess which alternative will occur since it would require to know the state of the program and to evaluate the expressions in the (generated) `for`. Instead we set breakpoints *on each of the possible next statements* and then resume execution until we hit one of them. The implementations of the `ISteppable` methods specify strategies for setting breakpoints on these possible next statements. The `contributeStepOverStrategies` method collects strategies for the *step over* case:

```
void contributeStepOverStrategies(list<IDebugStrategy> res) {
   ancestor
   statement list: this.body
}
```

The method is implemented using a domain-specific language for debugger specification, which is part of the mbeddr debugger framework[22]. It is an extension of MPS' `BaseLanguage`, a Java-based language used for expressing behavior in MPS. The `ancestor` statement delegates to the `foreach`'s ancestor; this will lead to a breakpoint on the subsequent statement. The second line leads to a breakpoint on the first statement of the `body` statement list.

[22] This simplifies the implementation of debuggers significantly. It is another example of where using DSLs for defining DSLs is a good idea.

Since the `array` and `len` expressions can be arbitrarily complex and may contain invocations of callables (such as function calls), we have to specify the *step into* behavior as well. This requires the debugger to

inspect the expression trees in `array` and `len` and find any expression that can be stepped into. Such expressions implement `IStepIntoable`. If so, the debugger has to step into each of those, in turn. Otherwise the debugger falls back to *step over*. An additional method configures the expression trees which the debugger must inspect:

```
void contributeStepIntoStrategies(list<IDebugStrategy> res) {
  subtree: this.array
  subtree: this.len
}
```

By default, the Watch window contains all C symbols (global and local variables, arguments) as supplied by the native C debugger. In case of the `foreach`, this means that `it` is not available, but `__it` and `__c` are. This is exactly the wrong way: the watch window should show `it`, but not `__it` and `__c`. To customize Watches, a concept has to implement `IWatchProvider`. Here is the code for `foreach`, also expressed in the debugger definition DSL:

```
void contributeWatchables(list<UnmappedVariable> unmapped,
                          list<IWatchable> mapped) {
  hide "__c"
  map "__it" to "it"
    type: this.array.type : ArrayType.baseType
    category: WatchableCategories.LOCAL_VARIABLES
    context: this    }
```

The first line hides `__c`. The rest maps a base-level C variable to a Watchable. It finds a C variable named `__it` (inserted by the `foreach` generator) and creates a watch variable named `it`. At the same time, it hides the base-level variable `__it`. The type of `it` is the base type of the array over which we iterate. We assign the `it` Watchable to the local variables section and associate the `foreach` node with it (double-clicking on the `it` in the Watches will highlight the `foreach` in the code).

Stepping into the `foreach` body does not affect the call stack, since the concept represents no callable (details see Section **??**). So we do not have to implement any stack frame related functionality.

■ *Debugger Framework Architecture.* The central idea of the debugger architecture is this: from the C code in MPS and its extensions (tree-level) we generate C text (text-level). This text is the basis for the debugging process by a native C debugger. We then use trace data to find out how the generated text maps back to the tree-level in MPS (R4).

At the core of the execution architecture is the `Mapper`. It is driven by the `Debugger UI` (and through it, the user) and controls the C debugger via the `Debug Wrapper`. It uses the `Program Structure` and

the Trace Data. The Mapper also uses a language's debug specification, discuss in the next subsection. Fig. 9.10 shows the components and their interfaces.

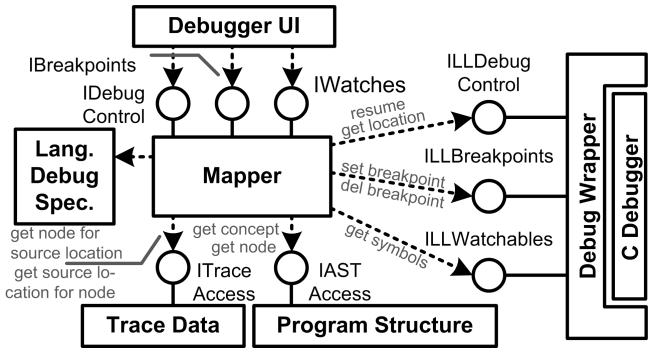


Figure 9.10: The Mapper is the central component of the debugger execution architecture. It is used by the Debugger UI and, in turn, uses the Debug Wrapper, the Program Structure and the Trace Data.

The IDebugControl interface is used by the Debugger UI to control the Mapper. For example, it provides a resume operation. IBreakpoints allows the UI to set breakpoints on program nodes. IWatches lets the UI retrieve the data items for the Watch window. The Debug Wrapper essentially provides the same interfaces, but on the level of C (prefixed with LL, for "low level"). In addition, ILLDebugControl lets the Mapper find out about the program location of the C Debugger when it is suspended at a breakpoint. IASTAccess lets the Mapper access program nodes. Finally, ITraceAccess lets the Mapper find out the program node (tree-level) that corresponds to a specific line in the generated C source text (text-level), and vice versa.

To illustrate the interactions of these components, we describe a *step over*. After the request has been handed over from the UI to the Mapper via IDebugControl, the Mapper performs the following steps:

- Ask the current node's concept for its *step over* strategies; these define all possible locations where the debugger could end up after the *step over*.
- Query TraceData for the corresponding lines in the generated C text for those program locations.
- Use the debugger's ILLBreakpoints to set breakpoints on those lines in the C text.
- Use ILLDebugControl to resume program execution. It will stop at any of the just created breakpoints.
- Use ILLDebugControl to get the C call stack.
- Query TraceData to find out for each C stack frame the corresponding nodes in the tree-level program.
- Collect all relevant IStackFrameContributors (see next section). The Mapper uses them to construct the tree-level call stack.

- Get the currently visible symbols and their values via `ILLWatchables`.
- Query the nodes for all `WatchableProviders` and use them to create a set of Watchables.

At this point, execution returns to the `Debugger UI`, which then gets the current location and Watchables from the `Mapper` to highlight the statement on which the debugger is suspended and populate the watch window.

In our implementation, the `Debugger UI`, `Program Repository` and `Trace Data` are provided by MPS. In particular, MPS builds a trace from the program nodes (tree-level) in MPS to the generated text-level source. The `Debug Wrapper` is part of mbeddr and relies on the Eclipse CDT Debug Bridge ([23]), which provides a Java API to `gdb` ([24]) and other C debuggers.

[23] www.eclipse.org/cdt/

[24] www.gnu.org/software/gdb/documentation/

■ *Debugger Specification.* The debugger specification reside in the respective language module. As we have seen in the `foreach` example, the specification relies on a set of interfaces, a number of predefined strategies as well as the debugger specification DSL.

The interface `IBreakpointSupport` is used to mark language concepts on which breakpoints can be set. C's `Statement` implements this interface. Since all statements inherit from `Statement` we can set breakpoints on all statements by default.

When the user sets a breakpoint on a program node, the mapper uses `ITraceAccess` to find the corresponding line in the generated C text. A statement defined by an extension may be expanded to several base-level statements, so `ITraceAccess` actually returns a range of lines, the breakpoint is set on the first one.

Stack Frames represent the nesting of invoked callables at runtime[25]. We create stack frames for a language concept if it has callable semantics. The only callables in C are functions, but in mbeddr, test cases, state machine transitions and component methods are callables as well. Callable semantics on extension-level does not necessarily imply a function call on the base-level. There are cases where an extension-level callable is *not* mapped to a function and where a non-callable *is* mapped to a function. Consequently, the C call stack may differ from the extension call stack shown to the user. Concepts with callable semantics on the extension-level or base-level implement `IStackFrameContributor`. The interface provides operations that determine whether a stack frame has to be created in the debugger UI and what the name of this stack frame should be.

[25] A *callable* is a language concept that contains statements and can be called from multiple call sites.

Stepping behavior is configured via the following interfaces: `IStackFrameContributor`, `ISteppable`, `ISteppableContext` `IStepIntoable` and `IDebugStrategy`. Fig. 9.11 shows an overview.

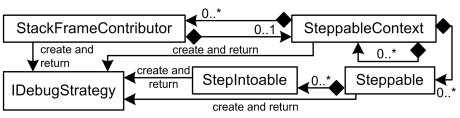


Figure 9.11: Structure of language concepts implementing the stepping-related interfaces. The boxes represent language concepts implementing the interfaces discussed in the text. Those concepts define the containments, so this figure represents a typical setup.

The methods defined by these interfaces return *strategies* that determine where the debugger may have to stop next if the user selects a stepping operation (remember that the debugger framework sets breakpoints to implement stepping). New strategies can be added without changing the generic execution aspect of the framework.

Stepping relies on `ISteppable` contributing *step over* and *step into* strategies. Many `ISteppables` are embedded in an `ISteppableContext` (e.g. `Statements` in `StatementLists`). Strategies may delegate to the containing `ISteppableContext` to determine where to stop next (the `ancestor` strategy in the `foreach` example).

For *step into* behavior, an `ISteppable` specifies those subtrees in which instances of `IStepIntoable` may be located (the `array` and `len` expressions in the `foreach` case). The debugger searches these subtrees at debug-time and collects all instances of `IStepIntoable`. An `IStepIntoable` represents a callable invocation (a `FunctionCall`), and the returned strategies suspend the debugger within the callable.

Step out behavior is provided by implementors of `IStackFrameContributor` (mentioned earlier). Since a callable can be called from many program locations, the call site for a particular invocation cannot be determined by inspecting the program structure; a call stack is needed. We use the ordered list of `IStackFrameContributors`, from which the tree-level call stack is derived, to realize the *step out* behavior. By "going back" (or "out") in the stack, the call site for the current invocation is determined. For *step out*, the debugger locates the enclosing `IStackFrameContributor` and asks it for its *step out* strategies.

Strategies implement `IDebugStrategy` and are responsible for setting breakpoints to implement a particular stepping behavior. Language extensions can either implement their own strategies or use predefined ones. Those include: setting a breakpoint on a particular node; searching for `IStepIntoables` in expression subtrees (step into) or delegating to the outer stack frame (step out).

For supporting *Watches*, language concepts implement `IWatchProvider` if they directly contribute one or more items into the Watch window. An `IWatchProviderContext` contains zero or more watch providers. Typically these are concepts that own statement lists, such as `Functions` or `IfStatements`. If the debugger is suspended on any particular statement, we can find all visible watches by iterating through all ancestor `IWatchProviderContexts` and asking them for their `IWatchProv`
Fig. 9.12 shows the typical structure of the concepts.

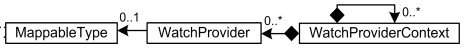


Figure 9.12: Typical structure of language concepts implementing the Watches-related interfaces

`IWatchProvider` implements the `contributeWatch- ables` operation. It has access to the C variables available in the native C debugger. Based on those, it creates a set of Watchables. The method may hide a base-level C variable (because it is irrelevant to the extension-level), promote C variable to a Watchable or create additional Watchables

based on the values of C variables. The representation of a Watchable often depends on the variable's type *as expressed in the extension program*. This type may be different from the one in the C program. For example, we represent values of type `Boolean` with *true* and *false*, even though they are represented as `int`s in C. As the Watchable is created, we specify the type that should be used. Types that should be used in this way must implement `IMappableType`. Its method `mapVariable` is responsible for computing a type-appropriate representation of a value.

■ *More Examples.* To illustrate our approach further, we have implemented the debugging behavior for mbeddr C and all default extensions. We discuss some interesting cases in this section.

We encounter many cases where we cannot know statically which piece of code will be executed when *stepping into* a callable. Consider polymorphic calls on interfaces.

The mbeddr components extension provides interfaces with operations, as well as components that `provide` and `use` these interfaces. The component methods that implement interface operations are generated to base-level C functions. The same interface can be implemented by *different* components, each implementation ending up in a *different* C function. A client component only specifies the *interface* it uses, not the component. Hence, we cannot know statically which C function will be called if an operation is invoked on the interface. However, we do know statically all components that implement the interface, so we know *all possible C functions* that may be invoked. A strategy implemented specifically for this case sets breakpoints on the first line *of all of these functions* to make sure we stop in the first line of any of them if the user *steps into* a method invocation[26].

In many cases a single statement on the extension-level is mapped to several statements or whole blocks on the base-level. *Stepping over* the single extension-level statement must step over the whole block or list of statements in terms of C. An example is the `assert` statement used in test cases. It is mapped to an `if` statement. The debugger has to step over the complete `if` statement, independent of whether the condition in the `if` evaluates to `true` or `false`. Note that we get this behavior for free[27]: the `assert` statement sets a breakpoint on the base-level counterpart of the *next tree-level statement*. It is irrelevant how many lines of C text further down this is.

An extension can provide language concepts with callable semantics that are not translated to a C function. In case a user *steps into* one of those concepts, the debugger has to create a stack frame on the tree-level call stack, although there is no corresponding stack frame on the C call stack. An example is triggering a transition in the state machine

[26] We encounter a similar challenge in state machines: as an event is fired into a state machine, we do not know which transition will be triggered. Consequently we set breakpoints in all transitions (translated to `case` branches in a `switch` statement) of the state machine.

[27] Remember that we never actually step over statements, we always set breakpoints at those next possible code locations where the debugger may have to stop next.

extension[28], which should cause the debugger to create a tree-level stack frame even though a transition is mapped on an `if` statement, not to a C function call. To achieve this, the `Transition` implements `IStackFrameContributor` and returns `true` from `isExtStackFrame`. Its `getStackFrameName` is shown below: the tree-level stack frame has the name of the `event` that triggered the transition and the name of the owning `state`.

[28] Triggering has callable semantics, as in "we are *in* the transition".

```
public string getStackFrameName() {
  this.ancestor<State>.name+" - "+this.trigger.event.name; }
```

The code snippet below shows a testing scenario with three stack frame contributors: a function `main`, two test cases, and an `executeTests` expression:

```
int32_t main() {
  return executeTests testIntRandom, testRealRandom;
}

test case testIntRandom {
  assert randomInt() != randomInt();
}

test case testRealRandom {
  assert randomReal() != randomReal();
}
```

The `main` function remains a function in generated C. The test case is mapped to a function as well, and it is possible to step into test cases. So these two have callable semantics *and* are mapped to functions. The `executeTests` expression is *also* mapped to a function (which invokes tests and sums up the number of failures) but it does *not* have callable semantics: the generated function is invisible to the user. Since all of them are mapped to functions, they implement the `IStackFrameContributor` interface. `isExtStackFrame` returns `true` for concepts with callable semantics (function and test case), but it returns `false` for the `ExecuteTestExpression`. `getStackFrameName` for the function and the test case returns the name of the respective node (`this.name`); it returns `null` for the `ExecuteTestExpression`. This way the debugger creates tree-level stack frames for called `functions` and `test cases` with their respective name. The following snippet shows the implementation for `functions`:

```
string getStackFrameName() {
  return this.name;
}

boolean isExtStackFrame() {
  true;
}
```

```
void contributeStepOutStrategies(list<IDebugStrategy> res) {
  if (name.equals("main")) new ResumeWithoutBreakpoint();
  else new StepToOuter(this);
}
```

For stepping out, we use the `StepToOuter` strategy which suspends the execution within the outer stack frame contributor. In C, the `main` function is called directly by the debugger, so there is no outer frame. We use `ResumeWithoutBreakpoint` in this case.

In contrast, the `ExecuteTestExpression` (which is also generated into a function) uses a `DelegateToOuter` to delegate the request to its outer stack frame contributor, i.e. a `Function`. We must delegate the request, since an `ExecuteTestExpression` has no callable semantics and does therefore not know where the execution should be suspended after a `step out`. The following code shows the `createBreakpoint` implementation of the custom strategy used in this case:

```
void configure(IStrategyConfiguration strategyConfiguration) {
  this.dslStackMapping = conf.getDSLStackMapping();
  this.strategyConfiguration = strategyConfiguration;
}

void createBreakpoint() throws Exception {
  list<IDebugStrategy> res = new arraylist<IDebugStrategy>();
  node<IStackFrameContributor> outerContributor =
            getOuterContributor(this, dslStackMapping);
  outerContributor.contributeStepOutStrategies(res);
  foreach strategy in res {
    strategy.configure(strategyConfiguration);
    strategy.createBreakpoint();
  }
}
```

Going back to the example, consider the debugger is suspended in the *testIntRandom* test case: the tree-level call stack contains *testIntRandom* on top of *main*. In contrast, the C call stack has another stack frame for the function generated from the `ExecuteTestExpression` between the two tree-level stack frames. A `step out` suspends the debugger in this function. However, the user expects to get back to `main` so we have to show the call stack to just contain the `main` function. The framework hides the stack frame for `ExecuteTestExpression`, since its *isExtStackFrame* implementation returns `false`. We can now perform a `step into` which causes the debugger to suspend in *testRealRandom*

Extensions may provide custom data types that are mapped to one or more data types or structures in the generated C. The debugger has to reconstruct the representation in terms of the extension from the base level data. For example, the state of a component is represented by a `struct` that has a member for each of the component fields. Component operations are mapped to C functions. In addition to the formal arguments declared for the respective operation, the generated C

function also takes this `struct` as an argument. However, to support the polymorphic invocations discussed earlier, the type of this argument is `void*`. Inside the operation, the `void*` is cast down to allow access to the component-specific members. The debugger performs the same downcast to be able to show Watchables for all component fields.

■ *Discussion.* To evaluate the suitability of our solution for our purposes, we revisit the requirements described earlier.

Modularity Our solution requires no changes to the base language or its debugger implementation to specify the debugger for an extension. Also, independently developed extensions retain their independence if they contain debugger specifications[29].

[29] In particular, MPS' capability of incrementally including language extensions in a program *without defining a composite language first* is preserved in the face of debugger specifications.

Framework Genericity The extension-dependent aspects of the debugger behavior are extensible. In particular, stepping behavior is factored into strategies, and new strategies can be implemented by a language extension. Also, the representation of Watch values can be customized by making the respective type implement `IMappableType` in a suitable way.

Simple Debugger Definition This challenge is solved by the debugger definition DSL. It supports the definition of stepping behavior and watches in a declarative way, without concerning the user with implementation details of the framework or the debugger backend.

Limited Overhead Our solution generates no debugger specific code at all (except the debug symbols added by compiling the C code with debug options). Instead we rely on trace data to map the extension-level to base-level and ultimately to text. This is a trade-off: first, the language workbench must be able to provide trace information. Second, the generated C text cannot be modified by a text processor before it is compiled and debugged, since this would invalidate the trace data[30] . Our approach has another advantage: we do not have to change existing transformations to generate debugger-specific code. This keeps the transformations independent of the debugger.

[30] The C preprocessor works, it is handled correctly by the compiler and debugger

Debugger Backend Independence We use the Eclipse CDT Debug Bridge to wrap the particular C debugger, so we can use any compatible debugger without changing our infrastructure. Our approach requires no changes to the native C debugger itself, but since we use breakpoints for stepping, the debugger must be able to handle a reasonable number of breakpoints[31]. The debugger also has to provide an API for setting and deleting breakpoints, for querying the currently visible symbols and their values, as well as for querying the code location where the debugger suspended.

[31] Most C debuggers support this, so this is not a serious limitation.

9.2.6 What's Missing?

The support from language workbenches for building debuggers for the DSLs defined with the language workbench is not where it should be. Especially in the face of extensible languages or language composition, the construction of debuggers is still a lot of work. The example discussed in Section 9.2.5 is *not* part of MPS in general, but instead a framework that has been built specifically for mbeddr — although we believe that the architecture can be reused more generally.

Also, ideally, programs should be debuggable at any abstraction level: if a multi-step transformation is used, then users should be able to debug the program at any intermediate step. Debug support for for Xbase-based DSLs is a good example for this; but it is only one translation step, and it is a solution that is specifically constructed for Xbase and Java, and not a generic framework.

So there is a lot of room for innovation in this space.

10
Modularization, Reuse and Composition

Language modularization, extension and composition (LMR&C) is an important ingredient to the efficient use of DSLs, just like reuse in general is important to software development. We discuss the need for LMR&C in the context of DSL design in , where we introduce the four classes of LMR&C . In this chapter, we look at the implementation approaches taken by the example tools Xtext, Spoofax and MPS.

10.1 Introduction

When modularizing and composing languages, the following challenges have to be addressed:

- The concrete and the abstract syntax have to be combined. Depending on the kind of composition, this requires the embedding of one syntax into another one. This, in turn, requires modular grammars.
- The static semantics, i.e. the constraints and the type system have to be integrated. For example in case of language extension, new types have to be "made valid" for existing operators.
- The execution semantics have to be combined as well. In practice, this may mean mixing the code generated from the composed languages, or composing the generators.
- Finally, the IDE that provides code completion, syntax coloring, static checks and other relevant services has to be extended and composed as well.

In this chapter we show how each of those is addressed with the respective tools. We don't discuss the general problems any further,

since those have been discussed in the chapter on DSL design

10.2 MPS Example

With MPS two of these challenges outlined above — composability of concrete syntax and modular IDEs — are a completely solved problem. Modular type systems are reasonably well supported. Semantic interactions are hard to solve in general, but can be handled reasonably in many relevant cases, as we show in this section as well. However, as we will see, in many cases, languages have to be designed *explicitly for reuse*, in order to make them reusable. After-the-fact reuse, without considering it during the design of the reusable language, is possible only in limited cases. However, this is true for reuse in software generally.

We describe LMR&C with MPS based on examples. At the center of this section is a simple `entities` language. We then build additional language to illustrate LMR&C . Fig. 10.1 illustrates these additional languages. The `uispec` (user interface specification) language illustrates *referencing* with `entities`. `relmapping` (relational database mapping) is an example of *reuse* with separated generated code. `rbac` (role-based access control) illustrates reuse with intermixed generated code. `uispec_validation` demonstrates *extension* (of the `uispec` language) and *embedding* with regards to the expressions language.

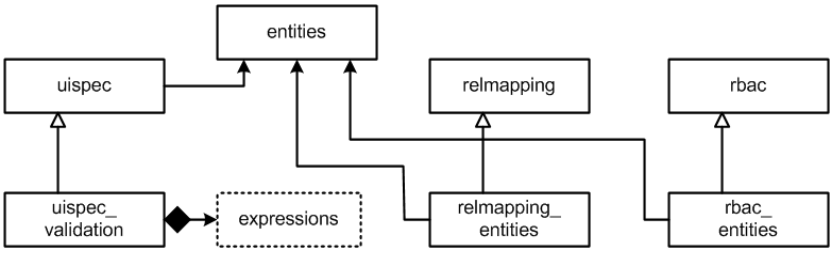


Figure 10.1: `entities` is the central language. `uispec` defines UI forms for the entities. `uispec_validation` adds validation rules, and composes a reusable expressions language. `relmapping` provides a reusable database mapping language, `relmapping_entities` adapts it to the entities language. `rbac` is a reusable language for specifying permissions; `rbac_entities` adapts this language to the `entities` language.

10.2.1 Implementing the Entities Language

At the center of the language extensions we will build later, we use a simple `entities` language. Below is an example model. *Modules* are root nodes. They live as top level elements in models[1].

[1] Referring back to the terminology introduced in the DSL design section , root nodes (and their descendants) are considered *fragments*, while the models are partitions (actually, they are XML files).

```
module company {
  entity Employee {
```

```
    id : int
    name : string
    role : string
    worksAt : Department
    freelancer : boolean
  }
  entity Department {
    id : int
    description : string
  }
}
```

■ *Structure and Syntax.* Fig. 10.2 shows a UML diagram of the structure of the `entities` language. Each box represents a language concept.

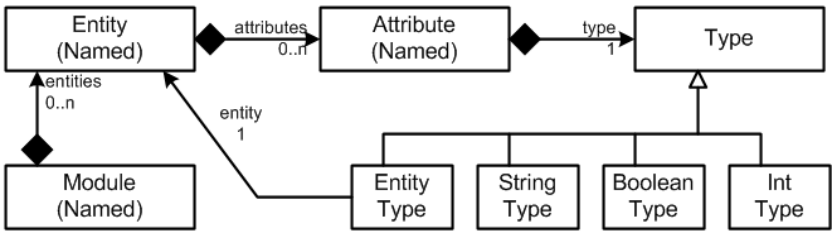


Figure 10.2: The abstract syntax of the entities language. Entities have attributes, those have types and names. `EntityType` extends `Type` and references `Entity`. This "adapts" entities to types (cf. the Adapter pattern).

The following code shows the definition of the `Entity` concept[2]. `Entity` implements the `INamedConcept` interface to inherit a `name` property. It declares a list of children of type `Attribute` in the `attributes` collection. Fig. 10.3 shows the definition of the editor for `Entity`.

[2] This is not the complete definition, concepts can have more characteristics. This is simplified to show the essentials.

```
concept Entity extends BaseConcept implements INamedConcept
  can be root: true
  children:
    Attribute  attributes  0..n
```

```
editor for concept Entity
  node cell layout:
[/
  [> entity { name } { <]
  [> ---> (> % attributes % <) <]
    /empty cell: <default>
  }
/]
```

Figure 10.3: The editor for `Entity`. The outermost cell is a vertical list. In the first line, we use a horizontal list that contains the "keyword" `entity`, the value of the `name` property and an opening curly brace. In the second line we use indentation and a vertical arrangements of the contents of the `attributes` collection. Finally, the third line contains the closing curly.

■ *Type System.* For the `entities` language, we specify two simple

typing rules. The first one specifies that the type of the primitives (`int`, `string`) is a clone of themselves:

```
rule typeof_Type applicable for concept = Type as type {
  do {
    typeof(type) :==: type.copy;
  }
}
```

The only other typing rule is an equation that defines the type of the attribute as a whole to be the type of the attribute's `type` property, defined as `typeof(attribute) :==: typeof(attribute.type);`.

■ *Generator.* From `entities` models we generate Java Beans expressed in MPS' `BaseLanguage`. For the `entities` language, we need a *root mapping rule* and *reduction rules*. Root mapping rules can be used to create new top level artifacts from existing top level artifacts (they map a fragment to another fragment). In our case we generate a Java class from an `Entity`. Reduction rules are in-place transformations. Whenever the transformation engine encounters an instance of the specified source concept somewhere in a program tree, it removes that source node and replace it with the result of the associated template. In our case we have to reduce the various types (`int`, `string`, etc.) to their Java counterparts. Fig. 10.4 shows a part of the mapping configuration for the `entities` language.

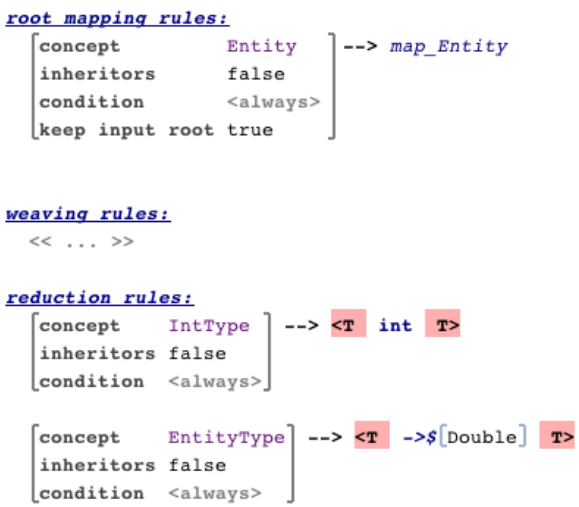


Figure 10.4: The mapping configuration for the `entities` language. The root mapping rule for `Entity` specifies that instances of `Entity` should be transformed with the `map_Entity` template. The reduction rules use inline templates. For example, the `IntType` is replaced with the Java `int` and the `EntityRefType` is reduced to a reference to the class generated from the target entity. The `->$` is a reference macro. It contains code (not shown) that "rewires" the reference to `Double` to a reference to the class generated from the referenced `Entity`.

Fig. 10.5 shows the `map_Entity` template. It generates a Java class — notice the complete structure of a Java class is present, because that is how BaseLanguage defines the editor for a Java class. We then generate a field for each entity `Attribute`. To do this we first create

a prototype field in the class (`private int aField;`). Then we use macros to "transform" this prototype into an instance for each `Entity` attribute. We first attach a `LOOP` macro to the whole field. It contains an expression `node.attributes;` where `node` refers to the input `Entity`[3]. We then use a `COPY_SRC` macro to transform the type. `COPY_SRC` copies the input node (the inspector specifies the current attribute's type as the input here) and applies reduction rules. So instances of the types defined as part of the `entities` language are transformed into a Java type using the reduction rules defined in the mapping configuration above. Finally we use a property macro (the `$` sign) to change the `name` property of the field we generate from the dummy value `aField` to the name of the attribute we currently transform (once again via an expression in the inspector).

[3] This code is entered in the Inspector window and is not shown in the screenshotq

```
root template
input Entity

public class $[map_Entity] extends <none> implements <none>
  <<static fields>>

  <<static initializer>>
  $LOOP$[private $COPY_SRC$[int] $[aField]; ]
  <<properties>>
  <<initializer>>
  public map_Entity() {
    <no statements>
  }

  $LOOP$[public void $[setter]($COPY_SRC$[int] newValue) {
           <<placeholder>> pre-set : $[attr]
           this.aField = newValue;
         }
  $LOOP$[public $COPY_SRC$[int] $[getter]() {
           return aField;
         }
```

Figure 10.5: The template for creating a Java class from an `Entity`. The running text explains the details. The `«placeholder»` is a special concept used later.

10.2.2 *Language Referencing*

■ *Structure and Syntax.* We define a language `uispec` for defining user interface forms based on the `entities`. Fig. 10.6 shows the abstract syntax and below is an example model. Note how the form is another, separate fragment. It is a *dependent* fragment, since it references elements from another fragment (expressed in the `entities` language). Both fragments are *homogeneous* since they consist of sentences expressed in a single language.

```
form CompanyStructure
  uses Department
  uses Employee
  field Name: textfield(30) -> Employee.name
  field Role: combobox(Boss, TeamMember) -> Employee.role
  field Freelancer: checkbox -> Employee.freelancer
  field Office: textfield(20) -> Department.description
```

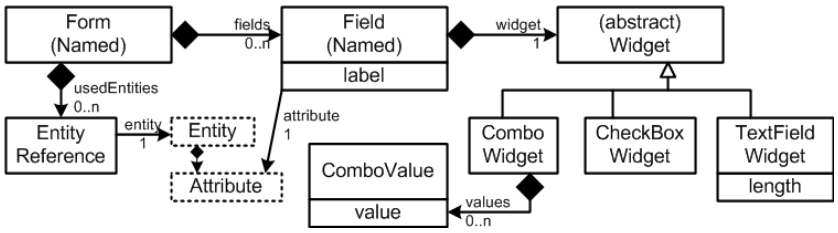


Figure 10.6: The abstract syntax of the `uispec` language. Dotted lines represent classes from another language (here: the `entities` language). A `Form` contains `EntityReferences` that connect to an `entities` model. A form also contains `Fields`, each referencing an `Attribute` from an `Entity` and containing a `Widget`.

The `uispec` language extends[4] the `entities` language. This means, that concepts from the `entities` language can be used in the definition of language concepts in the `uispec` language. A `Form` owns a number of `EntityReferences`, which in turn reference the `Entity` concept. Also, `Fields` refer to the `Attribute` that shall be edited via the field. Below is the definition of the `Field` concept. It owns a `Widget` and refers to an `Attribute`.

[4] MPS uses the term "extension" whenever the definition of one language uses or referes to concepts defined in another language. This is not necessarily an example of language extension as defined in this book.

```
concept Field extends BaseConcept implements <none>
  properties:
    label : string
  children:
    Widget      widget      1
  references:
    Attribute   attribute   1
```

■ *Type System.* The language enforces limitations regarding which widget can be used with which attribute type (a `checkbox` widget requires a Boolean type, a `ComboWidget` requires a `string` type). The necessary typing rule is defined in the `uispec` language and references types from the `entities` language. The following is the code for the type check.

```
checking rule checkTypes {
  applicable for concept = Field as field
  overrides false
  do {
    if (field.widget.isInstanceOf(CheckBoxWidget)
        && !(field.attribute.type.isInstanceOf(BooleanType))) {
      error "checkbox can only be used with booleans" -> field.widget;
```

```
    }
    if (field.widget.isInstanceOf(ComboWidget)
        && !(field.attribute.type.isInstanceOf(StringType))) {
      error "combobox can only be used with strings" -> field.widget;
    }
  }
}
```

■ *Generation.* The defining characteristic of language referencing is that the two languages only *reference* each other, and the instance fragments are dependent, but *homogeneous*. No syntactic integration is necessary in this case. In this example, the generated code exhibits the same separation. From the `Form` definition, we generate a Java class that uses Java Swing to build the UI form. It *uses* the beans generated from the entities: the classes are instantiated, and the setters are called. The generators are separate, but they are *dependent* because they share information. Specifically, the `uispec` generator knows about the names of the generated entity classes, as well as the names of the setters and getters. This dependency is implemented by defining a couple of behavior methods on the `Attribute` concept that are called from both generators (the colon represents the node cast operator and binds tightly; the code below casts the attribute's parent to `Entity` and then accesses the `name` property):

```
concept behavior Attribute {
  public string qname() {
    this.parent : Entity.name + "." + this.name;
  }
  public string setterName() {
    "set" + this.name.toFirstUpper();
  }
  public string getterName() {
    "get" + this.name.toFirstUpper();
  }
}
```

The original `entities` fragment is still *sufficient* for the transformation that generates the Java Bean. The `uispec` fragment is not sufficient for generating the UI, it needs the `entities` fragment. This is not surprising since *dependent* fragments can never be sufficient for a transformation, the transitive closure of all dependencies has to be made available.

10.2.3 Language Extension

We extend the MPS base language with block expressions and placeholders. These concepts make writing generators *that generate base language code* much simpler. Fig. 10.7 shows an example. We use a screenshot instead of text because we use non-textual notations (the big brackets) and color.

Figure 10.7: block expressions (in blue) are basically anonymous inline methods. Upon transformation, a method is generated that contains the block content, and the block expression is replaced with a call to this method. Block expressions are used mostly when implementing generators; this screenshot shows a generator that uses a block expressions.

■ *Structure and Syntax.* A block expression is a block that can be used where an `Expression` is expected[5]. The block can contain any number of statements; `yield` can be used to "return" values from within the block[6]. An optional name property of a block expression is used as the name of the generated method. The generator of the block expression in Fig. 10.7 transforms it into this structure:

```
// the argument to setName is what was the block expression,
// it is replaced by a method call to the generated method
aEmployee.setName(retrieve_name(aEmployee, widget0));

...

public String retrieve_name(Employee aEmployee, JComponent widget0) {
  String newValue = ((JTextField) widget0).getText();
  return newValue;
}
```

The `jetbrains.mps.baselanguage.exprblocks` language extends MPS' BaseLanguage. To make a block expression valid where BaseLanguage expects an `Expression`, `BlockExpression` extends `Expression`. Consequently, fragments that use the `exprblocks` language, can now use `BlockExpressions` in addition to the concepts provided by the BaseLanguage. The fragments become *heterogeneous*, because languages are syntactically mixed.

```
concept BlockExpression extends Expression implements INamedConcept
  children:
    StatementList body 1 specializes: <none>
```

■ *Type System.* The type of the `yield` statement is the type of the expression that is yielded, specified by `typeof(yield) :==: typeof(yield.result);` (the type of `yield 1;` would be `int`). Since the `BlockExpression` is used as an `Expression`, it has to have a type as well. Since it is not explicitly specified, the type of the `BlockExpression` is the common super type of the types of all the `yields`. The following typing rule computes this type:

```
var resultType ;
for (node<BlockExpressionYield> y :
        blockExpr.descendants<concept = BlockExpressionYield>) {
  resultType :==: typeof(y.result);
}
typeof(blockExpr) :==: resultType;
```

[6] So, in some sense, a block expression is an "inlined method", or a closure that is defined and called directly.

■ *Generator.* The generator for `BlockExpressions` reduces the new concept to pure BaseLanguage: it performs assimilation. It transforms a *heterogeneous* fragment (using BaseLanguage and `exprblocks`) to a *homogeneous* fragment (using only BaseLanguage). The first step is the creation of the additional method for the block expression (Fig. 10.8).

```
concept     BlockExpression
inheritors  false
condition   <always>
-->
weave_BlockExpression
context : (node, genContext, operationContext)->node< > {
            node<ClassConcept> cls = node.ancestor<concept = ClassConcept, +>;
            genContext.get copied output for (cls);
          }
```

Figure 10.8: We use a weaving rule to create an additional method for this. A weaving rule processes an input element (`BlockExpression`) by creating another node in a different place. The context function defines this other place. In this case, it simply gets the class in which we have defined the block expression.

The template shown in Fig. 10.9 shows the creation of the method. It assigns a mapping label to the created method. The mapping label creates a a mapping between the `BlockExpression` and the created method. We will use this label to refer to this generated method when we generate the method call that replaces the `BlockExpression` (shown in Fig. 10.10).

```
<TF b2M public $COPY_SRC$[string] $[amethod]($LOOP$v2P[ $COPY_SRC$[int] $[a]]) { TF> ]])
  $COPY_SRCL$[return "hallo"; ]
}
```

Figure 10.9: The generator creates a method from the `BlockExpression`. It uses `COPY_SRC` macros to replace the `string` type with the computed return type of the block expression, inserts a computed name, adds a parameter for each referenced variable outside the block, and inserts all the statements from the block expression into the body of the method (using the `COPY_SRCL` macro that iterates over all of the statements in the `ExpressionBlock`). The `blockExprToMethod` mapping label is used later in the method call.

A second concept introduced by the `exprblocks` language is the `PlaceholderStatement`. It extends `Statement` so it can be used inside method bodies. It is used to mark locations at which subsequent generators can add additional code. These subsequent generators will use a reduction rule to replace the placeholder with whatever they want to put at this location. It is a means to building extensible generator.

```
public void caller( ) {
  int j = 0;
  <TF [ ->$[callee]($LOOP$[$COPY_SRC$[j]]) ] TF>;
}
```

Figure 10.10: Here we generate the call to the previously generated method. We use the mapping label `blockExprToMethod` to refer to the correct method (not shown; happens inside the ->$ macro). We pass in the environment va riables as actual arguments.

Both, `BlockExpression` and `PlaceholderStatement` will be used in subsequent examples of LMR&C .

A particularly interesting feature of MPS is the ability to use several extensions of the same base language in a given program *without defin-*

ing a combining language. For example, a user could decide to use the block expression language defined above together with the `dispatch` extension discussed in Section 6.2. This is a consequence of MPS' projectional nature[7]. Let us consider the potential cases for ambiguity:

[7] These same benefits are also exploited in case of embedding multiple independent languages.

Same Concpet Name: The used languages may define concepts with the same name as the host language. This will not lead to ambiguity because concepts have a unqiue ID as well. A program element will use this ID to refer to the concept whose instance it represents.

Same Concrete Syntax: The projected representation of a concept is not relevant to the functioning of the editor. The program would still be unambiguous to MPS even if *all elements had the same notation*. Of course it would be confusing to the users.

Same Alias: If two concepts that are valid at the same location use the same alias, then, as the user types the alias, it is not clear which of the two concepts should be instantiated. This problem is solved by MPS opening the code completion window and requiring the user to explicitly select which alternative to choose. Once the user has made the decision, the unique ID is used to create an unambiguous program tree.

10.2.4 Language Reuse with Separated Generated Code

Language reuse covers the case where a language that has been developed independent of the context in which it should be reused. The respective fragments remain *homogeneous*. In this chapter, we cover two alternative cases: the first case (in this subsection) addresses a persistence mapping language. The generated code is separate of the code generated from the entities language. The second case (discussed in the next subsection) described a language for role-based access control. The generated code has to be "woven into" the `entities` code to check permissions when setters are called.

■ *Structure and Syntax.* `relmapping` is a reusable language for mapping arbitrary data to relational tables. The `relmapping` language supports the definition of relational table structures, but leaves the actual mapping to the source data unspecified. As you adapt the language to a specific reuse context, you have to specify this mapping. The following code shows the reusable part: a database is defined that contains tables with columns. Columns have (database-specific) data types.

```
Database CompanyDB
  table Departments
    number id
    char descr
  table People
    number id
```

```
    char name
    char role
    char isFreelancer
```

Fig. 10.11 shows the structure of the relmapping language. The abstract concept `ColumnMapper` serves as a hook: if we reuse this language in a different context, we extend this hook by context-specific code.

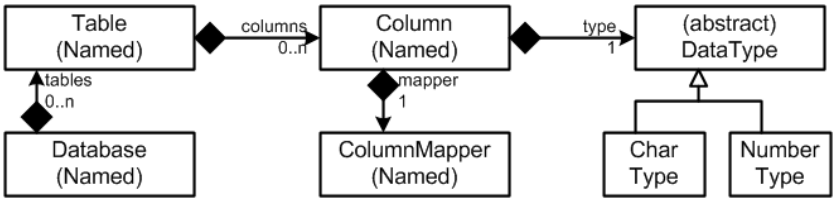


Figure 10.11: A `Database` contains `Tables` which contain `Columns`. A column has a name and a type. A column also has a `ColumnMapper`. This is an abstract concept that determines where the column gets its data from. It is a hook intended to be specialized in sublanguages that are context-specific.

The `relmapping_entities` language extends `relmapping` and adapts it for reuse with the `entities` language. To this end, it provides a subconcept of `ColumnMapper`, the `AttributeColMapper`, which references an `Attribute` from the `entities` language as a means of expressing the mapping from the attribute to the column. The column mapper is projected on the right of the field definition, resulting in the following (heterogeneous) code fragment[8]:

```
Database CompanyDB
  table Departments
    number id <- Department.id
    char descr <- Department.description
  table People
    number id <- Employee.id
    char name <- Employee.name
    char role <- Employee.role
    char isFreelancer <- Employee.freelancer
```

[8] This "mixed syntax" is pretty trivial since the `AttributeColMapper` just references an attribute with a qualified name (`Entity.attribute`). However, arbitrary additional syntax could be added, and we could use arbitrary concepts from the `entities` language mixed into the `relmapping` fragment.

■ *Type System.* The type of a column is the type of its `type` property. In addition, the type of the column must also conform to the type of the column mapper, so the concrete `ColumnMapper` subtype must provide a type mapping as well. This "typing hook" is implemented as an abstract behavior method `typeMappedToDB` on the `ColumnMapper`. It is acceptable from a dependency perspective to have this typing hook, since `relmapping` is designed to be reusable. With this in mind, the typing rules of the `relmapping` language look as follows:

```
typeof(column) :==: typeof(column.type);
```

```
typeof(column.type) :==: typeof(column.mapper);
typeof(columnMapper) :==: columnMapper.typeMappedToDB();
```

The `AttributeColMapping` concept from the `relmapping_entities` implements this method by mapping `int`s to `number`s, and everything else to `char`s.

```
public node<> typeMappedToDB()
  overrides ColumnMapper.typeMappedToDB {
  node<> attrType = this.attribute.type.type;
  if (attrType.isInstanceOf(IntType)) { return new node<NumberType>(); }
  return new node<CharType>();
}
```

■ *Generator.* The generated code is also separated into a reusable part (a class generated by the generator of the `relmapping` language) and a context-specific subclass of that class, generated by the `relmapping_entities` language. The generic base class contains code for creating the tables and for storing data in those tables. It contains abstract methods that are used to access the data to be stored in the columns. So the dependency structure of the generated fragments, as well as the depdendencies of the respective generators, resembles the dependency structure of the languages: the generated fragments are dependent, and the generators are dependent as well (they share the name, and implicitly, the knowledge about the structure of the class generated by the reusable relmapping generator). A `relmapping` fragment (without the concrete column mappers) is sufficient for generating the generic base class.

```
public abstract class CompanyDBBaseAdapter {

  private void createTableDepartments() {
    // SQL to create the Departments table
  }

  private void createTablePeople() {
    // SQL to create the People table
  }

  public void storeDepartments(Object applicationData) {
    StringBuilder sql = new StringBuilder();
    sql.append("insert into" + "Departments" + "(");
    sql.append("" + "id");
    sql.append(", " + "descr");
    sql.append(") values (");
    sql.append("" + "\"" + getValueForDepartments_id(applicationData) + "\"");
    sql.append(", " + "\"" + getValueForDepartments_descr(applicationData) + "\"");
    sql.append(")");
  }

  public void storePeople(Object applicationData) {
    // like above
  }

  public abstract String getValueForDepartments_id(Object applicationData);

  public abstract String getValueForDepartments_descr(Object applicationData);
```

```
  // abstract getValue methods for the People table
}
```

The subclass, generated by the generator in the `relmapping_entities` language implements the abstract methods defined by the generic superclass. The interface, represented by the `applicationData` object, has to be kept generic so any kind of user data can be passed in.

```
public class CompanyDBAdapter extends CompanyDBBaseAdapter {
  public String getValueForDepartments_id(Object applicationData) {
    Object[] arr = (Object[]) applicationData;
    Department o = (Department) arr[0];
    String val = o.getId() + "";
    return val;
  }
  public String getValueForDepartments_descr(Object applicationData) {
    Object[] arr = (Object[]) applicationData;
    Department o = (Department) arr[0];
    String val = o.getDescription() + "";
    return val;
  }
}
```

Note how this class references the Beans generated from the `entities`. So the generator for `entities` and the generator for `relmapping_entities` are dependent, the information shared between the two generator is the names of the classes generated from the entities. The code generated from the `relmapping` language is *designed* to be extended by code generated from a sublanguage (the abstract `getValue` methods). This is acceptable, since the `relmapping` language itself is designed to be extended to adapt it to a new reuse context.

10.2.5 Language Reuse with Interwoven generated code

■ *Structure and Syntax.* `rbac` is a language for specifying role-based access control, to specify access permissions for the `entities` language. Here is some example code:

```
RBAC

users:
  user mv : Markus Voelter
  user ag : Andreas Graf
  user ke : Kurt Ebert

roles:
  role admin : ke
  role consulting : ag, mv

permissions:
  admin, W : Department
  consulting, R : Employee.name
```

The structure is shown in Fig. 10.12. Like `relmapping`, it provides a hook, in this case, `Resource`, to adapt it to context languages. The sublanguage `rbac_entities` provides two subconcepts of `Resource`, namely `AttributeResource` to reference to an attribute, and `EntityResource` to refer to an `Entity`, to define permissions for entities and their attributes.

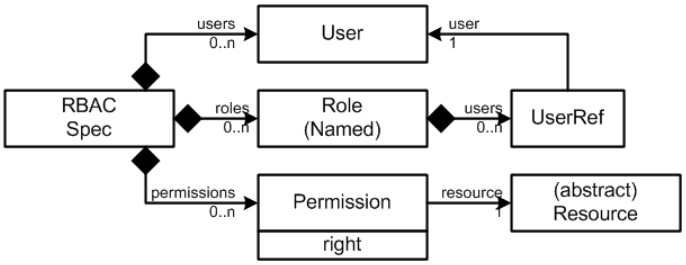


Figure 10.12: Language structure of the `rbac` language. An `RBACSpec` contains `Users`, `Roles` and `Permissions`. Users can be members in several roles. A permission assigns a right to a `Resource`.

■ *Type System.* No type system rules apply here.

■ *Generator.* What distinguishes this case from the `relmapping` case is that the code generated from the `rbac_entities` language is *not* separated from the code generated from `entities`. Instead, inside the setters of the Java beans, a permission check is required.

```
public void setName(String newValue) {
  // check permissions (from rbac_entities)
  if (!new RbacSpecEntities().currentUserHasWritePermission("Employee.name")) {
    throw new RuntimeException("no permission");
  }
  this.name = newValue;
}
```

The generated fragment is homogeneous (it is all Java code), but it is *multi-sourced*, since several generators contribute to the same fragment. To implement this, several approaches are possible:

- We could use AspectJ ([9]). This way, we could generate separate Java artifacts (all single-sourced) and then use the aspect weaver to "mix" them. However, we don't want to introduce the complexity of yet another tool, AspectJ, here, so we will not use this approach.
- An interceptor ([10]) framework could be added to the generated Java Beans, with the generated code contributing specific interceptors (effectively building a custom AOP solution). We will not use this approach either, since it would require the addition of a whole interceptor framework to the `entities`. This seems like overkill.
- We could "inject" additional code generation templates to the existing `entities` generator from the `rbac_entities` generator. This

[9] http://www.eclipse.org/aspectj/

[10] http://en.wikipedia.org/wiki/Interceptor_pattern

would make the generators *woven* as opposed to just dependent. Assuming this would work in MPS, this would be the most elegant solution. But it does not.

- We could define a hook in the generated Java Beans code and then have the `rbac_entities` generator contribute code to this hook. This is the appraoch we will use. The generators remain dependent, they have to agree on the way the hook works.

Notice that only the AspectJ solution can work without any preplanning from the perspective of the `entities` language, because it avoids mixing the generated code artifacts (it is handled "magically" by AspectJ). All other solutions require the original `entities` generator to "expect" certain extensions.

In our case, we have modified the original generator in the `entities` language to contain a `PlaceholderStatement` (Fig. 10.13). In every setter, the placeholder acts as a hook at which subsequent generators can add statements. While we have to preplan *that* we want to extend the generator in this place, we don't have to predefine *how*. The placeholder contains a key into the session object that points to the currently processed attribute. This way, the subsequent generator can know from which attribute the method with the placeholder in it was generated.

```
$LOOP$[public void $[setter]($COPY_SRC$[int] newValue) {
    <<placeholder>> pre-set : $[attr]
    this.aField = newValue;
}]
```

Figure 10.13: This generator fragment creates a setter method for each attribute of an entity. The `LOOP` iterates over all attributes. The `$` macro computes the name of the method, and the `COPY_SRC` macro on the argument type computes the type. The placeholder is used to mark the location at which the permission check will be inserted by a subsequent generator.

The `rbac_entities` generator contains a reduction rule for `PlaceholderStatements`. So when it encounters a placeholder (that has been put there by the `entities` generator) it removes it and inserts the code that checks for the permission (Fig. 10.14). To make this work we have to make sure that this generator runs *after* the `entities` generator (since the `entities` generator has to create the placeholder first) and *before* the BaseLanguage generator (which transforms BaseLanguage code into Java text for compilation). We use generator priorities, i.e. a partial ordering, to achieve this.

10.2.6 Language Embedding

■ *Structure and Syntax.* `uispec_validation` extends `uispec`, it is a sublanguage of the `uispec` language. It supports writing code such as the following in the UI form specifications:

```
reduction rules:
  concept     PlaceholderStatement
  inheritors false
  condition   (node, genContext, operationContext)->boolean {
                node.name.equals("pre-set");
              }
    --> content node:
        public void dummy() {
          <TF {{ // transparent block                                                    TF>
              // check permissions (from rbac_entities)
              if (new ->$RbacSpecEntities().currentUserHasWritePermission("$res
                  ")) { throw new RuntimeException("no permission"); }
              }}
        }
```

Figure 10.14: This reduction rule replaces `PlaceholderStatements` with a permission check. Using the condition, we only match those placeholders whose identifier is `pre-set` (notice how we have defined this identifier in Fig. 10.13). The inserted code queries another generated class that contains the actual permission check. A runtime exception is thrown if the check fails.

```
form CompanyStructure
  uses Department
  uses Employee

  field Name: textfield(30) -> Employee.name validate lengthOf(Employee.name) < 30
  field Role: combobox(Boss, TeamMember) -> Employee.role
  field Freelancer: checkbox -> Employee.freelancer
        validate if (isSet(Employee.worksAt)) Employee.freelancer == true else
                    Employee.freelancer == false
  field Office: textfield(20) -> Department.description
```

Writing the expressions is supported by embedding an `expressions` language. Fig. 10.15 shows the structure. To be able to use the expressions, the user has to use a `ValidatedField` instead of a `Field`. `ValidatedField` is also defined in `uispec_validation` and is a subconcept of `Field`.

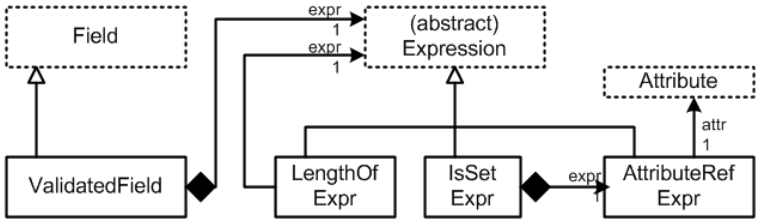


Figure 10.15: The uispec_validation language defines a subtype of `uispec.Field` that contains an `Expression` from an embeddable expression language. The language also defines a couple of additional expressions, specifically the `AttributeRefExpr`, which can be used to refer to attributes of entities.

To support the migration of existing models that use `Field` instances, we provide an intention: the user can press `Alt-Enter` on a `Field` and select `Make Validated Field`. This transforms an existing `Field` into a `ValidatedField`, so that validation expressions can be entered[11]. The core of the intention is the following script, which performs the actual transformation:

[11] Alternatively it could be arranged (with 5 lines of code) that users could simply type `validate` on the right of a field definition to trigger the transformation code below.

```
execute(editorContext, node)->void {
    node<ValidatedField> vf = new node<ValidatedField>();
    vf.widget = node.widget;
    vf.attribute = node.attribute;
    vf.label = node.label;
    node.replace with(vf);
}
```

The `uispec_validation` language extends the `uispec` language. We also extend the existing, embeddable `expressions` language, so we can use `Expressions` in the definition of our language. `ValidatedField` has a property `expr` that contains the validation expression. As a consequence of polymorphism, we can use any existing subconcept of `Expression` in validations. So without doing anything else, we could write `20 + 40 > 10`, since integer literals and the `+` and `>` operators are defined as part of the composed `expressions` language. However, to write anything useful, we have to be able to reference entity

attributes from within expressions[12]. To achieve this, we create the `AttributeRefExpr` as shown in Fig. 10.15. We also create `LenghtOf` and `IsSetExpression` as further examples of how to adapt an embedded language to its new context — i.e. the `uispec` and `entities` languages.

[12] We argued in the design part () that, in order to make an embedded language useful with its host language, it has to be extended: the following expressions are examples of this.

The `AttributeRefExpr` references an `Attribute` from the `entities` language; however, it may only reference those attributes of those entities that are used in the form in which we define the validation expression. The following is the code for the search scope:

```
(model, scope, referenceNode, linkTarget, enclosingNode)
                            ->join(ISearchScope | sequence<node< >>) {
  nlist<Attribute> res = new nlist<Attribute>;
  node<Form> form = enclosingNode.ancestor<concept = Form, +>;
  for (node<EntityReference> er : form.usedEntities) {
    res.addAll(er.entity.attributes);
  }
  res;
}
```

Notice that the actual syntactic embedding of the `expressions` language in the `uispec_validation` language is no problem at all as a consequence of how projectional editors work. We simply define `Expression` to be a child of the `ValidatedField`.

■ *Type System.* The general challenge here is that primitive types such as `int` and `string` are defined in the `entities` language *and* in the embeddable `expressions` language. Although they have the same names, they are not the same types. So the two sets of types must be mapped. Here are a couple of examples. The type of the `IsSetExpression` is by definition `expressions.BooleanType`. The type of the `LengthOf`, which takes an `AttrRefExpression` as its argument, is `expressions.IntType`. The type of an attribute reference is the type of the attribute's `type` property, as in `typeof(are) :==: typeof(are.attr.type);`. However, consider now the following code:

```
  field Freelancer: checkbox -> Employee.freelancer
        validate if (isSet(Employee.worksAt)) Employee.freelancer == false else
                    Employee.freelancer == true
```

This code states that if the `worksAt` attribute of an employee is set, then its `freelancer` attribute must be `false` else it must be `true` (freelancers dont `workAt` anything). It uses the == operator from the `expressions` language. However, that operator expects two `expressions.BooleanType` arguments, but the type of the `Employee.freelancer` is `entities.BooleanType`. In effect, we have to override the typing rules for the expressions languages's == operator. Here is how we do it:

In the expressions language, we define so-called overloaded operation rules. We specify the resulting type for an EqualsExpression depending on its argument types. Here is the code in the expressions language that defines the resulting type to be boolean if the two arguments are Equallable:

```
operation concepts: EqualsExpression
  left operand type: new node<Equallable>()
  right operand type: new node<Equallable>()
operation type:
  (operation, leftOperandType, rightOperandType)->node< > {
    <boolean>;
  }
```

In addition to this code, we have to specify that expressions.BooleanType is a subtype of Equallable, so this rule applies if we use equals with two expressions.BooleanType arguments. We have to tie this overloaded operation specification into a regular type inference rule.

```
rule typeof_BinaryExpression {
  applicable for concept = BinaryExpression as binaryExpression
  do {
    node<> opType = operation type( binaryExpression , left , right );
      if (opType != null) {
        typeof(binaryExpression) :==: opType;
      } else {
        error "operator " + binaryExpression.concept.name +
              " cannot be applied to these operand types " +
              left.concept.name + "/" + right.concept.name
          -> binaryExpression; }
      }
  }
```

To override these typing rules to work with entities.BooleanType, we simply provider another overloaded operation specification in the uispec_validation language:

```
operation concepts: EqualsExpression
  one operand type: <boolean> // this is the entities.BooleanType!
operation type:
  (operation, leftOperandType, rightOperandType)->node< > {
    <boolean>;  // this is the expressions.BooleanType
  }
```

■ *Generator.* The generator has to create BaseLanguage code, which is then subsequently transformed into Java text. To deal with the transformation of the expressions language, we can do one of two things:

- Either we can use the expressions language existing to-text generator and wrap the expressions in some kind of TextHolderStatement[13].
- Alternatively, we can write a (reusable) transformation from expressions code to BaseLanguage code; these rules would get used as

[13] Remember that we cannot simply embed text in BaseLanguage, since that would not work structurally: no concept in BaseLanguage expects "text" as children. A wrapper is necessary.

part of the transformation of `uispec` and `uispec_validation` code to BaseLanguage.

Since many DSLs will likely transform code to BaseLangauge, it is worth the effort to write a reusable generator from `expressions` to BaseLanguage. So we choose this second alternative. The generated Java code is multi-sourced, since it is generated by two independent code generators.

Expression constructs from the reusable `expressions` language and those of BaseLanguage are almost identical, so this generator is trivial. We create a new language project `de.voelter.mps.expressions.blgen` and add reduction rules[14]. Fig. 10.16 shows some of these reduction rules.

[14] In MPS, all "meta stuff" is called a language. So even though `de.voelter.mps.expressions.blgen` only contains a generator (from `expressions` to BaseLanguage) it is still a *language* in MPS terminology.

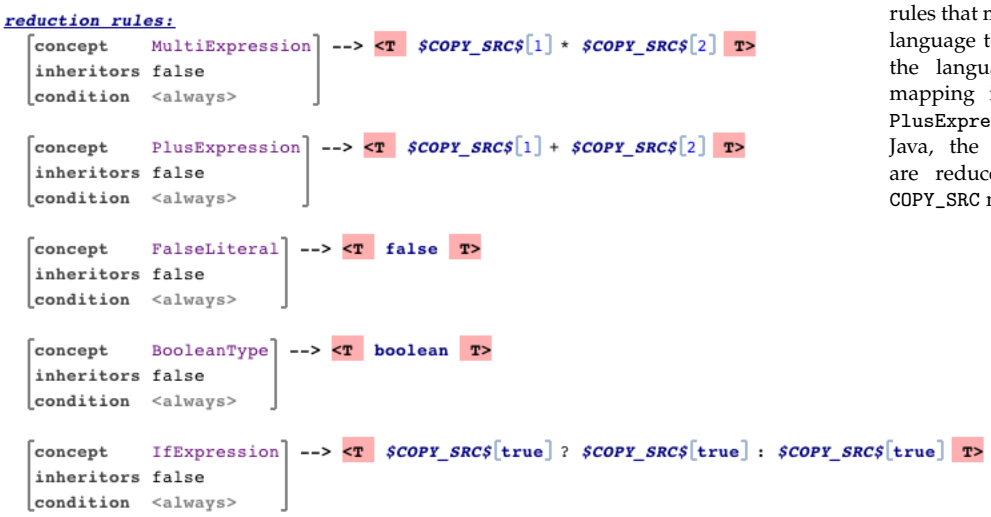


Figure 10.16: A number of reduction rules that map the reusable `expressions` language to BaseLanguage (Java). Since the languages are very similar, the mapping is trivial. For example, a `PlusExpression` is mapped to a `+` in Java, the `left` and `right` arguments are reduced recursively through the `COPY_SRC` macro.

In addition to these, we also need reduction rules for those new expressions that we have added specifically in the `uispec_validation` language (`AttrRefExpression, isSetExpression, LengthOf`). Those transformations are defined in `uispec_validation`, since this language is *not* reusable — it is specifically designed to integrate the `uispec` and the `expressions` languages. As an example, Fig. 10.17 shows the rule for handling the `AttrRefExpression`. The validation code itself is "injected" into the UI form via the same placeholder reduction as in the case of the `rbac_entities` language.

Language extension can also be used to prohibit the use of certain concepts of the base language in the sublanguage, at least in certain contexts. As a simple (but admittedly relatively useless) example,

```
reduction rules:
  [concept    AttributeRefExpr]  --> content node:
  [inheritors false                    public void dummy() {
  [condition  <always>                   Object anObj = null;
                                         <TF [ ->$[anObj].->$[toString]() ] TF>;
                                       }
```

Figure 10.17: References to entity attributes are mapped to a call to their getter method. The tempalte fragment (inside the TF) uses two reference macros (->$) to "rewire" the object reference to the Java bean instance, and the `toString` method call to a call to the getter.

we restrict the use of certain operators provided by the reusable expression language inside validation rules in `uispec_validation`. This can be achieved by implementing a `can be ancestor` constraint on `ValidatedField`.

```
can be ancestor:
  (operationContext, scope, node, childConcept)->boolean {
    return !(childConcept == concept/GreaterEqualsExpression/ ||
             childConcept == concept/LessEqualsExpression/);
  }
```

10.2.7 *Language Annotations*

In a projectional editor, the CS of a program is projected from the AST. A projectional system always goes from AS to CS, never from CS to AS (as parsers do). This has the important consequence that the CS does not have to contain all the data necessary to build the AST (which in case of parsers, is necessary). This has two consequences:

- A projection may be *partial* in the sense that the AS contains data that is not shown in the CS. The information may, for example, only be changeable via intentions (discussed in Section **??**), or the projection rule may project some parts of the program only in some cases, controlled by some kind of configuration data.
- It is also possible to project *additional* CS that is not part of the CS definition of the original language. Since the CS is never used as the information source, such additional syntax does not confuse the tool (in a parser-based tool the grammar would have to be changed to take into account this additional syntax to not derail the parser).

In this section we discuss the second alternative since it constitutes a form of language composition: the additional CS is composed with the original CS defined for the language. The mechanism MPS uses for this is called annotations. We have seen annotations when we discussed templates: an annotation is something that can be attached to arbitrary program elements and can be shown together with CS of the annotated element. In this section we use this approach to implement an alternative approach for the entity-to-database mapping. Using this

approach, we can store the mapping from entity attributes to database columns directly in the `Entity`, resulting in the following code:

```
module company
  entity Employee {
    id : int -> People.id
    name : string -> People.name
    role : string -> People.role
    worksAt : Department -> People.departmentID
    freelancer : boolean -> People.isFreelancer
  }

  entity Department {
    id : int -> Departments.id
    description : string -> Departments.descr
  }
```

This is a heterogeneous fragment, consisting of code from `entities`, as well as the annotations (e.g. `-> People.id`). From a CS perspective, the column mapping is "embedded" in the `Entity`. In the AST the mapping information is also actually stored in the `entities` model. However, the definition of the `entities` language does not know that this additional information is stored and projected "inside" entities. No modification to the `entities` language is necessary.

■ *Structure and Syntax.* We define an additional language `relmapping_annotations` which extends the `entities` language as well as the `relmapping` language. In this language we define the following concept:

```
concept AttrToColMapping extends NodeAnnotation
  references:
    Column column 1
  properties:
    role = colMapping
  concept links:
    annotated = Attribute
```

`AttrToColMapping` concept extends `NodeAnnotation`, a concept predefined by MPS. Concepts that extend `NodeAnnotation` have to provide a `role` property and an `annotated` concept link. Structurally, an annotation is a child of the node it annotates. So the `Attribute` has a new child of type `AttrToColMapping`, and the reference that contains the child is called `@colMapping` — the value of the `role` property. The `annotated` concept link points to the concept *to which this annotation can be added*. `AttrToColMapping`s can be annotated to instances of `Attribute`.

While structurally the annotation is a child of the annotated node, in the CS the relationship is reversed: The editor for `AttrToColMapping` wraps the editor for `Attribute`, as Fig. 10.18 shows. Since the annotation is not part of the original language, it must be attached to nodes via an intention.

```
editor for concept AttrToColMapping
  node cell layout:
    [- [> attributed node <] -> ( % column % -> * R/O model access * ) -]
```

Figure 10.18: The editor for the `AttrToColMapping` embeds the editor of the concept it is annotated to (using the `attributed node` cell). It then projects the reference to the referenced column. This way the editor of the annotation has control of if and how the editor annotated element is projected.

■ *Type System.* The same typing rules are necessary as in the `relmapping_entities` language described previously. They reside in `relmapping_annotations`.

■ *Generator.* The generator is also broadly similar to the previous example with `relmapping_entities`. It takes the `entities` model as the input, and then uses the column mappings in the annotations to create the entity-to-database mapping code.

The annotations introduced above were typed to be specific to certain target concepts (`EntityAttribute` in this case). A particularly interesting use of of annotations includes those that can be annotated to *any* language concept (formally targetting `BaseConcept`). In this case, there is no dependency between the language that contains the annotation and the language that is annotated. This is very useful for "meta data", as well as anything that can be processed generically.

An example of the first case is traceability links (Fig. 10.19). This annotation can be annotated to any language concept and adds pointers (trace links) to requirements. As a consequence of the projectional approach, the program can be shown with or without the annotations, controlled by a global switch.

Figure 10.19: **Left:** Requirements traces (green) can be annotated to any arbitrary program element. The annotation is targetted to `BaseConcept`, which means there is no explicit dependency any specific language. **Right:** Traces can then be queried in the reverse direction to find which requirements are traced from implementation code.

An example of the second case is product line variability annotations. Boolean expressions over configuration switches can be annotated to any model element (the colorful annotations in the left part of Fig. **??**). Such an annotation means that the respective element is only in the program variant, if the boolean expression is true for the given setting of configuration switches. The generic transformation simply removes

all elements whose annotation evaluates to false. The expressions can also be evaluated as part of the projection, showing the code for a given variant (the right part of Fig. **??** shows an example variant). The code is of course still editable. Details on this approach can be found in [15] and [16].

[15]
[16]

```
Variability from FM: DeploymentConfiguration
Rendering Mode: product line
module ApplicationModule from test.ex.cc.fm imports {!test} SensorModule {

  {logging}
  message list messages {
    INFO beginningMain() active: entering main function
    INFO exitingMain() active: exitingMainFunction
  }

  exported test case testVar {
    {logging}
    report(0) messages.beginningMain() on/if;
    int8_t x = SensorModule::getSensorValue(1) replace if {test} with 42;

    {logging}
    report(1) messages.exitingMain() on/if;
    assert(2) x == 10 replace if {test} with 42;

    {valueTest}
    int8_t vv = value;
    {valueTest}
    assert(3) vv == 42;

    int8_t ww = 22 replace if {valueTest} with 12 + value;
    {!valueTest}
    assert(4) ww == 22;

    {valueTest}
    assert(5) ww == 54;
  } testVar(test case)

  int32_t main(int32_t argc, string[ ] args) {
    return test testVar;
  } main (function)
}
```

```
Variability from FM: DeploymentConfiguration
Rendering Mode: variant rendering config: Production
module ApplicationModule from test.ex.cc.fm imports SensorModule {


  exported test case testVar {
    int8_t x = SensorModule::getSensorValue(1);
    assert(2) x == 10;
    int8_t ww = 22;
    assert(4) ww == 22;
  } testVar(test case)

  int32_t main(int32_t argc, string[ ] args) {
    return test testVar;
  } main (function)
}
```

Figure 10.20: **Left:** Feature dependency annotations are boolean expressions over configuration switches that determine whether the annotated program element is part of a program variant. The expressions can be annotated to arbitrary program elements. **Right:** The transformation removes all those elements for which the annotation evaluates to `false`. This can also be done in the editor to support variant-specific editing of programs.

10.3 Xtext Example

This section of the book has been written together with Christian Dietrich. Reach him via `christian.dietrich@itemis.de`.

In this section we look at an example roughly similar to the one for MPS discussed in the previous section. We start out with a DSL for entities. Here is an example program:

```
module company {
```

```
    entity Employee {
        id : int
        name : string
        role : string
        worksAt : Department
        freelancer : boolean
    }
    entity Department {
        id : int
        description : string
    }
}
```

The grammar is straight forward and should be clear if you have read the implementation part so far.

```
grammar org.xtext.example.lmrc.entity.EntityDsl with org.eclipse.xtext.common.Terminals

generate entityDsl "http://www.xtext.org/example/lmrc/entity/EntityDsl"

Module:
    "module" name=ID "{"
        entities+=Entity*
    "}";

Entity:
    "entity" name=ID "{"
        attributes+=Attribute*
    "}";

Attribute:
    name=ID ":" type=AbstractType;

Named: Module|Entity|Attribute;

AbstractType:
    BooleanType|IntType|StringType|EntityReference;

BooleanType: {BooleanType} "boolean";

IntType: {IntType} "int";

StringType: {StringType} "string";

EntityReference: ref=[Entity|FQN];

FQN: ID ("." ID)*;
```

10.3.1 Referencing

Referencing describes the case where programs written in one DSL reference (by name) program elements written in another DSL, both programs reside in different fragments and no syntactic composition is required. The example we use is the UI specification language where a `Form` defined in the UI model refers to `Entities` from the language defined above, and `Fields` in a form refers to entity `Attribute`. Here is some example code:

```
form CompanyStructure
  uses Department // reference to Department entity
  uses Employee   // reference to Employee entity
```

```
field Name: textfield(30)              -> Employee.worksAt
field Role: combobox(Boss, TeamMember) -> Employee.role
field Freelancer: checkbox             -> Employee.freelancer
field Office: textfield(20)            -> Department.description
```

■ *Structure* Referencing concepts defined in another language works by importing the target meta model then defining references to concepts defined in this meta model[17]. Here is the header of the `uispec` language:

```
grammar org.xtext.example.lmrc.uispec.UispecDsl with org.eclipse.xtext.common.Terminals

import "http://www.xtext.org/example/lmrc/entity/EntityDsl" as entity

generate uispecDsl "http://www.xtext.org/example/lmrc/uispec/UispecDsl"
```

[17] Note that the references to entities and fields do not technically reference into an entity source file. Instead, these references refer to the EMF objects in the AST that has been parsed from the source file. So, a similar approach can be used to reference to other `EObjects`. It does not matter whether these are created via Xtext or not. This is reflected by the fact that the grammar of the `uispec` language does not refer to the *grammar* of the `entity` language, but to the derived meta model.

Importing a meta model means that the respective *meta classes* can now be used. Note that the meta model import does not make the *grammar rules* visilble, so the meta classes can only be used in references and as base types (as we will see later). In case of referencing, we use them in references:

```
EntityReference:
    "uses" entity=[entity::Entity|FQN];

Field:
    "field" label=ID ":" widget=Widget "->" attribute=[entity::Attribute|FQN];
```

To make this work, no change is required in the `entities` language[18]. However, the workflow generating the `uispec` language has to be changed. The `genmodel` file for the meta model has to be registered in the `StandaloneSetup`[19].

[18] This is true as long as the referenced elements are in the index. The index is used by Xtext to resolve references against elements that reside in a different model file. By default, all elements that have a `name` attribute are in the index. `Entity` and `Attribute` have names, so this works automatically.

[19] This is necessary so that the EMF code generator, when generating the meta classes for the `uispec` language knows where the generated Java classes for the `entities` languages reside. This is an EMF technicality and we won't discuss it in any more detail

```
bean = StandaloneSetup {
    ...
    registerGenModelFile = "platform:/resource/org.xtext.example.lmrc.entity/
                src-gen/org/xtext/example/lmrc/entity/EntityDsl.genmodel"
}
```

We have to do one more customization to make the language work smoothly. The only `Attributes` that should be visible are those from the entities referenced in the current `Form`'s `uses` clauses, and they should be referenced with a qualified name (`Employee.role` instead of just `role`). Scoping has to be customized to achieve this:

```
public IScope scope_Field_attribute(Field context, EReference ref) {
```

```
    Form form = EcoreUtil2.getContainerOfType(context, Form.class);
    List<Attribute> visibleAttributes = new ArrayList<Attribute>();
    for (EntityReference useClause : form.getUsedEntities()) {
        visibleAttributes.addAll(useClause.getEntity().getAttributes());
    }
    Function<Attribute, QualifiedName> nameComputation = new Function<Attribute, QualifiedName>() {
        @Override
        public QualifiedName apply(Attribute a) {
            return QualifiedName.create(((Entity)a.eContainer()).getName(), a.getName());
        }
    };
    return Scopes.scopeFor(visibleAttributes, nameComputation , IScope.NULLSCOPE);
}
```

This scoping function performs two tasks: first, it finds all the `Attributes` of all `used` entities. We collect them into a list `visibleAttributes`. The second part of the scoping function defines a `Function` object[20] that represents a function from `Attribute` to `QualifiedName`. In the implementation method `apply` we create a qualified name made from two parts: the entity name and the attribute name (the dot between the two is default behavior for the `QualifiedName` class). When we create the scope itself in the last line we pass in the list of attributes as well as the function object. Xtext's scoping framework uses the function object to determine the name by which each of the attributes is referencable from this particular context.

[20] Note how we have to use the ugly function object notation, because Java does not provide support for closures or lambdas at this point. I am looking forward to being able to do this with Xtend, which does support closures.

■ *Type System* As we have discussed in the design section , dealing with type systems in the referencing case is not particularly challenging, since the type system of the referencing language can be built with knowledge of the type system of the referenced language.

■ *Generators* The same is true for generators. Typically they just share knowledge about the naming of generated code elements.

10.3.2 Reuse

Referencing concerns the case where the referencing language is built with the knowledge about the referenced language, so it can have direct dependencies. In the example above, the `uispec` language uses `Entity` and `Attribute` from the `entities` language. It directly imports the meta model, so it has a direct dependency.

■ *Structure* In case of reuse, such a direct dependency is not allowed. Our goal is to combine two *independent* languages. To show this case, we again use the same example as in the MPS section. We first introduce a `db` language. It is a trivial DSL for defining relational table structures. These can optionally be mapped to a data source, but the language makes no assumption about how this data source looks (and which language is used to define it). Consequently, the grammar has no dependency on another other one, and imports no other meta

model:

```
grammar org.xtext.example.lmrc.db.DbDsl with org.eclipse.xtext.common.Terminals
generate dbDsl "http://www.xtext.org/example/lmrc/db/DbDsl"

Root:
      Database;

Database:
    "database" name=ID
    tables+=Table*;

Table:
    "table" name=ID
    columns+=Column*    ;

Column:
    type=AbstractDataType name=ID (mapper=AbstractColumnMapper)?;

AbstractColumnMapper: {AbstractColumnMapper}"not mapped";

AbstractDataType:
    CharType | NumberType;

CharType: {CharType}"char";

NumberType: {NumberType}"number";
```

Just as in the MPS example, the `Column` rule has an optional `mapper` property of type `AbstractColumnMapper`. Since it is not possible to explicitly mark rules as generating abstract meta classes, we simply define the syntax to be `not mapped`[21]. This language has been designed *for* reuse, because it has this hook `AbstractColumnMapper`, which can be customized later. But the language is still independent. In the next step, we want to be able to reference `Attributes` from the `entities` language:

[21] Since the `mapper` property in `Column` is optional, you don't ever have to type this.

```
database CompanyDB

table Departments
  number id          <- Department.id
  char descr         <- Department.description

table People
  number id          <- Employee.id
  char name          <- Employee.name
  char role          <- Employee.role
  char isFreelancer  <- Employee.freelancer
```

To make this possible, we create a new language `db2entity` which *extends* the `db` language and *references* the `entities` language[22]. This is reflected by the header of the `db2entity` language (notice the `with` clause):

[22] Notice that we only extend `DbDsl`. The `entities` meta model is just *referenced*. This is because Xtext can only extend one base grammar. For this reason we cannot embed language concepts from the `entities` language in a `db2entity` program, we can only *reference* them.

```
grammar org.xtext.example.lmrc.db2entity.Db2EntityDsl with org.xtext.example.lmrc.db.DbDsl

import "http://www.xtext.org/example/lmrc/db/DbDsl" as db
import "http://www.xtext.org/example/lmrc/entity/EntityDsl" as entity

generate db2EntityDsl "http://www.xtext.org/example/lmrc/db2entity/Db2EntityDsl"
```

We now have to overwrite the `AbstractColumnMapper` rule defined in the `db` language.

```
AbstractColumnMapper returns db::AbstractColumnMapper:
    {EntityColumnMapper} "<-" entity=[entity::Attribute|FQN];
```

We create a rule that has the same name as the rule in the supergrammar. So when the new grammar calls the `AbstractColumnMapper` rule, our new definition is used. Inside, we define the new syntax we would like to use, and as part of it, we reference an `Attribute` from the imported `entity` meta model. We then use the `{EntityColumnMapper}` action to force instantiation of an `EntityColumnMapper` object: this also implicitly leads to the creation of an `EntityColumnMapper` class in the generated `db2entity` meta model. Since our new rule `returns` an `db::AbstractColumnMapper`, this new meta class extends `AbstractColumnMapper` from the `db` meta model — which is exactly what we want.

There are two more things we have to do to make it work. First, we have to register the two `genmodel` files in the `db2entity`'s `StandaloneSetup` bean in the workflow file. Second, we have to address the fact that in Xtext, the first rule in a grammar file is the entry rule for the grammar, i.e. the parser starts consuming a model file using this rule. In our `db2entity` grammar, the first rule is `AbstractColumnMapper`, so this won't work. We have to add the real root, which is simply the same one as in the `db` language:

```
Root returns db::Root :
    Database;
```

■ *Type System* The primary task of the type system in this example would be mapping the primitive types used in the `entities` language to those used in the `db` language to make sure we only map those fields to a particular column that are type-compatible. Just as the column mapper itself, this code lives in the adapter language. It is essentially just a constraint that checks for type compatibility.

■ *Generator* Let us assume there is a generator the generates Java Beans from the `entities`. Further, we assume that there is a generator that generates all the persistence management code from `DbDSL` programs, except the part of the code that fetches the data from whatever the data source is — essentially we leave the same "hole" as we do with the `AbstractColumnMapper` in the grammar. And just in the same

way as we define the `EntityColumnMapper` in the adapter language, we have to adapt the executing code. We can use two strategies.

The first one uses the composition techniques of the target language, i.e. Java. The generated code of the `DbDsl` could for example generate an abstract class that has an abstract method `getColumnData` for each of the table columns. The generator for the adapter language would generate a concrete subclass that implements these methods to grab the data from entities[23]. This way the modularity (`entities`, `db`, `db2entity`) is propagated into the generated artifacts as well. No *generator* composition is required[24].

However, consider a situation where we'd have to generate inlined code, for reasons of efficiency e.g. in some kind of embedded system. In this case the `DbDsl` generator would have to be built in an extensible way. Assuming we use Xtend for generation, this can be done easily by using dependency injection[25]. Here is how you would do that:

- In the generator that generates persistence code from a `DbDsl` program, the code that generates the inlined "get data for column" code delegates to a class that is dependency-injected[26]. The Xtend class we delegate to would be an abstract class that has one abstract method `generateGetDataCodeFor(Column c)`.

```
class GetDataGenerator {
    def void generateGetDataCodeFor(Column c)
}


class DbDslGenerator implements IGenerator {

    @Inject GetDataGenerator gdg

    def someGenerationMethod(Column c) {

        // ...
        String getDataCode = gdg.generateGetDataCodeFor(c)
        // then embed getDataCode somewhere in the
        // template that generates the DbDsl code
    }
}
```

- The generator for the adapter language would contain a subclass of this abstract class that implements the `generateGetDataCodeFor` generator method in a way suitable to the `entities` language.
- The adapter language would also set up Google Guice dependency injection in a way to use this a singleton instance of this subclass when instances of the abstract class are expected.

10.3.3 Extension

We have already seen the mechanics of extension in the previous example, since, as a way of building the reuse infrastructure, we have

[23] This is how we did it in the MPS example.

[24] In a Java/Enterprise world this would most likely be the way we'd do it in practice. The next alternative is a bit constructed.

[25] Sometimes people complain about the fact that Xtend is a general purpose language, and not some dedicated code generation language. However, the fact that one can use abstract classes, abstract methods and dependency injection is a nice example of how and why a general purpose language (with some dedicated support for templating) is useful for building generators.

[26] This is a nice illustration of building a generator that is *intended to be extended* in some way.

extended the `db` language. In this section we look at extension in more detail. Extension is defined as syntactic integration with explicit dependencies. However, as we had discussed in the design chapter there are two use cases that feel different:

1. In one case we provide (small scale) additional syntax to an otherwise unchanged language. The `db2entity` language shown above is an example of this. The `db2entity` programs look essentially like programs written in the `db` base language, but in one (or few) particular places, something is different. In the example, the syntax for referencing attributes is such a small scale change.

2. The other case is where we create a completely new language, but reuse some of the syntax provided by the base language. This use case *feels* like embedding (we embed syntax from the base language in our new language), but in the classification according to syntactic integration and dependencies, it is still extension. Embedding would prevent explicit dependencies. In this section we look at extension with an embedding flavor.

To illustrate extension-with-embedding-flavor, we will show how to embed bBase expressions in a custom DSL. Xbase is a reusable expression language that ships with Xtext[27]. It provides primitive types, various unary and binary operators, functions and closures. As we will see it is very tightly integrated with Java[28]. As an example, we essentially create another entity language; thanks to Xbase, we will be able to write:

[27] It is also the functional core of Xtend.

[28] This is a mixed blessing. As long as you stay in a JVM world (use Java types, generate Java code), it means that many things are very simple. However, as soon as you go outside of the JVM world, a lot of things become quite complex and it is questionable whether using Xbase makes sense in this case at all.

```
 module sample {
    entity Person {
        lastname : String firstname : String String fullName(String from) {
            return "Hello " + firstname + " "  + lastname + " from " + from
        }
    }
}
```

Below is the essential part of the grammar. Note how it extends the Xbase grammar (the `with` clause) and how it uses various elements from Xbase throughout the code (those whose names start with an `X`).

```
 grammar org.xtext.example.lmrc.entityexpr.EntityWithExprDsl with
org.eclipse.xtext.xbase.Xbase

generate entityWithExprDsl
"http://www.xtext.org/example/lmrc/entityexpr/EntityWithExprDsl"

Module:
    "module" name=ID "{"
        entities+=Entity*
    "}";

Entity:
```

```
    "entity" name=ID "{"
        attributes+=Attribute* operations+=Operation*
    "}";

Attribute:
    name=ID ":" type=JvmTypeReference;

Operation:
    type=JvmTypeReference name=ID "("  (parameters+=FullJvmFormalParameter
        (',' parameters+=FullJvmFormalParameter)*)? ")"
            body=XBlockExpression;
```

Let's look at some of the details. First, the `type` properties of the `Attribute` and the `Operation` are not defined by our grammar, instead we use a `JvmTypeReference`. This makes all Java types legal in this place[29]. We use an `XBlockExpression` as the body of `Operation`, which essentially allows us to use the full Xbase language inside the body of the `Operation`. To make the `parameters` visible, we use the `FullJvmFormalParameter` rule[30]. The `JvmModelInferrer`, shown below, maps a model expressed with this language to a structurally equivalent Java "model". By doing this, we get a number of benefits "for free", including scoping, typing and a code generator. Let us look at this crucial step in some detail .

[29] Limiting this to the primitive types (or some other subset of the JVM types) requires a scoping rule.

[30] Above we wrote that Xbase is tightly integrated with the JVM and Java. The use of `FullJvmFormalParameter` and `JvmTypeReference` is a sign of this. However, the next piece of code makes this even clearer.

```
class EntityWithExprDslJvmModelInferrer extends AbstractModelInferrer {

    @Inject extension IQualifiedNameProvider
    @Inject extension JvmTypesBuilder

    def dispatch void infer(Entity entity,
                            IAcceptor<JvmDeclaredType> acceptor,
                            boolean isPrelinkingPhase) {
        ...
    }
}
```

This Xtend class extends `AbstractModelInferrer` and implements its `infer` method to create structurally equivalent Java code as an EMF tree and registers it with the `acceptor`. The method is marked as `dispatch`, so it can be polymorphically overwritten for various language concepts. We override it for the `Entity` concept. We also have the `IQualifiedNameProvider` and `JvmTypesBuilder` injected. The latter provides a builder API for creating all kinds JVM objects, such as fields, setters, classes or operations.

```
acceptor.accept(
    entity.toClass( entity.fullyQualifiedName ) [
        documentation = entity.documentation
        ...
    ]
)
```

At the top level, we map the `Entity` to a `Class`. `toClass` is one of the builder methods defined in the `JvmTypesBuilder`. The class we create should have the same name as the `entity`; the name of the class is passed into the constructor. The second argument, written conveniently behind the parentheses, is a closure. Inside the closure, there is a variable `it` that refers to the target element (the class in this case). It is possible to omit the `it`, so when we write `documentation = ...` this actually means `it.documentation = ...`. In other words, we set the documentation of the created class to be the documentation of the `entity`. Next we create a field, a getter and a setter for each of the attributes of the `Entity` and add them to the `Class`' `members` collection:

```
for ( attr : entity.attributes ) {
    members += attr.toField(attr.name, attr.type)
    members += attr.toGetter(attr.name, attr.type)
    members += attr.toSetter(attr.name, attr.type)
}
```

`toField`, `toGetter` and `toSetter` are all builders contributed by the `JvmTypesBuilder`. Let us better understand what they do. Here is the implementation of `toSetter`. Note that the first argument supplied by the object in front of the dot, i.e. the `Attribute`.

```
public JvmOperation toSetter(EObject sourceElement, final String name, JvmTypeReference typeRef) {
    JvmOperation result = TypesFactory.eINSTANCE.createJvmOperation();
    result.setVisibility(JvmVisibility.PUBLIC);
    result.setSimpleName("set" + nullSaveName(Strings.toFirstUpper(name)));
    result.getParameters().add(toParameter(sourceElement, nullSaveName(name), cloneWithProxies(typeRef)));
    if (name != null) {
        setBody(result, new Functions.Function1<ImportManager, CharSequence>() {
            public CharSequence apply(ImportManager p) {
                return "this." + name + " = " + name + ";";
            }
        });
    }
    return associate(sourceElement, result);
}
```

The method first creates a `JvmOperation` and sets the visibility and the name. It then creates a parameter that uses the `typeRef` passed in as the third argument as its type. As you can see, all of this happens via model-to-model transformation. This is important, because these created objects are used implicitly in scoping and typing. The body, however, is created textually; it is not needed for scoping or typing, it is used only in code generation. Since that is a to-text transformation anyway, it is good enough to represent the body of the setter as text already at this level. The last line is important: it associates the source element (the `Attribute` in our case) with the created element (the setter `Operation` we just created). As a consequence of this asso-

ciation, the Xbase scoping and typing framework can work its magic of providing support for our DSL without any further customization!

Let us now continue our look a the implementation of the `JvmModelInferrer` for the `Entity`. The last step before our detour was that we created fields, setters and getters for all attributes of our `Entity`. We have to deal with the operations of our `Entity` next.

```
for ( op : entity.operations ) {
    members += op.toMethod(op.name, op.type) [
        for (p : op.parameters) {
            parameters += p.toParameter(p.name, p.parameterType)
        }
        body = op.body
    ]
}
```

This code should be easy to understand. We create a method for each operation using the respective builder method, pass in the name and type, create a parameter for each of the parameters of our source operation and then assign the body of the created method to be the body of the operation in our DSL program. The last step is particularly important. Notice that we don't clone the body, we assign the object *directly*. Looking into the `setBody` method (the assignment is actually mapped to a setter in Xtend), we see the following:

```
public void setBody(JvmExecutable logicalContainer, XExpression expr) {
    if (expr == null)
        return;
    associator.associateLogicalContainer(expr, logicalContainer);
}
```

The `associateLogicalContainer` method is what makes the automatic support for scoping and typing happen:

- because the operation is the container of the expression, the expression's type and the operation's type must be compatible
- because the expression(s) live inside the operation, the parameters of the operation, as well as the current class's fields, setters and getters are in scope automatically.

This approach of mapping a DSL to Java "code" via this model transformation works nicely as long as it maps to Java code in a simple way. In the above case of entities, the mapping is trivial and obvious. If the semantic gap becomes bigger, the `JvmTypeInferrer` becomes more complicated. However, what is really nice is this: within the type inferrer, you can of course use Xtend's template syntax (discussed in) to create implementation code. So it is easy to mix model transformation

(for those parts of a mapping that is relevant to scoping and type calculation) and then use traditional to-text transformation using Xtend's powerful template syntax for the detailed implementation aspects.

■ *Generator* The JVM mapping shown above already constitutes the full semantic mapping to Java. We map entities to Java classes and fields to members and getters/setters[31]. So in fact, we do not have to do anything else to get a generator, we can reuse the existing Xbase-to-Java code generator.

In case we build a language that cannot easily be mapped to a JVM model we can still reuse the XBase expression compiler by injecting the `JvmModelGenerator` and then delegating to it at the respective granularity.

You can also change or extend the behavior of the default `JvmModelGenerator` by overriding its `_internalDoGenerate(EObject, IFileSystemAccess)` method for your particular language concept[32].

[31] Notice how we did specify the implementation of the getters and setters, although this would not strictly be necessary for the scoping and typing support we get from the JVM mapping.

[32] Notice that you also have to make sure via Guice that your subclass is used, and not the `JvmModelGenerator`.

■ *Extending Xbase* In the above example we had embedded the (otherwise unchanged) Xbase language into a simple DSL. Let us now look at how to extend Xbase itself by adding new literals and new operators. We start by defining a literal for dates:

```
XDateLiteral:
    'date' ':' year=INT '-' month=INT '-' day=INT;
```

These new literals should be literals in terms of Xbase, so we have to make them subtypes of `XLiteral`. Notice how we override the `XLiteral` rule defined in Xbase. We have to repeat its original contents, there is no way to "add" to the literals.

```
XLiteral returns xbase::XExpression:
    XClosure |
    XBooleanLiteral |
    XIntLiteral |
    XNullLiteral |
    XStringLiteral |
    XTypeLiteral |
    XDateLiteral;
```

We use the same approach to add an additional operator represented with ===:

```
OpEquality:
    '==' | '!=' | '===';
```

Xtend supports operator overloading by mapping operators to methods that can be overridden. The === operator does not yet exist in

Xtend, so we have to specify the name of the method that should be called if the operator shows up in a program. The second line of the `initializeMapping` method maps the new operator to a method named `operator_identity`:

```
public class DomainModelOperatorMapping extends OperatorMapping {

    public static final QualifiedName IDENTITY = create("===");

     @Override
    protected void initializeMapping() {
        super.initializeMapping();
        map.put(IDENTITY, create("operator_identity"));
    }
}
```

We implement this method in a new class which we call `ObjectExtensions2`[33][34]:

```
public class ObjectExtensions2 {
    public static boolean operator_identity(Object a, Object b) {
        return a == b;
    }
}
```

[33] The existing class `ObjectExtensions` contains the implementations for the existing `==` and `!=` operators, hence the name

[34] Through the `operator_identity` operation, we have expressed all the semantics: the Xbase generator will generate a call to that operation in the generated Java code.

We also want to override the existing + operator for the new date literals. We don't have to specify the mapping to a method name, since the mapping for + is already defined in Xbase. However, we have to provide an overloaded implementation of + for dates:

```
public class DateExtensions {
    public static Date operator_plus(Date a, Date b) {
        return new Date(a.getTime() + b.getTime());
    }
}
```

To make Xtend aware of these new classes, we have to register them. In order to do so, we extend the `ExtensionClassNameProvider`. It associates the classes that contain the operator implementation methods with the types for which these classes contain the applicable methods:

```
public class DomainModelExtensionClassNameProvider extends ExtensionClassNameProvider {

    @Override
    protected Multimap<Class<?>, Class<?>> simpleComputeExtensionClasses() {
        Multimap<Class<?>, Class<?>> result = super.simpleComputeExtensionClasses();
        result.put(Object.class, ObjectExtensions2.class);
        result.put(Date.class, DateExtensions.class);
        return result;
    }
}
```

We now have to extend the type system: it has to be able to derive the types for date literals. We create a type provider that extends the default `XbaseTypeProvider`[35]:

[35] Don't forget to register this class with Guice, just like all the other DSL-specific subclasses of framework classes.

```
@Singleton
public class DomainModelTypeProvider extends XbaseTypeProvider {

    @Override
    protected JvmTypeReference type(XExpression expression,
                    JvmTypeReference rawExpectation, boolean rawType) {
        if (expression instanceof XDateLiteral) {
            return _type((XDateLiteral) expression, rawExpectation, rawType);
        }
        return super.type(expression, rawExpectation, rawType);
    }

    protected JvmTypeReference _type(XDateLiteral literal,
                    JvmTypeReference rawExpectation, boolean rawType) {
        return getTypeReferences().getTypeForName(Date.class, literal);
    }
}
```

We don't have to specify typing rules for the `===` operator. Since we have mapped it to the `operator_identity` operation, the type system uses the types specified in this operation: the type of `===` is `boolean`, and there are no restrictions on the two arguments; they are typed as `java.lang.Object`. If customizations were required these could be done by overriding the `_expectedType` operation in `XbaseTypeProvider`.

Finally we have to extend the Xbase compiler so it can handle date literals:

```
public class DomainModelCompiler extends XbaseCompiler {

    protected void _toJavaExpression(XDateLiteral expr, IAppendable b) {
        b.append("new java.text.SimpleDateFormat(\"yyyy-MM-dd\").parse(\"" +
              expr.getYear() + "-" + expr.getMonth() + "-" +
              expr.getDay() + "\")");
    }

}
```

10.3.4 Embedding

Embedding is not supported by Xtext. The reason is that, as we can see from the , the adapter language would have to inherit from *two* base languages. However, Xtext only supports extending one base grammar.

We have shown above how to embed Xbase expressions into a custom DSL. However,as we have discussed, this is an example of extension with embedding flavor: we create *a new* DSL into which we embed the existing Xbase expressions. So we only have to extend from *one* base language — Xbase. An example of embedding would be to take an existing, independent SQL language and embed it into the Entity DSL created above. This is not possible.

10.4 Spoofax Example

In this section we look at an example roughly similar to the one for MPS and Xtext discussed in the previous sections. We start with Mobl's data modelling language, which we have already seen in previous chapters.

To understand some of the discussions later, we first have to understand how Spoofax organizes languages. In Spoofax, language definitions are typically modularized and organized in directories. For example, Mobl's syntax definition comes with a module for entities, which imports modules for statements and expressions. These modules reside in the same directory:

```
module MoblEntities

imports
  MoblStatements
  MoblExpressions
```

Similarly, rewrite rules for program analysis, editor services, program transformation, and code generation are organized in modules, which are imported from Mobl's main module. Thereby, the various modules for program analysis, editor services, and program transformation are organized in subdirectories:

```
module mobl

imports
  analysis/names
  analysis/types
  analysis/checks
  editor/complete
  editor/hover
  editor/refactor
  trans/desugar
  trans/normalize
  generate
```

10.4.1 Referencing

We will illustrate references to elements written in another DSL with Mobl's screen definition language, where a `Screen` defined in the UI model refers to `Entities` from Mobl's entity language, and `Fields` in a screen refer to entity `Property`. Here is some example code:

```
 entity Task {
  name        : String description : String done        : Bool date        :
  DateTime
}
```

This fragment is written in Mobl's data definition language. It defines an entity `Task` with some properties. We discussed the language in several examples throughout the book already.

```
screen root() {
  header("Tasks")
  group {
    list(t in Task.all()) {
      item { label(t.name) }
    }
  }
}
```

This is a screen definition written in Mobl's screen definition language. It defines a root screen for a list of tasks, using the `name` of a `Task` as a label for `list` elements. There are two references to the data model: `Task` refers to an `Entity`, and `name` refers to a property of that `Entity`.

■ *Structure* When referencing elements of another language, both languages typically share a definition of identifiers. For example, the screen definition language imports the same lexical module as the data modelling language does via the expression module:

```
module MoblEntities

imports
\ldots
MoblExpressions
```

```
module MoblExpressions

imports
lexical/Common
```

```
module MoblScreens

imports
lexical/Common
```

However, Spoofax also supports the use of different, typically overlapping identifier definitions[36]. In this case, the referencing language needs to import the identifier definition of the referenced language.

Independent of the used identifiers in both languages, the reference has to be resolved. Definition sites are already defined by the referenced language. The corresponding references need to be defined in

[36] This requires scannerless parsing, since a scanner cannot handle overlapping lexical definitions.

the referencing language by using the namespaces from the referenced language:

```
"list" Item@=ID "in" Collection "{" Item* "}" -> List       {"ScreenList"}
Entity@ID "." "all" "(" ")"                   -> Collection {"Collection"}
```

This fragment from the syntax definition of the screen definition language specifies lists and items in these lists. The screen definition language defines its own namespace `Item`. Items are declared in the list head, introducing a variable for the current item of a collection. For example, the screen definition we have seen earlier defines an item `t`. When we describe the collection, we can refer to entities. The corresponding namespace `Entity` is defined in the data modelling language. The screen definition language uses the same namespace, to resolve the references into the referred language.

Similarly, the screen definition language uses the namespace `Property` from the data modelling language, to refer to properties of entities:

```
"item" "{" ItemPart* "}"                -> Item     {"Item"}
"label" "(" Item@ID "." Property@ID ")" -> ItemPart {"LabelPart"}
```

■ *Type System* Similar to the name resolution, the type system of the referencing language needs to be defined with the knowledge of the type system of the referenced language.

```
constraint-error:
  LabelPart(item, property) -> (property, "Label has to be a string.")
  where
    type := <type-of> property
    not (!type => !StringType())
```

This constraint checks whether a label is of type string or not. Therefore, it needs to determine the type of the property used as a label.

■ *Generators* The same holds for generators. Typically, they just share knowledge about the naming and typing of generated code elements.

10.4.2 Reuse

As discussed in the previous sections, referencing concerns the case where the referencing language is built with the knowledge about the referenced language, so it can have direct dependencies. In the example above, the screen definition language directly uses the namespaces and types from the entity language.

■ *Structure* In case of reuse, such a direct dependency is not allowed. Our goal is to combine two *independent* languages. To show this case, we again use the same example as in the MPS section. We first introduce a trivial DSL for defining relational table structures. These can optionally be mapped to a data source, but the language makes no assumption about how this data source looks (and which language is used to define it). Consequently, the grammar has no dependency on another other one:

```
module DBTables

  "database" DB@=ID Table*         -> Database {"DB"}
  "table" Table@=ID Column*        -> Table    {"DBTable"}
  DataType Column@=ID ColumnMapper -> Column   {"DBColumn"}

  -> ColumnMapper {"DBMissingMapper"}

  "char"   -> DataType {"DBCharType"}
  "number" -> DataType {"DBNumberType"}
```

Again, the `Column` rule has an optional `ColumnMapper` which works as the hook for reuse. The reusable language provides only a rule for a missing column mapper. Rules for a concrete mapper are missing, but can be added later in a sublanguage. In the next step, we want to be able to reference properties from Mobl's data modelling language:

```
database TaskDB

table Tasks

  char name        <- Task.name
  char description <- Task.description
```

To do this, we define an adapter module, which imports the reusable table module and the data modelling language:

```
module MoblDBAdapter

imports
  DBTables
  MoblEntities

context-free syntax

   "<-" Entity@ID "." Property@ID -> ColumnMapper {"PropertyMapper"}
```

There is only one rule in this module, which defines a concrete mapper. On the right-hand side, it uses the same sort as the rule in the table module (`ColumnMapper`). On the left-hand side, it refers to a property from the imported data modelling language.

■ *Type System* The type system needs to connect types from the abstract but reusable language to types from the language which actually

reuses it. In our example, the types of the database language needs to be connected to the primitive types used in Mobl. Constraints ensure we only map those fields to a particular column that are type-compatible:

```
module reuse/dbtables/analysis/types

rules
  constraint-error:
    DBColumn(type, _, mapper) -> (mapper, "Incompatible type")
    where
      type' := <type-of> mapper ;
      <not(compatible-types)> (type, type')

  compatible-types: _ -> <fail>
```

The code above is defined in the generic implementation of the database language. It assumes that a mapper has a type and checks if this type is compatible to the declared column type. It defines a default rule for type compatibility, which always fails.

The connection to the type system of the entity language can now be made in an adapter module:

```
module analysis/types/adapter

imports
  module analysis/types
  module reuse/dbtables/analysis/types

rules

  type-of: PropertyMapper(e, p) -> <type-of> p

  compatible-types: (DBCharType(), StringType()) -> <id>
  compatible-types: (DBNumberType(), IntType())  -> <id>
```

The first rule defines the type of a `PropertyMapper` to be the type of the property. Then, two rules define type compatibility for Mobl's `String` type with the `char` type in the table language, and Mobl's `int` type with the `num` type.

■ *Generator* Similar to the Xtext example, we can use two strategies to reuse a generator for the database language. The first strategy relies on composition techniques of the target language. Like in the MPS example, the code generator of the database language generates an abstract Java class for fetching data while Mobl's original code generator generates Java classes from entities. We can then define an additional generator, which generates a concrete subclass which fetches data from entities.

The second strategy we discussed in the Xtext example addressed the generation of inlined code, which requires an extendable generator of the reusable language. With rewrite rules, this can be easily

achieved in Spoofax. The reusable generator calls a dedicated rule for generating the inlined code, but defines only a failing implementation of this rule:

```
module reuse/table/generate

rules
  db-to-java: Column(t, n, mapper) ->
       [Field([PRIVATE], t', n), Method([PROTECTED], BoolType, n', params, stmts)] where
    n'    := <to-fetch-method-name> n ;
    param := <to-fetch-method-parameters> mapper ;
    stmts := <to-fetch-statements(|n)> mapper

  to-fetch-method-parameters: _ -> <fail>
  to-fetch-statements(|n)   : _ -> <fail>
```

This rule generates code for columns. It generates a private field and a protected method to fetch the content. This method needs a name, parameters, and an implementation. We assume that the method name is provided by the generic generator. For the other parts, the generic generator only provides failing placeholder rules. These have to be implemented in a concrete reuse setting:

```
module generate/java/adapter


imports
  generate/java
  reuse/table/generate

rules

 to-fetch-method-parameters:
   PropertyMapper(entity, property) -> [Param(type, "entity")]
   where
     type := <entity-to-java-type> entity

 to-fetch-statements(|field-name):
   PropertyMapper(entity, property) ->
         [Assign(VarRef(field-name), MethodCall(VarRef("entity"), m, []), Return(True()))]
   where
     m := <property-to-get-method-name> property
```

This adapter code generates a single parameter for the fetch method. It is named `entity` and its type is provided by a rule from the entity language generator. This rule maps entity names to Java types. For the implementation body, the second rule generates an assignment and a return statement. The assignment calls the getter method for the property. Again, the name of this getter method is provided by the entity language generator.

10.4.3 Extension

Because of Spoofax' module system and rule-based nature, language extension feels like ordinary language development. When we want to

add a new feature for a language, we simply create new modules for syntax, type system, and code generation rules. These modules import the existing modules as needed. In the syntax definition, we can extend a syntactic sort with new definition rules. In the type system, we add additional `type-of` rules for the new language constructs and define constraints for well-typedness. Finally, we add new rules to the generator, which can handle the new language constructs.

Since in case of extension, the extending language has a dependency on, and is developed with knowledge of the base language, it can be designed in a way that will not lead to parsing ambiguities.

10.4.4 Embedding

Embedding can be easily achieved in Spoofax. The procedure is very similar to reuse[37]. We will use the embedding of the Mobl's expression language into Mobl's screen definition language as another example here. Instead of just referring to entities, we reuse the whole expression language now.

[37] We discussed embedding already in the chapter on code generation (), where we embedded the target language into Stratego.

■ *Structure* There are two styles for embedding. The first one requires an embedding module which imports the main modules of the host and guest language and defines additional syntax rules for embedding (quotations and antiquotation). We have seen this in the target language embedding in templates (). In the second style, the syntax definition of the host language directly employs syntactic sorts of the guest language. For the screen definition language, we want to reuse variable definitions and expressions in list heads, as well as expressions in labels:

```
module MoblScreens

imports MoblExpressions

context-free syntax

  "list" VarDef "in" Exp "{" Item* "}" -> List     {"ScreenList"}
  "label" "(" Exp ")"                  -> ItemPart {"LabelPart"}
```

■ *Type System* The type system needs to connect the types from the host and guest languages. This can be achieved by adding typing rules for embedded and antiquoted constructs. In the embedding of expressions, we do not have antiquotes. However, we expect certain types for embedded expressions. For example, the expression in a list head needs to be a collection. The type from the variable definition needs to be compatible with the content of the collection. Finally, the expression for a label should provide a string. We can capture these requirements in constraints:

```
constraint-error:
  ScreenList(_, e, _) -> (e, "Needs to provide a collection.")
  where
    type := <type-of> e;
    <not(collection-type)> type

constraint-error:
  ScreenList(VarDef(t, _), e, _) -> (t, "Cannot assign collection elements (incompatible types).")
  where
    type  := <type-of> e ;
    type' := <collection-type> type ;
    <not(is-assignable)> (t, type')
```

■ *Generator* There are two strategies for code generation for embedded languages. If the guest language provides a suitable code generator, we can combine it with the code generator of the host language. First, we need rules which generate code for embedded constructs. These rules have to extend the host generator by delegating to the guest generator. Next, we need rules which generate code for antiquoted constructs. These rules have to extend the guest generator by delegating to the host generator.

The screen definition language needs to generate a method which returns the text of a label. Therefore, it relies on the code generator for expressions, to translate the expression into Java code:

```
generate-label-text-method:
  LabelPart(exp) -> Method([PUBLIC()], StringType(), "getText", [Return(java-exp)])
  where
    java-exp := <to-java> exp
```

Another strategy is to define a model-to-model transformation which desugars (or "assimilates") embedded constructs to constructs of the host language. This transformation is then applied first, before the host generator is applied to generate code. The embedding of a target language into Stratego is an example for this approach. The embedded target language will be represented by abstract syntax trees for code generation fragments. These trees need to be desugared into Stratego pattern matching constructs. For example, the embedded |[return |[x]|;]| will yield the following abstract syntax tree:

```
ToJava(
  Return(
    FromJava(
      Var("x")
    )
  )
)
```

In ordinary Stratego without an embedded target language, we would have written the pattern `Return(x)` instead. The corresponding abstract syntax tree looks like this:

```
NoAnnoList(
  App(
    Op("Result"),
    [Var("x")]
  )
)
```

The desugar transformation now needs to transform the first abstract syntax tree into the second one:

```
desugar-all: x -> <bottomup(try(desugar-embedded))> x

desugar-embedded: ToJava(e) -> <ast-to-pattern> e

ast-to-pattern:
  ast -> pattern
  where
    if !ast => FromJava(e) then
      pattern := e
    else
      c       := <constructor> ast ;
      args    := <arguments> ast ;
      ps      := <map(ast-to-pattern)> args ;
      pattern := NoAnnoList(App(Op(c), ps))
```

The first rule drives the desugaring of the overall tree. It tries to apply `desugar-embedded` in a bottom-up traversal. The only rule for desugaring embedded target language code matches the embedded code and applies `ast-to-pattern` to it. If this is applied to an antiquote, the contained subnode is already a regular Stratego pattern. Otherwise, the node has to be an abstract syntax tree of the target language. It is deconstructed into its constructor and subtrees, which are desugared into patterns as well. The resulting patterns and the constructor are then used to construct the overall pattern.