MARKUS VOELTER, VOELTER@ACM.ORG

DSL ENGINEERING

Part I

DSL Implementation

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 of language workbenches. For more example language implementations using more language workbenches, take a look at the LWC 11 website.

TODO: cite to LWC11

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

TODO: Show stuff in MPS with tables and graphics

TODO: reference to the respective place in the introduction

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. Note that syntactic information which 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 quite a bit.

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 "UI" 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 dichotomy between 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. A possible AST is illustrated in

Fig. 1.1. The boxes represent program elements, and the names in the boxes represent language concepts that make up the abstract syntax.

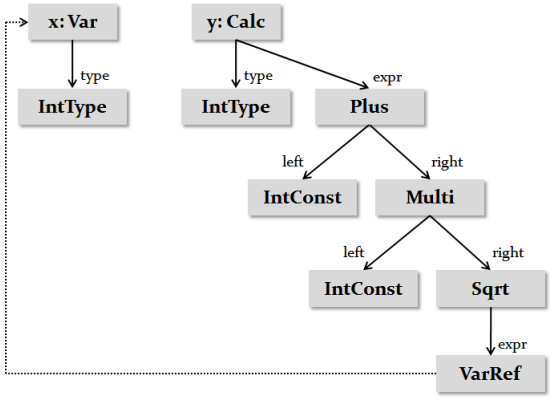


Figure 1.1: Abstract syntax tree for the above program. Boxes represent instances of language concepts, solid lines represent containment, dotted lines represent cross-references

There are two ways of defining the relationship between the CS and the AS.

AS first An existing AS is annotated with the concrete syntax. Mapping rules determine how the concrete syntax maps to existing abstract syntax structures.

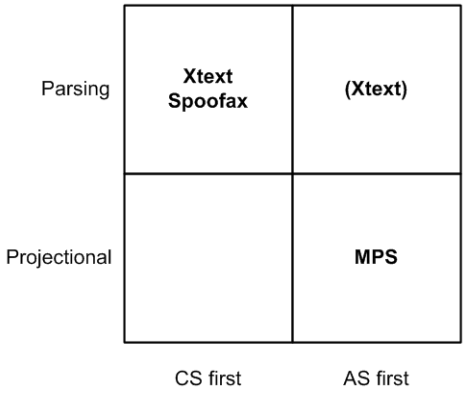
CS first From a concrete syntax definition, an abstract syntax is derived, either automatically and/or using hints in the concrete syntax specification.

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:

Parsing In parser-based systems, the abstract syntax tree is derived 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*.

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 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. In the next sections we look more closely at parsing and projection.



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 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 from information extracted from the textual source. Most modern IDEs perform this task in real-time as the user edits the program. The AST is always kept in sync with the program text. Many IDE features — such as content assist, validation, navigation, refactoring support, etc. — are based on the real-time 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 which a particular syntax must be specified. It is important to be aware of this as 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, PEG. These all pertain to a certain class of parser implementation approaches. We'll 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 cre-

ate 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 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 required is **much** lower than writing a custom parser from scratch. The grammar definition is also much more readable, maintainable and adaptable than the actual parser implementation (either custom-written or generated).

MV: Do we want to say something regarding our dislike of actions (procedural code) embedded in grammars? We'd like to separate these concerns...

EV: note that I changed lexer to scanner: lexer is more commonly used nowadays but since it's called 'scannerless parsing' it probably makes more sense to use 'scanner'

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, whitespace, etc.) by a *scanner* (sometimes also called lexer). The *parser* then constructs the actual AST from the token sequence. 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`. Modern parsing frameworks, such as ANTLR, do their own scanner generation.

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. Usually the former meaning is intended (both in this book as well as outside) 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. 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) uses scannerless parsing.

MV: [symbol tables] should be discussed in the scoping and linking section?

Grammars Grammars are the formal definition for concrete textual syntax. They consist of so-called production rules which define how valid input ("sentences") looks like. They can also be used to "produce" valid input, hence their name. Grammars form the basis for syntax definitions in text-based workbenches such as Spoofax and Xtext. 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.

Fundamentally, grammar production rules can be expressed in Backus-Naur Form (BNF)[1], written as [1]

$S ::= P_1 \ \ldots \ P_n$

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, or the patterns can be separated by the '|' operator to indicate a choice. An extension of BNF, called Extended BNF (EBNF), 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.

As an example, Fig. 1.2 shows 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.

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

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

Figure 1.2: A grammar for a simple expression language. The `NUM` symbol refers to number literals.

Figure 1.3: Classes of grammars.

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.

TODO: as opposed to what?

Parser generation was first applied in command-line tools such as `yacc` in the early seventies. 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, as illustrated in Fig. 1.3. A particular parser tool supports a specific class of grammars — e.g., ANTLR supports LL(k) and LL(*).

In the classification of Fig. 1.3, 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.

TODO: Venn diagram of grammar classes: LL, LR, ...

Supporting only a subclass of all possible context-free grammars poses restrictions on the languages that are supported b 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

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

Figure 1.4: A grammar for property access expressions of the form `customer.name` or `"Tim".length`.

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.

As an example, consider a grammar for property or field access, expressions of the form `customer.name` or `"Tim".length`:[2]

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

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 grammar can be *left-factored* 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 production rules which are recursive 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
```

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 one from Fig. 1.4, but conforms to the LL(1) grammar class.

Unfortunately, it is not possible to factor all possible context-free grammars 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.

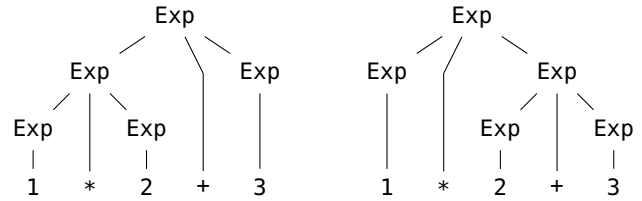
General parsers Research into parsing algorithms has produced parser generators specific to various grammar classes, but there has also been research in parsers for the full class of context-free grammars. Naive algorithms that support the full class of context-free grammars use backtracking, i.e. they allow themselves to try to match the input using another production rule even though an earlier-tried rule already seemed to match. This carries the risk of exponential execution times or non-termination and usually exhibits poor performance.

MV: I don't understand the previous sentence

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

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, 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.

For example, the expression language grammar given in Fig. 1.2 is ambiguous. 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 the grammar in Fig. 1.2. Determining whether a grammar is ambiguous is a classical, 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 of Fig. 1.4), 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 addition 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 addition operator in the highest layer to give it the lowest priority, 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 ().

TODO:

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

Ambiguity with generalized parsers General parsers accept grammars regardless of recursion or ambiguity. That is, the grammar of Fig:exp-grammar-simple 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.

Language developers can use *disambiguation filters* to indicate which interpretation should be preferred. For example, left-associativity can be indicated on a per-production basis:

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

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

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

indicating that the multiplication operator binds stronger than the addition 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.

TODO:

Compositionality encoded precedence, associativity, etc. makes grammars resistent to change and composition

grammar classes and composition don't mix

can you compose the arithmetic expressions grammar with the field access grammar Fig:field-access-grammar?

and what about

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

reserved keywords: for, SELECT, FROM

requires a context-sensitive scanner, or a parser that operates on characters instead of on tokens (a scannerless parser)

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.

In projectional editors, every program element 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). The AST is actually an ASG, 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. 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.

1.3 *Comparing Parsing and Projection*

MV: The following paragraph needs to be ripped apart and the respective points need to be moved into the respective subsubsections

There's a lot to say in favor of free text editing. The linear, two-dimensional structure of text (especially when using mono-spaced fonts) is well-known and easily understood, while text editing operations are more-or-less universal. It's very convenient to have a CS that also serves as the persistence format: it makes the language essentially tool-independent and allows fairly easy integrations with other text-based tools like versioning systems, issue trackers, documentation systems, or email clients.

1.3.1 *Editing Experience*

To edit programs, you need a suitable editor. A normal text editor is obviously not sufficient as you would be editing some textual representation of the AST. 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.

- 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 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.

- Delete actions are used to similar effect when elements are deleted. 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 in a given location. Consider again a local variable declaration `int a;`. The concept represented could be LocalVariableDeclaration, a subtype 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 and 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 them 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 rep because of that. In case of MPS this tool support is available, and hence, MPS provides a productive and pleasant working environment.

MV: Here

1.3.2 *Language Modularity*

Projectional editors can cope with textual notations that would be ambiguous if they had to be parsed. This is important in two contexts:

existing languages for which a language-specific parser had been manually written before, should now be handled with a LWB. Projectional editors can deal with any "strange" syntax defined by legacy systems.

language composition After composing separately developed languages, the resulting language may be ambiguous[3]. In projectional systems, this cannot happen. Any combination of languages will be syntactically valid (semantics is a different issue). If a composed language would be ambiguous, the user has to make a disambiguating decision as the program is built. 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. We discuss details in the chapter on language composition.

[3] Many grammar formalisms are not closed under composition.

1.3.3 *Notational Freedom*

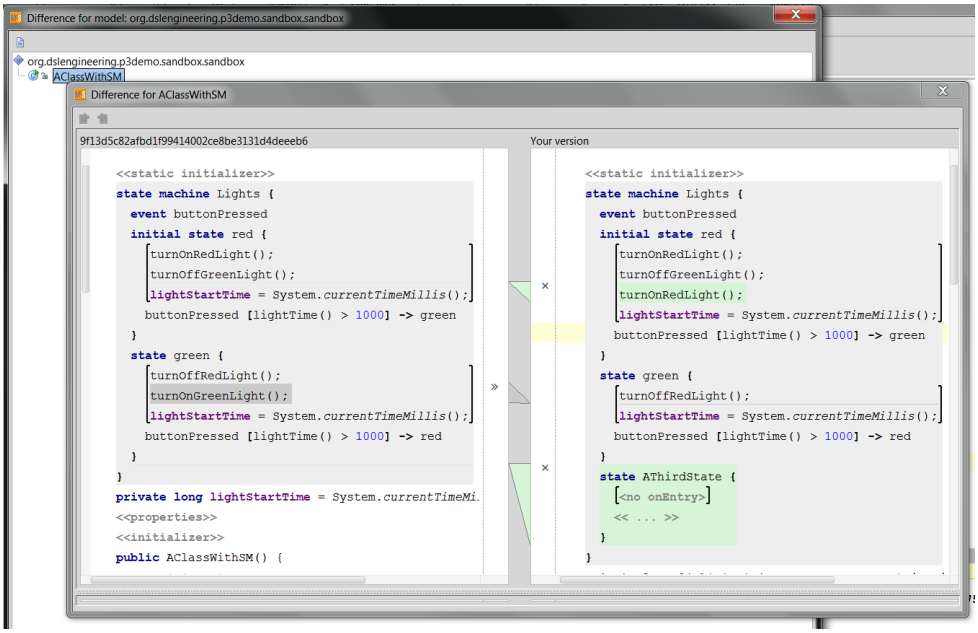
Since no parsing is used in projectional editors, and the mechanism works basically like a graphical editor, notations other than text can be used in the editor. For example, MPS supports tables as well as simple diagrams. Since these non-textual notations are handled the same way as the textual ones (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. Textual notations can also be used inside boxes or as connection labels in diagrams.

1.3.4 *Language Evolution*

Evolution of languages can be a bit more complex, since, if language concepts are not present anymore in a new revision of the language, existing instances of these concepts will break. MPS solves this problem with deprecation and "automatic" migration scripts. We will discuss this in more detail in the chapter on language evolution.

1.3.5 *Infrastructure Integration*

Special support also needs to be provided for infrastructure integration. Since the CS is not pure text, a generic persistence format needs to be devised. Therefore, special tools need to be provided for diff and merge. MPS provides integration with the usual VCS systems and handles diff and merge in the tool, using the concrete, projected syntax. Note that since every program element has a UUID, *move* can be distinguished from *delete* and *create*, providing more useable semantics for diff and merge. Fig. 1.3.5 shows an example of an MPS diff.



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 the 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, are ways to represent trees or graphs. Usually, such an AST formalism is "meta circular" in the sense that it can describe itself.

1.4.1 EMF Ecore

The Eclipse Modeling Framework (EMF) is at the core of all of Eclipse's modeling related activities. Its core component is the Ecore meta meta model, a version of the EMOF standard. The central concepts are: EClass, EAttribute, EReference and EObject, the latter providing an in-memory/run-time representation of EClass instances. EReferences can be containing or not - each EObject can be contained by at most one EReference instance. Fig. 1.4.1 shows a class diagram of Ecore.

The Ecore file is at the heart. From that, all kinds of other aspects are derived, specifically, a generic tree editor, and a generated Java API for accessing the AST of models and programs. This also forms the basis for Xtext's processing.

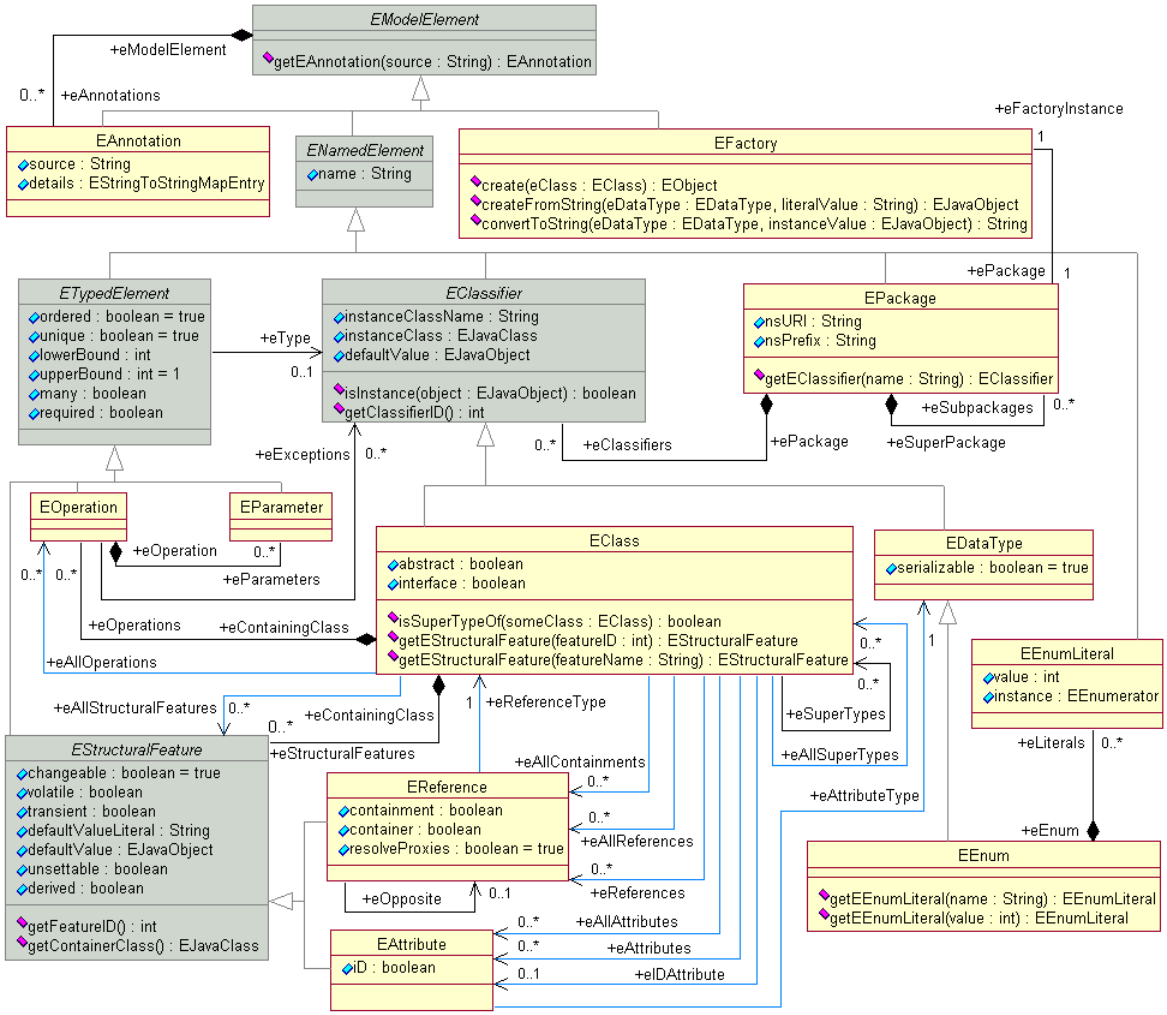
EMF has grown to be a fairly large ecology within the Eclipse community and numerous projects use EMF as a basis for model manipulation and persistence, with Ecore for meta model definition.

1.4.2 Spoofax' ATerm

Spoofax uses the ATerm format to represent abstract syntax. ATerms are a generic tree structure representation format that can be serialized textually similar to XML or JSON. ATerms consist of the following elements:

- Strings, e.g. `"Mr. White"`
- Numbers, e.g. `15`
- Lists, e.g. `[1,2,3]`
- Constructor applications, e.g. `Order(5, 15, "Mr. White")`

In addition to these basic elements, ATerms can also be annotated with additional information, e.g. `Order(5{ProductName("Apples")}, 15, "Mr. White")`. These annotations are often used to represent references to other parts of a model.



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 tools-specific fashion (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/xml2aterm tools can convert between ATerms and XML.

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 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. 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 (those stating with "concept"). In addition, concepts can have behavior methods.

1.5 *Xtext Example*

TODO: We assume an introduction of the case studies somewhere before this point

Cooling programs represent the behavioral aspect of the refrigerator language. Here is a trivial program that can be used to illustrate some of the features of the language.

```
cooling program EngineProgram0 {

    var v: int
    event e

    init { set v = 1 }

    start:
       on e { state s2 }

    state s2:
        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 s2 state, where it uses an entry action to set v back to 0. More complex programs include checks of changes of measurements (`compartment1->currentTemp`) and commands to the hardware (`do compartment1->coolOn`), as shown in the next snippet:

```
start:
    check ( compartment1->currentTemp > maxTemp ) {
        do compartment1->coolOn
        state initialCooling
    }
    check ( compartment1->currentTemp <= maxTemp ) {
        state normalCooling
    }

state initialCooling:
    check ( compartment1->currentTemp < maxTemp ) {
        state normalCooling
    }
```

Grammar basics In Xtext, the language is specified using a grammar which is a collection of *parser rules*. These rules specify (by default) the concrete syntax of a program element, as well as its mapping to the AS. 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. It 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, again using a BNF-like syntax:

```
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 as defined above. However, in 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 get a property name that holds the contents of the ID in the program text. Since nothing else is specified in the ID terminal rule, the type of this property defaults to EString, Ecore's implementation 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 asterisk specifies unbounded multiplicity), an optional init block (optionality is specified by the question mark) 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 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 is a list as well. 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 and discuss 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 with that of a +=. Inside the parentheses we expect either a CustomEvent or a Variable. 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. Im the end, it depends on your end users which notation is more suitable.

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

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

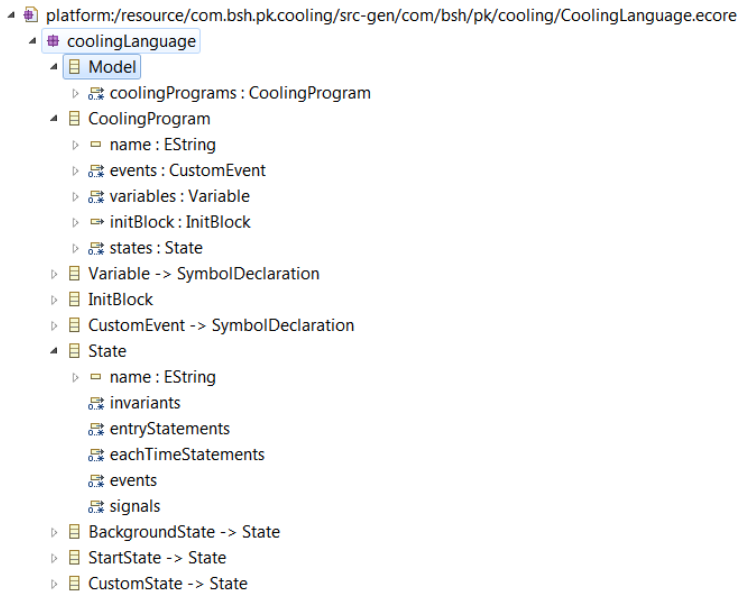
```
State:
    BackgroundState | StartState | CustomState;
```

The vertical bar operator is used here to express alternatives regarding the syntax. This is translated to inheritance in the meta model. The definition of custom state is shown in the following code snippet. It uses the same grammar language features as 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, Xtexts AS derivation algorithm pulls them up into the abstract State meta class so they can be accessed polymorphically. Fig. 1.5 shows the resulting meta model using EMF's tree view.



References Let us now look at statements and expressions. States have entry and exit statements, procedural code that is executed when a state is entered and left respectively. The `set v = 1` is an example. Statement itself is of course abstract and has all the various kinds of statements as subtypes/alternatives (in the actual language there are many more):

EV: maybe this should first show the grammar rule without the cross-reference, and then explain that's not how you normally do it in Xtext, but that you should make it a cross-reference instead. I suppose you could then even show it with `targetState=[State|QID]` and say that `targetState=[State]` is a short-hand in case it's an `ID`. that way maybe we can maintain a grammar perspective in this section

```
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 of the overall state-based program. It uses the keyword *state* and then a reference to the actual target state in the form of its name. Notice how Xtext uses brackets to express the fact that the `targetState` property points to an existing State as opposed to containing a new one (written as `targetState=State`), i.e.: a non-containing cross-reference. In traditional grammars, the rule would be written as `"state" targetStateName=ID;` and the language implementation would have to provide functionality to follow the `targetState` link separately.

Note that the cross-reference definition only specifies the target type of the cross-reference, but not the CS itself: by default, the ID terminal is used for the syntax. Initially, this ID ends up in a proxy URI which is resolved lazily as soon as we try and access the property. The Xtext framework takes care of most of this out-of-the-box, but you can exert a lot of control on its behavior by means of a custom scope provider.

There are two more details to explain. In case the actual terminal or datatype rule used to represent the reference is not an ID, but, for example, a name with dots in it (qualified name), this can be specified as part of the reference. Notice how 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 object, several objects with the same name might exist, or some target elements are not visible based on visibility rules of the language. To control the possible reference targets, scoping functions are used. These will be explained in the next section.

TODO: MB: I still have to reconcile the duplicate explanation of scoping in this paragraph...

Expressions The AssignmentStatement is one of the statements that uses expressions. This leads us to discussing the implementation of expressions in Xtext. Let us look at the definition of expressions as a whole. 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 Expression:
    MultiplicationLevel ((({Plus.left=current} "+") |
                         ({Minus.left=current} "-")) right=MultiplicationLevel)?;

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

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

AtomicLevel returns Expression:
    ({TrueExpr} "true") |
    ({FalseExpr} "false") |
    ({ParenExpr} "(" expr=Expr ")") |
    ({NumberLiteral} value=DECIMAL_NUMBER) |
    ({SymbolRef} symbol=[SymbolDeclaration|QID]);
```

We first have to explain in more detail how the 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 when the first assignment to one of its properties is performed. If no assignment is performed at all, no object is created. For example in case of `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";` Notice that the action can instantiate meta classes other than those that are derived from the rule name. 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.

Now we know enough to understand how expressions are encoded and parsed. 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:

- We start (by definition) with the Expr rule. The Expr rule just calls the ComparisonLevel rule. Note that a rule call can have two effects: it returns an object which represents the parsed result or null in case no object was instantiated, or it internally throws an exception indicating that the input couldn't be matched to the called rule. The returned object is available in the calling rule using the *current* keyword.

- The parser now "dives down" until it matches 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 actual 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, we unwind once more.

- We're now back at the MultiplicationLevel. The rule we have on the stack 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) to its left property. Then it assigns the newly created Multi to be the current object. At this point we have a subtree with the * at the root, and the NumerLiteral 2 in the left property.

- The rule hasn't ended yet. We dive down to PrefixOpLevel once more, matching the 3 in the same way as the two before. The NumerLiteral for 3 is assigned to the right property.

- 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: inside those, we can again embed an Expr, i.e. we jump back to the lowest precedence level.

TODO: Explain why Xtext does it this way (left recursion, etc.) and refer back to general parsing discussion.

To use Xtext with expressions it is not necessary to completely understand the mechanism, since expressions are always structured in the way shown above. The code shown here can be used as a template. Adding new operators or new levels can be done easily.

Polymorphic References In the cooling language, expressions also include references to other entities, such as configuration parameters, variables and hardware elements such as compressors and fans (defined in a different model, not shown above). All of these referencable 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 reference 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 symbol reference. A more natural design of the language would use different reference constructs for the different referencable elements. In this case, the reference itself is specific to the referenced meta class, making processing much easier.

```
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 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} "v:" var=[Variable]);
    ({ParameterRef} "p:" param=[Parameter]);
    ({HardwareBuildingBlockRef} "bb:" 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.

TODO: Explain how to go from AS to CS

1.6 Spoofax Example

Mobl's data modeling language provides entities and entity functions. To illustrate the language, here is an example of two data type definitions related to a shopping list app:

```
module shopping

entity Item {
  name     : String
  checked  : Bool
  favorite : Bool
  onlist   : Bool
  order    : Num
  store    : Store
}

entity Store {
  name  : String
  open  : Time
  close : Time

  function isOpen() {
    return open <= now.getTime() && close <= now().getTime();
  }
}
```

In mobl, most files starts with a module header, which can be followed by a list of entity type definitions. Our example *shopping* module defines two entities for items on a shopping list and stores for those items. Entities are persistent data types that are stored in a database and can be retrieved using mobl's querying API.

Syntax:

```
"entity" QId ":" Type "{" EntityBodyDecl* "}"  -> Definition {cons("Entity")}
"entity" QId "{" EntityBodyDecl* "}"  -> Definition {cons("EntityNoSuper")}
ID ":" Type "(" {Anno ","}* ")" -> EntityBodyDecl {cons("Property")}
ID ":" Type                     -> EntityBodyDecl {cons("PropertyNoAnnos")}
FunctionDef                               -> EntityBodyDecl
ID                              -> Anno {cons("SimpleAnno")}
"inverse" ":" ID                -> Anno {cons("InverseAnno")}


"function" QId "(" {FArg ","}* ")" ":" Type "{" Statement* "}" -> FunctionDef {cons("Function")}
"return" Exp ";"                            -> Statement
{cons("Return")}

Exp "." ID "(" { Exp  ")" -> Exp {cons("MethodCall")}
Exp "." ID                -> Exp {cons("FieldAccess")}
ID                        -> Exp {cons("Var")}
```

1.7 MPS Example

We start by defining a simple language for state machines. 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:

```
statemachine linefollower {
  event initialized;
  event bumped;
  event blocked;
  event unblocked;
  initial state initializing {
    initialized [true] -> running
  }
  state paused {
    entry int16 i = 1;
    unblocked [true] -> running
  }
  state running {
    blocked [true] -> paused
    bumped [true] -> crash
  }
  state crash {
    <<transitions>>
  }
}
```

TODO: Add UML diagram of the language. Generate via MPS.

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 Events. The alias is defined as "statemachine", so typing this word inside C modules instantiates a state machines. Statemachine also implements a couple of interfaces; IHasIdentifierName contributes a property name, IModuleContent makes the state machine embeddable in C Modules — the module owns a collection of IModuleContents, just like the Statemachine contains States and Events.

```
concept Statemachine extends MedBase
                     implements IHasIdentifierName
                                IModuleContent
  properties:
    << ... >>

  children:
    State states 0..n specializes: <none>
    Event events 0..n specializes: <none>

  references:
    << ... >>

  concept properties:
```

```
    alias = statemachine
```

The State contains collections of EntryActions and ExitActions. These both extend Action, which in turn owns a StatementList to include C code. StatementList is a concept defined by the C core language. To make that visible, our statemachine language extends C core. A state contains a boolean attribute initial, to mark the initial state. Finally, a State contains a collection of Transitions.

```
concept Transition extends MedBase
                   implements <none>
  properties:
    << ... >>

  children:
    Trigger trigger 1 specializes: <none>
    Expression guard 1 specializes: <none>

  references:
    State target 1 specializes: <none>

  concept properties:
    alias = transition
```

Transitions contain a Trigger, a guard condition and 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 of type Expression; another concept inherited from the C core language. A typesystem 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.

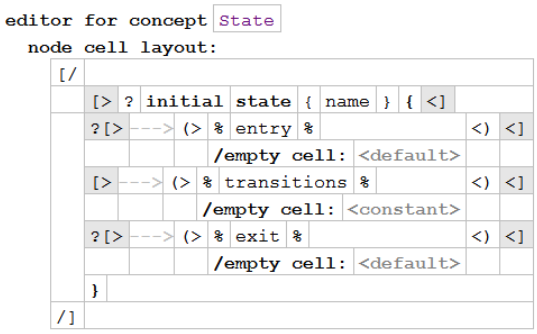
Editor Definition Editors are defined via projection rules. Editors are made of cells. When defining editors, various cell types are arranged so that the resulting syntax has the desired structure. Fig. 1.7 shows the definition of the editor for a transition. It arranges the trigger, guard and target state in a horizontal list of cells, the guard surrounded by brackets, and an arrow (->) in front of the target state.

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

TODO: Add a table projection to show off non textual notations

Fig. 1.7 shows the editor definition for the State. It uses a vertical list to arrange several horizontal elements. The first line contains the keyword initial, the keyword state and the name property of the state. The question mark in front of the initial label denotes this cell as being optional; an expression determines whether it is shown or not. In this case, the expression checks the initial property of the current state,

and shows the keyword only if is true. A quick fix is used to toggle the property.



The rest of the editor arranges the entry actions, transitions and exit actions in a vertical arrangement. Each of the collections is indented (—->). The entry and exit actions are only shown if the collections are not empty (another optional cell). A quick fix is used to create the first entry and exit actions, subsequent ones can be created by pressing enter at the end of an existing action.

Intentions Since the use of quick fixes (called intentions in MPS) is an essential part of editing with MPS, we show how they are created in this section. The following piece of code shows the definition on the one that triggers the initial properties. Intentions specify for which language concept they apply, a text that describes the intention in the intentions menu, an optional boolean expression that determines whether the intention is currently applicable, and the actual code that performs the quickfix. Note how this is simply a model-to-model transformation expressed with MPS APIs.

```
intention makeInitialState for concept State {

  description(editorContext, node)->string {
    "Statemachine: Make Initial";
  }

  <isApplicable = true>

  execute(editorContext, node)->void {
    foreach s in node.ancestor<concept = Statemachine>.states {
      s.initial = false;
    }
    node.initial = true;
  }
}
```

Expressions Since we inherit the guard expression structure and syntax from the C core language, we don't have to define expressions. It

is nonetheless interesting to look at its implementation.

Expressions are arranged into a hierarchy starting with the abstract concept Expression. All other kinds of expressions extend Expression, directly or indirectly. For example, the PlusExpression extends the 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 plus expression, they are a horizontal list of: editor for left argument, the plus 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 but putting the PlusExpression "on top" of the current node. Using the alias of expressions and the inheritance hierarchy, it is possible to factor all side transformations for all binary operations into one single action declaration, resulting in much less implementation effort.

```
side transform actions makeArithmeticExpression

  right transformed node: Expression tag: default_

  actions :
    add custom items  (output concept: PlusExpression)
      simple item
        matching text
          +
        description text
          <default>
        icon
          <default>
        type
          <default>
        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;
          }
```

In the AST, operator precedence is encoded by where in an expression tree a given expression is located. For example, in `2+3*4` the + is higher up in the tree than the *, encoding that the * has higher precedence. This is natural since processing of ASTs is typically depth-first, so the * is evaluated before the +. However, as we enter an expression linearly using the side transformations actions, we have to make sure

that we don't accidentally put a + "below" a *. There are two ways to deal with this:

- The first alternative allows lower-precedence operators below higher-precedence operators in the tree, but the editor automatically renders parentheses to make sure visual appearance is in line with the AST.
- The other alternative reshuffles the tree automatically whenever an expression is edited, putting higher-precedence operators further down in the tree. Parenthesis can still be used explicitly.

In both cases the system needs information about the precedence level of operators or expressions. This information can be stored in a concept property (comparable to a static member in a Java class).

Polymorphic References We have explained above how references work in principle: they are actual pointers to the references element (the necessary scopes are explained in the next section). 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 local variable (a) and an event parameter (timestamp) from within expressions:

EV: even more so than the previous bit on references, this part doesn't fit here and should be moved to the section on Scoping and Linking

TODO: make sure we explain that first

```
int a;
int b;

statemachine linefollower {
  event initialized(int timestamp);
  initial state initializing {
    initialized [now() - timestamp > 1000 && a > 3] -> running
  }
}
```

Both references to local variables and to event parameters use the same syntactic form: simply a name. In Xtext, this has to be implemented with a single reference (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 SymbolReferenece. Processing code has to inspect the type of the symbol to find out what a SymbolReference actually means.

In projectional editors this done differently. In the example above there is a LocalVariableReference and an EventParameterReference. Both have an editor that simply renders the name of the referenced element. Entering the reference happens by typing the name of the referenced element (cf the concept of smart references introduced above).

In the (rare) case where there's a LocalVariable and a EventParameter, the user has to make an explicit decision, at the time of entry (the name won't bind, and the CC menu requires a choice). It is important to understand that, although the names are similar, the tool still knows which one refers to a LocalVariable and which one refers to a EventParameterReference. Upon selection, the correct reference objects are created.

2
Scoping and Linking

As we have elaborated in the previous section, the concrete 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 "syntactical tree" to the "semantic graph", or how to establish the cross-links.

Often , a language's structure definition defines what concepts constitute valid target elements for references, but this is usually too imprecise: e.g., in the state machine example, only the States that also own the transition for which we want to define the target state are valid target elements.

MV: also for Spoofax?

There is a marked difference between the projectional and parser-based case:

- In projectional editors, cross-references are directly established as such. Since every model element has a unique ID, a cross-reference is simply a pointer to the target element's ID. The reference is established directly as the program is edited: the code completion menu shows potential target elements for a reference (typically in a pick list) and selection of one of them creates the link.
- In parser-based systems, the cross-references have to be established, or *resolved*, from the parsed text after the AST has been created. This means that one specific target element has to be chosen (unambiguously) from the potential target elements, based on the syntax for the cross-reference.

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. a namespace prefix, the location inside the larger structure

of the site of the cross-reference or something that's essentially non-structural in nature. The scope can also be used to provide content assist or code completion in the editor.

In both situations we need to be able to compute a useful, visual representation of the elements in the scope of a cross-reference. Typically, the name or non-technical ID is used as representation, although more complicated schemes (e.g., fully qualified names For projectional systems, this representation is used for the visualization of the cross-reference as well as for the items in the pick list. Parser-based systems require this representation to resolve cross-references by *matching* the syntax for the cross-reference with a representations of the elements in the scope. The computation and matching of those textual representations is often simply name-based but sometimes more advanced schemes are used.

As an example of such a scheme, consider how Java types can be referenced in source code either through a fully qualified name (e.g., `java.util.Map.Entry`) or through an unqualified or partly-qualified name (e.g., `Map.Entry`) in the presence of an *import* (in this case, `import java.util.Map;`). In this case, Java types are *globally exposed* through their fully qualified names, but *locally matched* taking *import* statements into account.

MV: Doesn't the following para relate more to the "visibility" view of the world?
EV: @MV I replaced occurrences relating to "visible". I also made the hierarchy more explicit again — using short(er) sentences :)

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 none of its elements can be matched to the cross-reference's syntax, then 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 feature reference consists of all the elements present in the directly-encompassing "block" while the outermost scope is the *global* scope. In parser-based systems, 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".

When designing or evolving a language, there are several trade-offs with respect to scopes:

- A scope with a large number of elements makes picking the right element harder for the user.
- A poorly thought-out scope might cause matching collisions, i.e.: two or more model elements which can't be readily distinguished from each other by the resolution mechanism. In that case, the mechanism either has to pick one at random or refuse to resolve the cross-reference.
- The notion of validity in "valid target elements" often doesn't con-

sider the language's constraints. This is often done because evaluating the constraints can be relatively time-consuming or even impossible to do reliably on incomplete syntax, but also because it's often more useful to have a meaningful error message coming from the constraints than just seeing an error message "could not resolve cross-reference".

Note that our notion of scope is inverted from the usual one which is centered around the *visibility* of elements from the perspective of other elements, where only visible elements can be referenced. Our notion is more convenient from the cross-reference viewpoint, however, as it centers around resolving particular cross-references one at a time.

The computation of the scope of a cross-reference typically involves navigating and querying the model. Being able to define the scope using a language that can express these things comfortably (and declaratively) is a definite plus for language workbenches. Higher order functions and chained expressions are very useful.

2.1 *Name resolution in Spoofax*

2.2 *Scoping in Xtext*

MV: Explain this one first, since this one is the most low level; good fit to introduce the basic mechanism

Xtext uses Java code (in the future probably Xtend2 code) for implementing all aspects of languages except the grammar. Language developers implement various classes to build the language beyond the grammar. Xtext framework and custom language implementatino classes are contributed to the runtime using Google Guice, a dependency injection framework. 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.

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. The method gets the EReference (which identifies the reference on the meta level) as well as the actual instance of the language concept whose reference should be scoped.

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

To make scoping implementation easier, Xtext provides so-called declarative scope provider through the AbstractDeclarativeScopeProvider base class: instead of having to inspect the EReference and context objects and decide how to compute the scope, the language implementor

can declare the computation of the scope for a specific reference in a separate 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 below 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. It was defined as

```
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 sub type of State) from any reference of a ChangeStateStatement. In fact, it includes children of the context type, so we could write an even more general alternative, which scopes the visible States from anywhere in a CoolingProgram, independent of the actual reference.

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

The implementation of the scopes is simple, and relatively similar in all three cases. We write Java code that crawls up the containment hierarchy until we arrive at a CoolingProgram (in the last alternative, we already have the CoolingProgram, so we don't need to move up the tree). 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());
}
```

Another feature of the declarative scope providers is that it comes built-in with the notion of a *global scope*. If no applicable methods

can be found in the declarative scope provider class (or if they return *null*), then scope computation is delegated to a IGlobalScopeProvider. The specific implementation is configured through a Guice binding. By default, the ImportNamespacesAwareGlobalScopeProvider is configured, 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. The language implementor can influence this configuration in the MWE2 workflow for the language or bind his own global scope provider implementations through Guice.

EV: I need to "fix" a later discussion of global case: see also the remark for the global scopes para below. Having a global scope by default really is a courtesy of the AbstractDeclarativeScopeProvider class.

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 name by which it will be referenced. So if a state is called normalCooling, then the ChangeStateStatement instance `state normalCooling` has to be used. The term normalCooling acts as the reference - pressing Ctrl-F3 on that program element will go to the referenced state.

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 scopes a different approach is recommended. Here is an example of a program expressed in a language with nested blocks and scopes:

```
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 variables (`sum`), the arguments (`x` and `y`) and the global variables 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 (very 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 typ-

ically hide symbols in outer blocks. The symbols in outer blocks are either not accessible, or a special name has to be used, for example, by prefixing them with the *outer* keyword.

Xtext's scopes support this scenario. IScopes can have outer scopes. If a symbol is not found in a scope, the scope queries its outer scope and tries to find the symbol there. Since inner scopes are searched first, this implements shadowing as expected.

Also, scopes are not just collections of elements. Instead, they are basically a map between a string and an element. 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 `x`. However, this name can also be changed. This enables the global `x` in the example above to be available under the name `outer.x` if it is referenced from location 1. The following is pseudo-code that implements this behaviour:

```
// 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 )
}
```

EV: I'm going to expand quite a bit on the following paragraph, because there's a notable difference between the standalone case and the IDE case. Also, the way that the global scope comes about and "sees" what elements are visible is a bit complicated. This is usually quite difficult for users to grasp.

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 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 the referencing files had to all be updated.

In the current release of Xtext both of these problems are solved with the index. The index is a data structure that stores (String,IEObjectDescription)-

pairs. An IEObjectDescription contains information about a model element, including its name, its URI (a kind of global pointer that also includes the file in which the element is stored) as well as arbitrary user data. All references are resolved against this index, not against the actual object. 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, the reference (aganst the index) is still valid. Also, the respective file is loaded only if and when the reference is resolved, improving scalability.

There are two ways to customize what gets stored in the index, and how. The IQualifiedNameProvider returns a qualified name for each program element. If it returns null, the element is not stored in the index, which means it is not referenceable. The other way is the IDefaultResourceDescriptionStrategy which allows language developers to build their own IEObjectDescription for program elements. This is important if custom user data has to be stored in the IEObjectDescription for later use during scoping.

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. We have shown this in the previous section. 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 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 CC menu by the language developer) of the targetted 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 extends MedBase
  // ...
  references:
    State target 1 specializes: <none>
```

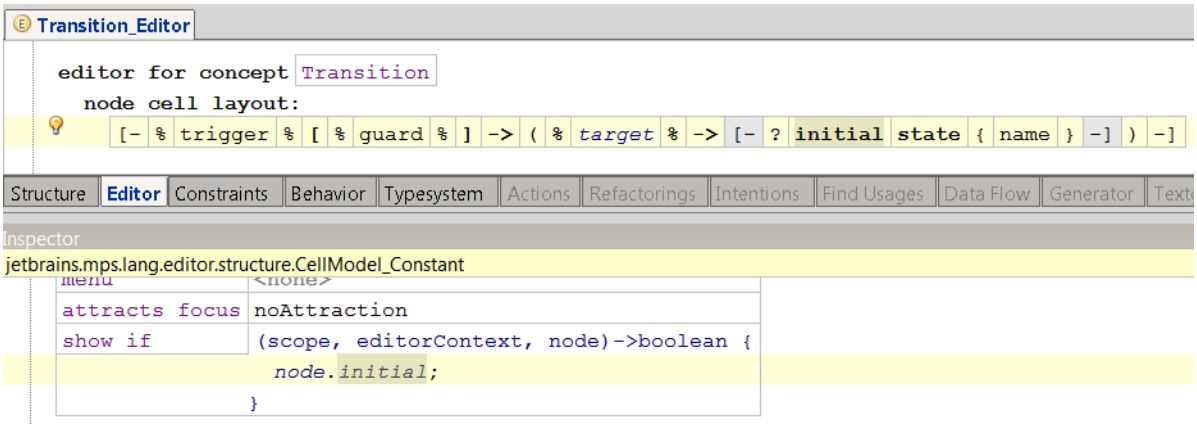
The scope itself is defined via the following search scope constraint. The system provides an anonymous function search scope that gets a

number of arguments that describe the context. The expression in the function then has to be written by the language developer. The expression crawls up the containment hierarchy until it finds a Statemachine. The set of valid target elements is the states of that state machine. The code used here can be arbitrarily complex, and is implemented in MPS' BaseLanguge.

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

In addition to the search scope, users can provide code that should be executed if a new reference target is set, additional validation, as well as customized presentation in the code completion menu.

In case of the transition, the projection of the transition uses the name of the target concept as the representation of the actual reference. This can be seen from the transition editor (see previous section), that contains this expression: `(%target% -> {name} )`. This means that the name property of the referenced State is shown as the reference. By default, this is also what is used in the code completion menu, resulting in text-editor-like behavior. Fig. 2.3 shows an example where the reference includes the "keyword" state, as well as the prefix initial in case the state is an initital state.



To make a point, the editor could also be defined as `(%target% -> X )`, where every reference is rendered as an X. The system would still work (because the underlying GUIDs are still used), but the information conveyed to the user has been reduced to zero.

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 references (such as blocks, functions or methods) implement an interface IScopeProvider.
- This interface provides a method getVisibleElements<concept C) that returns all elements 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 overshadowed from the result.

We have already discussed in the previous section that polymorpic references are not a problem in MPS. Each one simply defines its own search scope, possibly using the IScopeProvider-based approach introduced above: a LocalVariableReference returns all local variables as its scope, the ParameterReference returns all function parameters. In the resulting code completion menu in an expression where LocalVariableReference and ParameterReference are allowed, the superset of both scopes is shown.

3
Constraints

TODO: MPS dataflow

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

- unicity of names of various siblings;
- 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.

A particular kind of constraints are *type systems*: these verify the correctness of types in programs, e.g. making sure you don't assign a *float* to an *int*. Particularly in expression languages, type calculations and checking can become quite complicated and therefore warrant special support. This is why, we distinguish between simple constraints and type systems in this chapter, treating them in separate sections, even though there is essentially no conceptual difference.

TODO: Discuss diff approaches for checking constraints, and what are good languages for that?

3.1 Constraints in Xtext

Just like scopes, constraints are implemented in Java. Developers fill in check methods into the validator class generated by the Xtext project creation Wizard. In the end, these validations plug into the EMF validation framework, and other EMF EValidator implementations can be used in Xtext as well.

A check method is a Java method with the following characteristics: it is public, returns nothing, it can have 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):

```
@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 ChangeStateStatement ) {
            ChangeStateStatement css = (ChangeStateStatement) s;
            if ( css.getTargetState() == ctx ) return;
        }
    }
    error("no transition ever leads into this state",
     CoolingLanguagePackage.eINSTANCE.getState_Name());
}
```

The method retrieves the owning cooling program, then retrieves all of its descendants and iterates over them. If the descendant is a ChangeStateStatement, and if the targetState property of the ChangeStateStatement is 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 report an error. The CheckType.NORMAL in the annotation defines when this check should run:

- CheckType.NORMAL: run on save of the file
- CheckType.FAST: run after each keypress
- CheckType.EXPENSIVE: run only if requested via the context menu

The amount of code that needs to be written for constraint checks depends a lot on the language used to express the constraints. Just like scoping, constraints basically navigate and query the model. The more efficiently a language is able to do that, the more suitable it is. Java is not very suitable, because it lacks higher order functions and chain expressions. Subsequent versions of the Xtext validation API are expected to be based on Xtend2, a language that has these features. We'll discuss it in the section on code generation and transformation.

3.2 *Constraints in MPS*

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:

```
non type system 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;
    }
  }
}
```

Notice how the constraint is called a "non type system rule". This illustrates how in MPS type system rules are the default, and this "simple" constraint is an exception.

3.3 *Constraints in Spoofax*

4
Type Systems

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 values and then checks whether these types conform to predefined typing rules.

We distinguish between dynamic type systems which perform the type checks as the program executed, and static type systems, where type checks are performed ahead of execution, mostly based on type specifications in the program. This section focuses exclusively on static type checks.

4.1 Type System 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)
```

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

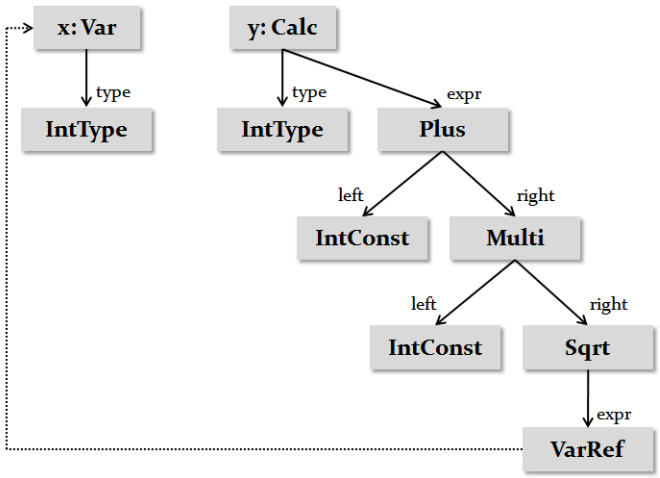


Figure 4.1: Abstract syntax tree for the above program. Boxes represent instances of language concepts, solid lines represent containment, dotted lines represent cross-references

and previously known. Examples include the IntConst (whose type is IntType), the sqrt concept (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 other 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, 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 (IntType can be used wherever DoubleType is expected). This subtype relationship has to be specified. 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". examples are the Plus and Multi constructs: if two IntTypes are added, the result is an IntType. If two DoubleTypes are added, the result is a DoubleType. If an IntType and a DoubleType are added, 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 language 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 not generally the same as it's language concept. For example, the concept (meta class) of the number 1 is IntConst and it's 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. Specifically, several instances of the same concept can have different types: a plus calculates it's type as the more general of the two arguments. So the type of each Plus instance depends on what kinds of arguments the particular instance has.

The core of a type system can be considered to be a function typeof that calculates the type for a program element:

$$typeof ::= concept => type$$

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 both cases, we 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`).

MV: Spoofax?

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. Wel'll explore each of these conceptually, and then provide examples in the tool sections.

MV: Eelco mentioned something about Data Flow stuff in Scala. Should we add this? Does anyone know something about that?

4.2.1 *Recursion*

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

> 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:

```
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 method typeof, which takes a program element and returns its type, while calling itself 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):

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

The following examples are structurally valid example instances:

```
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 cut 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 recusively calls typeof for its type property. Recursion ends with the typeof functions for the types; they return themselves.

However, while this implementation successfully calculates the type of the LocalVarDecl, 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). 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 that the type specification itself 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 )

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

This is relatively 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:

```
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 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 sophisticated, we have all used unification in high-school for solving sets of linear equations. The "equational theory" is algebra, and substitution refers to assignment of values to x and y. A solution is `x := 5, y := 10`.

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

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 be 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 equations. If an equation cannot be satisfied for any assigment 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 means type equation (left side must be the same type as right side), :<=: refers to subtype-equation (left side must be same type or supertype of right side). The operators are taken from MPS, which uses this technique.

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

Let us look at the four examples. 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.

```
var i: int          // 1

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 from the equations specified for the LocalVarDecl. So the first equation has no free type variables, but it is true with the type values specified. The second equation works the same as above, deriving T to be int.

```
var i: int = 42     // 2

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    // 3

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     // 4

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, incl. array initializers.

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

The additional complication in this case is that we need to make sure that all the initialiation expressions (inside the curlies) have the same or compatible types. Here are the typing equations:

```
type var T
foreach ( init.elements as e )
    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 all of these elements and T. This results in a set of equations that each must be satisfied. 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 $T := \mathit{double}$, but the equations would still work because we use the :<=: operator.

4.2.3 *Pattern Matching*

In pattern matching, we simply 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

4.3 *Xtext Example*

Up until version 1.0 Xtext provided no support for implementing type systems (beyond manually implementing constraints). In 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.

As a consequence, two third-party libraries have been developed: the Xtext Typesystem Framework (developed by Markus Voelter, `http://code.google.com/a/eclipselabs.org/p/xtext-t` and XTypes (developed by Lorenzo Bettini, `http://xtypes.sourceforge.net/`).
In the remainder of this section we will look at the Xtext Typesystem Framework.

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 framework has to be integrated manually:

```
@Inject
private ITypesystem ts;

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

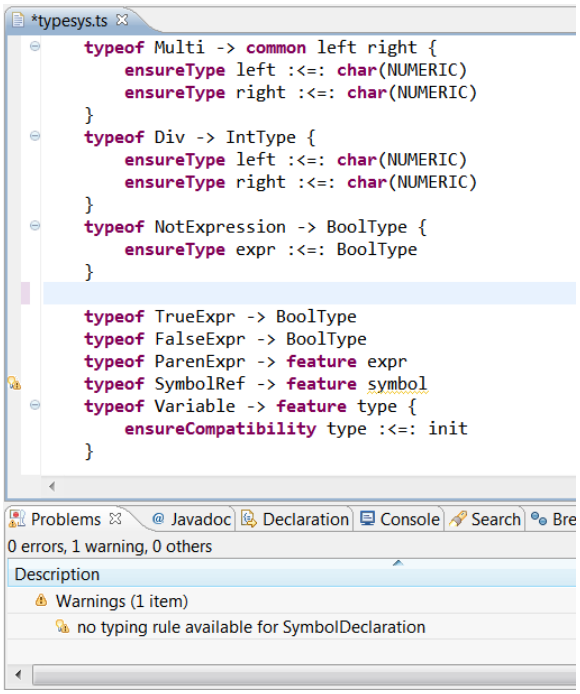
Most type systems are built from a limited set of typing rules (assigning fixed types, deriving the type of an element from one of its properties, calculating the type as the common type of its two arguments). The DefaultTypesystem class provides support for declaratively. In the code below, the initialize 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 CLTypesytemGenerated {

    private CoolingLanguagePackage cl = CoolingLanguagePackage.eINSTANCE;

    @Override
    protected void initialize() {
        try {
            useCloneAsType(cl.getIntType());
            ensureFeatureType(cl.getIfStatement(),
                cl.getIfStatement_Expr(), cl.getBoolType());
        } catch ( TypesystemConfigurationException tsce ) {
            tcse.printStackTrace();
        }
    }

    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 frame-

work; however, this is a nice example of how a DSL can provide added value over a framework or API (Fig. 4.3 shows a screenshot):

- the notation is much more concise
- code completion into the target language meta model is provided
- if the typing rules are incomple, a static error is shown in the editor, as opposed to getting runtime error when the framework starts up (see the warning in (Fig. 4.3)
- pressing ctrl-space on a property jumps to the typing rule that defines the type for that property.



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. 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. This is specified as follows:

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

For concepts that have a fixed type, but not 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 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 together. 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 booleanm that the left and right arguments must be COMPARABLE, and that the left and right arguments are 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.

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

There is also support for ordered compatibility, as can be seen from the typing rule for AssignmentStatement. 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, and *not* vice versa.

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

4.4 MPS Example

MPS includes a DSL for type system rule definition. The semantics are based on unification, and to some extent, pattern matching.

The type of a LocalVariableReference is calculated with the following typing rule[1]. 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.

[1] Since only the expression within the `do { ... }` block has to be typed by the developer, we'll only show that expression in future 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 rules. The first one makes sure that the notted 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. In MPS there are two different ways how language concepts can be instantiated. The first one (as shown in the first rule) 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.

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

A more interesting example is the typing of structs. Let us consider the following piece of 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: `typeof(parameter) :==: typeof(parameter.type);`. This is not specific to the fact that the parameter refers to a struct. The language concept that represents the Person type in the parameter is a StructType. A StructType refers to the StructDeclaration it represents a type of and extends Type, which acts as the super type for all types in the Embedded C language. So in essence, this means that the type of p

is an instance of StructType that has a pointer to the StructDeclaration Person.

The p.age is an instance of a StructAttributeReference. It is defined as follows. It is an expression, owns another expression variable (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
  instance can be root: false

  children:
  Expression variable 1 specializes: <none>

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

The typing rule for the StructAttributeReference looks as follows. The variable, 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.variable) :<=: new node<StructType>();
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<> varType = referenceNode.variable.type;
      if (varType.isInstanceOf(StructType)) {
        return (varType as StructType).struct.attributes;
      } else {
        return null;
      }
    }
```

MPS also uses pattern matching. 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.

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; // (4+7i)
```

The plus in `c1 + c2` should be the Plus defined by the original language. Alternatively, we could define a new plus for complex numbers. While this would work technically (remember there is no parser ambiguity problems), it would mean that users, when entering a Plus, 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.

However, this requires that the typing rules defined for plus in the original C language will now accept complex numbers; the original typing rules must be extended. To enable this, MPS supports so-called overloaded operations containers. The following container defines the type of plus and minus if both arguments are int or double.

```
overloaded operations rules binaryOperation

operation concepts: PlusExpression | MinusExpression
left operand type: <int> is exact: false use strong subtyping false
right operand type: <int> is exact: false use strong subtyping false
operation type: (operation, leftOperandType, rightOperandType)->node<> {
  <int>;
}

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

To tie these definitions into the regular typing rules, the following typing rule must be written[2]. Using the operation type construct, the typing rules ties in with overloaded operation containers.

[2] 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
  overrides false

  do {
    when concrete (typeof(be.left) as left) {
        when concrete (typeof(be.right) as right) {
            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> is exact: false use strong subtyping false
is applicable:
<no isApplicable>
operation type:
(operation, leftOperandType, rightOperandType)->node<> {
  <complex>;
}
-----------------------------------------------
```

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

4.5 Spoofax Example

Explain why expressions are different than "normal" programs (precedence, etc.)

Show how Xtext, MPS and SDF do it differently

Explain why type systems are important in this context, although they are "just constraints". Use as basis: TypesystemsForDSLs.pdf in the materials directory

explicit types vs. full type inference vs. local type inference

Recursion, Unification, Pattern Matching, Data Flow stuff (Scala, Eelco?)

attribute grammars

5
Transformation and Generation

Transformation of models is an essential step in working with models. 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) 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 transformed to artifacts expressed in a different language, in case of interpretation no such change 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: the actions performed as the tree is traversed were 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 langiage concepts as defined by the (dynamic) semantics of the concepts. A later chapter explores the notions of semantics in more detail.

We elaborate on the tradeoffs 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. In this context, a clear distinction is made between models and source code. Models are represented as an AST represented with some preferred AST formalism (or meta

model). The AST is represented as programming language data structures (typically objects) and an API exists for the programmer to interact with the AST. In contrast, the generated source code is treated only as text, i.e. a sequence of characters. The tool of choice for this kind of approach are template languages. The support the 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. Xtend2, the language used for code generation in Xtext is an example of this approach.

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). These languages also 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 Xtext 2 as well as QVT operational and ATL. 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 appraoch has the advantage that it supports transformation in both directions, and also supports model diff. QVT relational is an example of this approach.

As a point in case, please note that code generators and transformations fit our definition of interpreters: they traverse a model and perform actions: creating artifacts in a target language.

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 syntacic structure and the static semantics of the template language *and* the target language. Both MPS and Spoofax/SDF support this approach.

In MPS, program is actually projected and every editing operation directly modifies the AST, while using a 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 appraoch based on annotations . As we have

MV: Have we introduced annotations already?

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. The MPS example 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 sytnax of the target language to specify its structure. The Spoofax example will provide details.

5.2 *Code generation for incomplete languages*

An incomplete language is one where the complete semantics of the generated program cannot be expressed by the source model. Consequently, after a transformation/generation run, there are still "holes" in the generated code that need to be filled in by code directly written in the target language. Notice that this problem may occur in what's classically referred to as code generation and in transformation, although it is more often encountered in the former case.

To fill in the hole, two different approaches are possible. The first one involves invasively editing the generated artifact, adding in the missing functionality. Some mechanism must be provided to highlight areas in the generated code into which manually written code may be added, since only in these areas, manually written code is not overwritten during subsequent generation runs. The approach works with any target language, as long as the generator framework provides the necessary features described above.

The other alternative is to never modify the generated code at all, and instead use the composition mechanisms of the target language to "join" generated and manually written code. The most well-known case is captured in the generation gap pattern where a base class is generated from the model, and the manually written code is put into a manually written subclass.

In theory, with perfect tool support, the first approach is preferable

since no artificial composition relationships have to be introduced just to join the two kinds of code. In practice, with today's tooling, the second approach is usually more pragmatic. A particular advantage of the second approach is that the generated artifacts can always be thrown away and can be regenerated, since no changes are applied to them after generation.

TODO: When do we use M2M?

5.3 Xtext Example

5.3.1 Generator

5.3.2 Model-to-Model Transformation

5.4 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.

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 instances of the target language. So-called macros are used to express dependencies on 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 tranformation 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). The true node will be replaced by what the macro expression return.

Translating the State Machine State Machines are translated to the following lower level C entities:

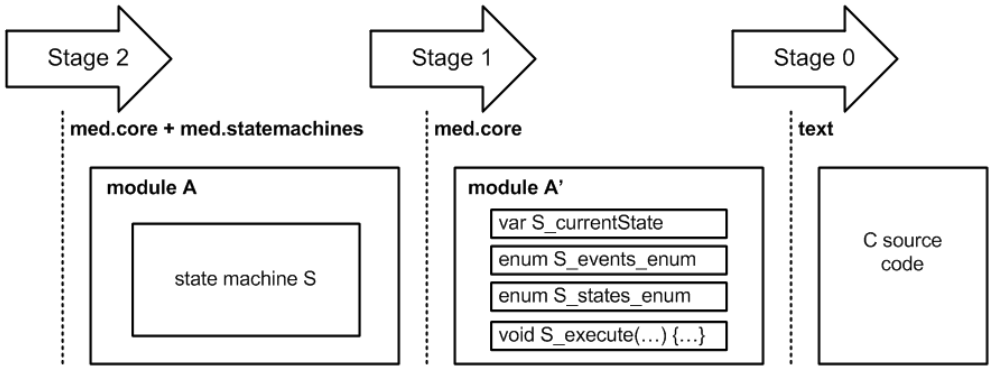
- a variable that keeps track of the current state

- an enum for the states, with a literal for each state
- an enum for the events, with a literal for each event
- and a function that implements the behaviour of the state machine using a switch/case statement. The function takes an event as an argument, checks whether the current state can handle the event, evaulates the guard, and if a transition fires, executes exit and entry actions and updates the current state.

This high level structure is clearly discernable from the main template in Fig. 5.4. It integrates into the overall translation process as follows.

```
content node:
module amodule imports <<imports>> {

  <TF currentStateVariable [ var statesenum $[currentState] = ->$[statesenum::stateLiteral]; ] TF>

  <TF [ enum $[eventsenum] { $LOOP$eventEnumLiternal[ $[eventLiteral] ] } ] TF>

  <TF [ enum $[statesenum] { $LOOP$stateEnumLiteral[ $[stateLiteral] ] } ] TF>

  <TF statemachineProcedure [ void $[statemachine]( eventsenum event ) {                                  ] TF>
                            [   $LOOP$[if ( currentState == statesenum::stateLiteral ) {                  ]
                            [           $LOOP$[if ( event == $COPY_SRC$[eventsenum::eventLiteral] ) {     ]
                            [                    if ( $COPY_SRC$[true] ) {                              ]
                            [                      $COPY_SRCL$[int i;]                                  ]
                            [                      currentState = ->$[statesenum::stateLiteral];        ]
                            [                      $COPY_SRCL$[int i;]                                  ]
                            [                      return;                                              ]
                            [                    }                                                      ]
                            [           }]                                                               ]
                            [ }                                                                          ]

}
```

The MPS transformation engine works in phases. Each phase transforms models expressed in some languages to other models expressed 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.



Revisiting Fig. 5.4, the picture becomes clearer: the various elements highlighted as template fragments (those enclosed by <TF TF>) are what the state machine is reduced to. State machines and variables, enum declarations and functions all live inside modules, so the replacement makes structural sense (an error would result otherwise).

References between model elements are based on identities and "real" pointers, not just similarity of names. So, in order to reference element that have been generated, they need to get unique labels. In Fig. 5.4 these are highlighted with an orange background. These so-called mapping labels have to be declared in the generator's mapping configuration. The give names to the relationship between a source model element and what has been generated from it.

```
mapping labels:
  label eventEnumLiternal : Event -> EnumLiteral
  label stateEnumLiteral : State -> EnumLiteral
  label statemachineProcedure : Statemachine -> Procedure
  label currentStateVariable : Statemachine -> ModuleVariable
```

For example, the label currentStateVariable can be ued the instance of ModuleVariable that was created from an instance of Statemachine. Attaching these mapping labels to the transformation rules (as shown in Fig. 5.4) establishe the relationship. An API call can be used to lookup the target; we'll show this later.

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. 5.4). The expression that determines what we iterate over is entered in the Inspector; Fig. 5.4 shows the code for iterating over the states.

Note that the only really interesting part of Fig. 5.4 is the body of the expression (node.states;), which is why in the future we will only show this part. For the literals of the events enum we use a similar

```
comment       : <none>
mapping label : stateEnumLiteral
mapped nodes  : (node, genContext, operationContext)->sequence<node< >> {
                  node.states;
                }
```

expression (node.events;). The $ macro is called a property macro. It is used to replace values of properties. In this case we use it to define the name property of the generated enum literal. Here is the implementation expression of the property macro (also entered in the inspector):

```
node.stateConstantName();
```

stateConstantName is a behavior method. It is defined as part of the behavior aspet of the State concept:

```
concept behavior State {

  constructor {
    <no statements>
  }

  public string stateConstantName() {
    return "STATE_" + this.name.toUpperCase();
  }

}
```

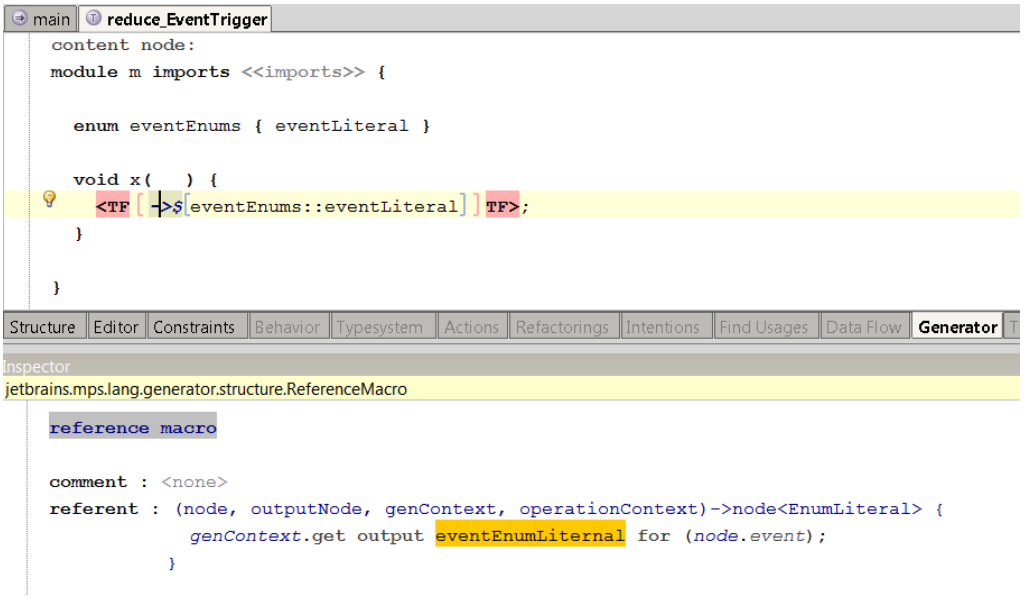
Let us next look at the part of the generator that generates the execute function. We first generate an if statement for each state that has at least one outgoing transition. We use a LOOP macro with the following expression:

```
node.states.where({~it => it.transitions.isNotEmpty; });
```

The if statement checks whether the current state corresponds to the state literal that corresponds to the state we're currently iterating over. Notice how we do not have to use a macro to rewire the eventsenum::eventLiteral to the actual literat for the corresponding state. Since we iterate over the same colletion (the list of states) of nodes both where we generate the literals and where we reference the literals, MPS automatically selects the correct reference target.

Inside the if, we now generate another if for each of the outgoing transitions of the state. We use a LOOP macro with the expression node.transitions. In the if, we compare the event that is passed into the method as an argument to the event enum literal that corresponds to the transition's trigger. Note that triggers are represented in the state

machine language using the abstract concept Trigger. A subtype, the EventTrigger, is used for those cases where the trigger is an incoming event (there may be other kinds of triggers in the future). To handle this polymorphic variability correctly, we use a COPY_SRC macro. It "copies" the trigger object, while also executing existing reduction rules. We have defined one of these that replaces the EventTrigger with a reference to the enum literal (see Fig. 5.4). Notice how in the inspector we use the generation context to resolve a label: we retrieve the enum literal that has been generated for the event.



We then generate the code that check the guard condition. We use a COPY_SRC macro to replace the dummy true with the guard expression in the model. Since we already use C expressions in the state machine, this really is just a copy process - no reduction rules are specified.

Now we have to deal with the exit action of the current state and the entry actions of the target state. These are lists of C statements, so we can simply COPY_SRC them into the generated code. Finally, we have to set the current state variable to the target state of the transition. We use another reference macro (->) to rewire this reference. The following code is used in the macro:

```
genContext.get output stateEnumLiteral for (node.target)
```

As the last example I want to show how the FireEventStatement is translated. It is used as follows:

```
initialize {
  ...
  event linefollower:initialized
}
```

It ha to be translated to a call to the generated statemachine execute method, with the event translated to the respective enum literal. A reduction rule is defined, so that every occurence of the FireEventStatement is translated to an ExpressionStatement that contains a call to the respective method. Fig. 5.4 shows the code.

```
module amodule imports <<imports>> {

  void statemachineExecute( eventsEnum e ) {
    << ... >>
  }

  enum eventsEnum { anEvent }

  void anotherOne(   ) {
    <TF [ ->$[statemachineExecute](->$[eventsEnum::anEvent]); ] TF>
  }

}
```

The goal is to generate a method call with an enum literal as the argument. However, we have to first create a module so we have a place to put the called method and the enum declaration. Only the code between the <TF TF> brackets (i.e., the method call) will be used as the result of the template. The rest is "scaffolding" so we have nodes we can reference on the example code that will then be marked up with macros. The first reference macro rewires the method. It uses the following expression to retrieve the execution function generated from the state machine. It uses the label associated with that function to retrieve the node from the context.

```
genContext.get output statemachineProcedure for (node.machine);
```

The second reference macro uses a similar approach to get to the correct enum literal.

```
genContext.get output eventEnumLiternal for (node.trigger);
```

discuss someehere the benefits of cascading and stuff. The MPS stuff is a good example of the benefits (platform indep of higher level DSLs)

5.5 *Spoofax Example*

special languages (target lang level + meta level) what are good gen languages (some criteria) integrating manually written code (generation gap) parts from My DSL Best Practices Paper

modifying generated code: not

generation gap pattern

extension of tatget lang to make it more easily generatable (expr blocks)

when to use M2M (see best practices paper) what are good trafo languages unidirectional, bidirectional (relational) incremental vs. batch) parts from My DSL Best Practices Paper

6
Building Interpreters

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 interpreter implementation looks depends a lot on the programming language used for implementing them. Also, the complexity of the interpreter directly reflects the complexity of the language it processes. For example, building an interpreter for a pure expression language with an object oriented or functional programming language is almost trivial. In constrast, the interpreter for languages that support paralellism can be much more challenging.

The following list explains some typical ingredients that go into building interpreters. It assumes a programming language that can polymorphically invoke functions or methods.

- For program elements that can be evaluated to values, i.e., expressions, there is typically a function eval that is polymorphically defined for the various expression types in the language. 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 the semantic action on the result. Consider an expression `3 * 2 + 5`. Since the plus is at the root of the AST, eval(Plus) would be called (by some outside entity). It is implemented as actually adding 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. 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.

- 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, typically called a StatementList. To execute those, there is typically a function execute that is overloaded for all of the different statement types. It is also overloaded for StatementList which iterates over all statements and calls execute for each one. 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 respecrive statement lists.

- Languages that have can express assignment to variables require an environment for execution. Consider `int a = 1; a = a + 1;`. In this example, the a in a+1 is a variable reference. When evaluating the 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 hashtable 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 assignable 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 mentionings of are references to the same variable, and both are expressions (otherwise a couldn't be used as part of the plus operator, which expects expressions as arguments). However, only a can be assigned to: writing `2 * a = a + 1;` 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 aren't, 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).

- The ability to call other entities (functions, procedures, methods) introduces further complexity, especially regarding parameter and return value passing. 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 associations. 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 recursive callable entities (as in `int fac(int i) { return i == 0 ? 1 : fac(i-1) };` requires that for each subsequent call to fac a new environment must be 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.

6.0.1 Building an Interpreter with Xtext

This example describes an interpreter for the cooling language. It is used to allow DSL users to "play" with the programs. The interpreter can execute test cases (and report success or failure) as well as simulate the program interactively.

The execution engine, as the interpreter is called here, has to handle the following language aspects:

- Statements and expressions, as described above, as supported by the DSL and must be executed
- 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 the following 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 behaviour for hardware elements. A set of constructs exists to express this mock behaviour, 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 kind of expression. However, Java doesn't have polymorphically overloaded member methods.

Instead, and Xtext workflow fragment generates a dispatcher. The fragment is configured with the abstract meta classes that represent expressions and statements (all specific expressions and statements inherit directly or indirectly from these classes). The following code shows the fragment configuration:

```
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 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 statement executor is similar to the expression evaluator.

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. 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 this code, it is worth mentioning that the "global data" held by the execution engine is stored and possibly around using the engine execution context. 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. Execution context is available through the eec() method.

```
    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;
    }
```

With this in mind, the implementation of even outnumber literal should be obvious. We first retrieve the actual value from the NumberLiteral object, and we find out that type of the number literal. The type system basically inspects, whether the value contains a dot or not and returns either a DoubleType or IntType. Based on this distinction the evaluate method returns either a Java Double or Integer as the value of the NumberLiteral. In addition, it creates log entries that document these decisions.

The evaluator for NumberLiteral was simple because number literals are leaves in the AST and have no other children, so no recursive invocations of eval are required. This is different for the logical and for example. The following code shows the implementation of the logical and. It has two more children in the left and right properties. the first two statements recursively called the evaluator, for the left and right children 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 = ((Boolean)evalCheckNullLog( expr.getRight(), log ))
                      .booleanValue();
    return leftVal && rightVal;
}
```

So far, we haven't used the environment, since we haven't worked with variables and their current values. Let's now look and how variable assignment is handled. We first look at the assignment statement, which is implemented in the statement executor, not in the expression evaluator.

```
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). 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 evaluator for a symbol reference (if it is used not as an lvalue) is shown in the following code. We use the same environment to lookup 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. Otherwise we return the value.

```
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 it. 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)* )?  ")" "{"
        (elements+=Element)*
    "}";

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.

```
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 the FunctionDeclaration uses a predefined utility method callAndReturnWithPositionalArgs. It accepts as arguments the list of formals of the called function, the list of actuals passed in, the list of statements in the function body, a symbol that should be used for the return value 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;
}
```

It first creates a new environment and pushes it on the call stack. The it iterates over all the actual argument, 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 statement executor (available through the context) to execute all the statements in the body of the function. Notice that if they deal with their own variables and functions, the use the new environment pushed onto the stack 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 where to find it. Finally, we pop the environment, restoring the caller's state of the world and return the return value.

States, Events and the Main program Changing a state happens 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) {
    l.child(Kind.debug, s, "change state to " + s.getTargetState().getName());
    engine.enterState(s.getTargetState(), log);
}

public void enterState(State ss, LogEntry logger )
        throws TestFailedException, InterpreterException, TestStoppedException {
    logger.child(Kind.info, ss, "entering state "+ss.getName());
    context.currentState = ss;
    executor.execute(ss.getEntryStatements(), logger);
    throw new NewStateEntered();
}
```

It 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 engine. 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 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 this in the next section. 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 terminated. Lastly, we process signal handlers (the check statements in the 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 deh: getCurrentState().getEvents()) {
        if ( reactsOn( deh, event ) ) {
            executor.execute(deh.getStatements(), logger);
        }
    }
}
```

Raising events is just another statement that can be put into any statement list. It adds the respective event to the queue. Also, events can be raised by external actors (the hardware in the real system, and statements in test cases).

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 interpreter for this statement is simply the following method:

```
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));
    }
}
```

TODO: mention interpreter framework!
TODO: Say something about Xtend2

6.0.2 *An interpreter in MPS*

Building an interpreter in MPS is fundamentally similar to building one in Xtext and EMF. All concept would apply in the same way, intead 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 language extensions can be defined in the first place.

For example, BaseLanguage (effectively Java) could be extended with support for polymorphic dispatch (similar to what Xtend2 does). An alternative solution involves a dispatch statement, a kind of pimped switch. Fig. 6.0.2 shows an example.

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.

figures/6/mps-dispatch.png

6.1 Testing DSLs

DSL testing is a multi-faceted problem. The following aspects of a DSL need to be tested:

- can the syntax cover all required sentences? Is the concrete syntax "correct"?
- 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?

TODO: Better argue, which kinds of tests are necessary: syntax, constraints/TS, semantics are in a natural order. Better check semantics than generated text. But this is an integration test. Maybe test intermediate results? That would be a structural test. Try to avoid it. Look at negative and positive testing in each of these cases. Maybe add additional lang constructs to "inspect" internal state. Allow this only in tests, not anywhere else (better than making everything public...). Testing semantics works only if executable. And target env is required! Is some kind of integration test. Maybe simulate? Mock? the whole simualtion story goes here as well. And interpreter. BSH.

TODO: I suggest we put the various tool examples into each of the following sections

6.1.1 Syntax Testing

Testing the syntax is simple. Developers simply try to write all relevant programs and see if they can be expressed with the language.

The following pieve of code is the fundamental code that needs to be written in Xtext to test a DSL program using the Xtext testing utilities at `http://code.google.com/a/eclipselabs.org/p/xtext-utils/wiki/Unit_Testing`. It is a JUnit4 test case with special support for the Xtext infrastructure.

```
@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) serializer is tested.

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:` is 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.

Note that these unit tests do not at all depend on the Eclipse UI or the Xtext editor, so they can be run from the command line or any other way of running JUnit tests. They only need a couple of Xtext libraries to be on the classpath. Consequently, it is simple to integrate these tests with continuous regression test infrastructures or a build server.

6.1.2 Constaints Testing

Especially for languages with complex constraints, such as those implied by type systems, testing of constraints is essential. 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. 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 support 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). 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)
  ...
```

If the test fails earlier in the filter expression, more than one output is provided:

```
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)
  ...
```

A similar facility is available for MPS, it is called a NodesTest (Fig. 6.1.2). It supports special annotations to express assertions on types and errors. For example, the third line of the nodes section reads `var double d3 = d` without annotations. This is a valid variable declaration in our C language. 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 message.

In addition to using these annotations to check models, developers can also write more detailed test cases. In the example I 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.

TODO: Testing Scopes

```
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;
  }
```

6.1.3 *Semantics Testing*

There are fundamentally two different ways of how a transformation or a generator can be tested. In both cases, example programs are transformed, and we expect a valid (syntax and constraints) target program. The first alternative then involves asserting the correctness of the structure of the generated program by writing additional, source-program specific constraints. Alternative two involves running the generated program and writing tests against it.

The second alternative is generally preferrable since it does not test syntactic structures of the target program, but instead really tests the semantics. Ideally, the test themselves are expressed as part of the DSL program (or a special fragment, built specifically for testing). However, the approach works only if the to-be-tested DSL expresses behavior and the target program can be executed. If the target is an intermediate representation that needs to be further transformed for execution, this kind of test of course tests the whole stack and not the single transformation. However, for testing code generators of behavioural languages, this approach is the best way to go. Interpreters are also typically tested this way.

The first alternative should be used in the remaining cases, i.e. for languages that express structures only (and hence programs cannot be executed and unit-tested), or for white-box testing of several chained transformations.

TODO: can we provide an example for the latter?

Testing an Interpreter with Xtext

The cooling language provides a way to express test cases for the cooling programs within the cooling language itself. These tests are executed with an interpreter inside the IDE, and they can also be executed on the level of the C program. To make this work, the interpreter, as well as the code generator, can handle the cooling programs as well as the tests.

Testing DSL programs by running tests expressed in the same language runs the risk of doubly-negating errors. If the DSL program is

wrong, and the test is wrong in a compatible way, the error will not be found. Also, this approach does 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. However, this implicitly also tests the interpreter (and generator), since, unless the interpreter works "as expected", the expectations expressed as assertions in the test case will fail.

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 (the language is stepped) in line 3. At this point nothing should happen, since no even was triggered (we don't test this). 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 (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());
    }
}
```

The runAllTestsInFile method is called, passing in the model's root element. The method 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.

```
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);
    }
}
```

Testing generators in this way works similarly. The generator generates the DSL program to an executable representation, for example, Java or C code. The tests are generated to unit test code in the same language. Running the tests simply involves executing the program and the test code together.

The following is a test case expressed using the testing extension to C. It contributes test cases to modules. The first one, testSimpleAdding, simply adds two numbers. It then asserts that the adding algorithm works. The second test, testPhysicalQuantities, tests whether the encoding of physical quantities work correctly. Physical quantitites are another extension to C that supports the definition of special types (speed in the example). By specifying a min and max value, the optimal resolution of the value range to an underlying 16 bit integer is used. The test assigns 10 km/h to the variable s1 using the p<..> notation. In the assertion, we then assert that the actual value is 2180, which is 10 times 2180, the resolution factor derived from mapping 300 km/h units to 65535 possible int16 values. Finally, the test contains an initialization statement that executes the tests. This statement is specific to the Osek target platform.

```
module tests imports <<imports>> {

  test testSimpleAdding / tests adding two numbers {
    int16 i = 0;
    int16 j = i + 1;
    assert j == 1  =>  adding failed
  }

  quantity speed range 0 .. 300 unit km/h

  test testPhysicalQuantities / tests the encoding of physical quantities {
    speed s1 = p<speed:10>;
    assert s1 == i<2180>  =>  speed encoding failed
  }

  initialize {
    run test testSimpleAdding;
    run test testPhysicalQuantities;
  }

}
```

From this program the following low-level C is generated (plus a makefile and an Osek configuration file). Notice how the generator takes into account the Osek platform specifics: for example, it uses ecrobot_stats_monitor to output messages to the screen of the lego robot.

```
#include "include/Tests.h"

// used resources
#include "ecrobot_interface.h"

// custom includes
#include "kernel.h"
#include "kernel_id.h"
#include "stdint.h"
#include "stdint.h"

void Tests_tests_testproc_testSimpleAdding(void) {
    ecrobot_status_monitor ("running test: testSimpleAdding" );
    int16_t i = 0;
    int16_t j = (i + 1);
    if ( j != 1 ) {
          ecrobot_status_monitor ("  FAILED: adding failed" );
    } // end if
}

void Tests_tests_testproc_testPhysicalQuantities(void) {
    ecrobot_status_monitor ("running test: testPhysicalQuantities" );
    int s1 = (10 * 218);
    if ( 2180 != s1 ) {
          ecrobot_status_monitor ("  FAILED: speed encoding failed" );
    } // end if
}

void ecrobot_device_initialize(){
    Tests_tests_testproc_testSimpleAdding ( );
    Tests_tests_testproc_testPhysicalQuantities ( );
}
```

While the above examples are trivial test cases, they nontheless illustrate the benefit of testing semantics vs. testing syntax. Inspecting the generated C code for syntactic correctness, based on the input program, would be much more work. Also, in case the DSL has several different generators to achieve a degree of platform independence, the tests automatically benefit from this platform independence as well. This is specifically true since in case of the unit testing extension, its generator output MPS-C, which is subsequently transformed into one of several versions of platform specific C code. The unit testing extension gets this platform independency "for free".

6.1.4 Testing for language completeness

6.1.5 Testing Editor Services

Testing whether the DSL can express what it should - review, example code... (short discussion, relate to process and interaction with domain experts)

These programs are made persistent and the tests can be rerun ar any time, making sure the grammar doesn't stop handling existing programs.

Testing the syntax: just write examples and see....

Coverage problematic, since unlimited number of programs

Test AUtomation /Nightly

Testing whether all constraints work: tests (e.g. using the type system f/w)

Testing whether the semantics are correct express "tests" in the same (or related) DSL

Then interpret or generate both and see if they fit ... you can easily automate that!

If a DSL only describes structure: Constraints, and "xpath" on the genreated stuff

6.2 Semantics

use BSH SM example as an example of testing the semantics (intp and gen, systematische Tests for Event Handling)

explain how this section is more than (systematic) testing. (see if we can say sth sensible here)

Do we need this here? We have it in the Dimensions paper.

Semantics is "what it all means" Use statemachines as an example of a notation that may have many semantics (Ptolemy) Approaches to defining semantics

Translators (denotational semantics) - Typically used: map to something whose semantics you (think you) know

Interpreters (operational semantics) - interpreters as model of semantics - see WebDSL semantics

Reasoning about semantics - Problem: How do you know the mapping is correct

you define it away: what the mapping does it what it means, by definition! What do you do if you have several mappings to different targets? How do you make sure these do the same thing? Pragmatics: Textual docs and (high coverage) testing computational models: temproal data, reactive (mobl) interpreter + generator must be the same proofing vs. testing "Interpreter is easier to understand and reason about" model checking!

6.3 Debugging DSLs

debugging the DSL definition (use the debugger of your favourite LWB)

debugging the models: only makes sense for models with behavior state machine debuggers exist not a lot of support for "getting a debugger for free" from the language definition problem: debugging on several levels (DSL, intermediate, implemetation), error traceability through the levels MPS as an example?

6.4 Evolution

See my best practices paper.

6.5 Editor Services

Not here, but at the respective place in the other chapters: * Code Completion * Go To Definition * Find Referneces * Pretty Priting

here: * Simulation (as in Cooling!) * Quick Fixes * Outlines * Syntax coloring * Refactoring * what else?