The Rascal Language Workbench

Tijs van der Storm
CWI, Amsterdam
storm@cwi.nl

May 3, 2011

Contents

Chapter 1

Introduction

1.1 Rascal: Meta Programming Made Easy

Rascal is a programming language for source code analysis and transformation. This means that typically the input of a Rascal program is a program in some programming language, and the output is often yet another program. So Rascal is a *meta programming language*. Source code is thus primary object of manipulation in Rascal.

Many of the use cases that Rascal is designed to address, follow the Extract-Analyze-SYnthesize, or *EASY* paradigm (shown in Figure 1.1). Meta programs often start by extracting information (facts) from the input program. This is the extraction phase. An example could be the call-graph of a program. Then, this extracted information is often subject to analysis: derived facts are computed, the information is enriched. For the call graph, a simple analysis is determining the root or leaf routines in the a source program by analysing the extracted call-graph. Another analysis could be concerned by identifying routines that are never called (dead code). Finally, the meta program will synthesize some kind of result. This can be transformed source code (e.g., removal of dead code from the input program), a report (e.g., statistics on the number of root and leaf routines), or a visualization (e.g., a graphical depiction of the call-graph). Of course, these phases are not strictly sequential: there may be feedback loops. Some analysis leads to new extraction, synthesis of a result may lead to new analyses and so on. Rascal has elaborated features to support each of the phases of the EASY paradigm fully integrated in the language.

Naturally, the implementation of domain specific languages (DSLs), or more generally, model-driven engineering (MDE) fits the EASY paradigm very well. When implementing a DSL compiler or interpreter the input is, of course, DSL source code. *Extraction* could, for instance, include the derivation of an AST from the concrete syntax tree. Another extracted model could be a graph-like structure representing the input in a more abstract way, or a performance model. Such abstractions are input to *analyses* such as constraint checking or type checking, verification, quality-of-service analysis etc. Finally, *synthesis* covers tasks such as graphical visualization, code generation, and optimization. To conclude, in the context of Rascal, we see DSL implementation as an instance of source code analysis and transformation.

1.1.1 Highlights of the Rascal Language

Design Rationale

Rascal is new and extensive language. Below we discuss the principles that have guided its design.

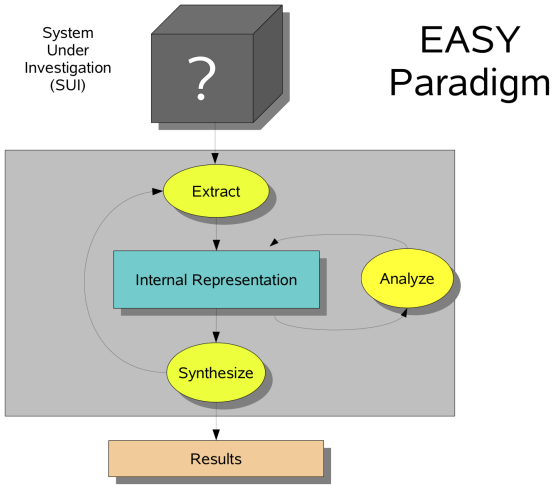

Figure 1.1: The Extract-Analyze-SYnthesize (EASY) paradigm

Scaling up and down Rascal is designed to support large, complicated tasks that require state-of-the-art techniques and algorithms. Analyzing and transforming millions of lines of COBOL legacy code is such a task: it requires powerful parsing technology to handle the complicated grammar(s) of (many dialects of) COBOL and the ability to handle enormous volumes of source code. On other hand, however,—and this is a fact that is sometimes forgotten—it should also be possible to do simple things in a simple way. For instance, source code analysis tasks typically executed using `grep` should also be easy to express in Rascal. Another example is string-based code generation: this does not give you syntax-safety, but for many cases it is more than sufficient and does not require a grammar of the target language. Rascal thus tries to strike a balance in features for making complicated things possible, while keeping simple things simple.

Expressivity without magic Instead of doing a lot for you automatically, heuristically or behind the scenes, Rascal provides you with powerful constructs to allow you to *program* what you want. This is important in order to let users make precision/performance trade-offs. There's no straight-jacket. If you want you can program your analyses and transformations just using while loops and if statements in order to get the best performance possible. In many cases, however, performance requirements are not that strict, and for those situations Rascal provides numerous built-in facilities, for instance, to parse source text (context-free grammars), to traverse tree structured data (the `visit` statement), to analyse relational data (comprehensions, transitive closure) or to generate code (string templates). A related principle is: what you see is what you get (WYSIWYG). For instance, the basic control flow of a Rascal meta program can be followed from the source code itself; there is no magic. Another example of WYSIWYG is: *all* data can printed, or written on and read from file, and deconstructed (pattern matching) in a uniform fashion. This is true for all data, including syntax values (parsed source code). To summarize, Rascal aims to put user in control of any magic she requires.

1.1.2 Rascal at a Glance

In the previous subsection we presented a high-level overview of the design principles of Rascal. To make things a little more concrete, we give a quick overview of the most distinguishing language features of Rascal.

Familiar syntax and control flow Although RASCAL is a meta programming language, its syntax will appear familiar to anyone who is acquainted with C, Java, JavaScript or C#. Most of the usual suspects (while, if, block, assignment, declaration) obey the same syntactic (and often semantic) rules of curly-based general purpose languages.

Immutable data RASCAL can be considered an imperative language: it has side-effects in variables, while-loops, if-statements etc. There's one big difference, however: all data is immutable. As a result, a changing a value (e.g., adding an element to a set) will always produce a new value. Even though you'll sometimes see statements that seem to destructively update data, new values are constructed internally. The only side-effects in RASCAL happen in variables.

Built-in data types and pattern matching RASCAL features a large collection of built-in data types: integer, boolean, string, real, tuple, list, set, relation, map, parse trees, source locations (for identifying fragments of source code), date-time. Additionally, RASCAL supports user-defined algebraic data types (ADTs). All data types have a syntactic, literal representation. Furthermore, all data[1] can be used in pattern matching, a concept well-known from functional programming. *Concrete syntax* patterns can be used pattern matching parse trees. That is, you can use the syntax of the object language (the source code you're processing) in order to do pattern matching.

Domain specific constructs Although RASCAL has most of the features of a general purpose language, it also features a number of domain specific constructs. One example is the `visit` statement for structure-shy traversal and transformation of arbitrary data, including parse trees. For any task in the source code analysis and transformation space this feature is indispensable. Other domain specific constructs include: comprehensions (for querying and creating sets, relations, maps or lists), regular expressions (for string matching), transitive closure (on binary relations), and a `solve` statement (for fix-point computations).

Arbitrary context-free grammars Context-free grammars are important for any source code analysis and transformation task, including the implementation of DSLs. Hence, they are an integral part of RASCAL. Context-free grammars define types of parse trees, similar to the way ADTs may be used to type abstract syntax trees. Such values can be manipulated in the same way as other values. RASCAL automatically generates parsers for these grammars using state-of-the-art scannerless, generalized parsing technology. Since RASCAL supports arbitrary context-free grammars, syntax definitions can be decomposed in separate modules and be extended at will.

String templates Since code generation is such a common task in the domain of source code analysis and transformation, RASCAL has built-in support for string templates. String templates are just like ordinary string values, except they may contain holes to interpolate data. Simple interpolations consist of simply injecting the string representation of a value into a string. More complex interpolations can be constructed using embedded if, while and for statements. Finally, the evaluation of string templates maintains indentation with respect to user defined margins. This facilitates the generation of well-indented code.

Integration with Java Sometimes a certain task requires a feature that is not provided by RASCAL or its standard library, or is better implemented in a general purpose language (for instance for performance reasons, or the availability of third-party library). To cater for such situation, it is possible to interface to the Java world simply by annotating a RASCAL function

[1] At the time of writing, the only exception is maps.

header with the qualified name of a Java method. All built in data types of Rascal are available from within Java to serialize foreign data back into Rascal.

IDE integration with Eclipse Programming without IDE support is hard to imagine these days. For this reason, Rascal can be installed as an Eclipse-based IDE, featuring syntax highlighting, outlining, interactive visualization, and a Read-Eval-Print-Loop (REPL). Additionally, you can dynamically extend this IDE for your own languages. You may register parsers, outliners, hyperlinkers, type checkers etc.—all developed in Rascal—to get IDE support for your own languages without having to recompile and/or restart Eclipse.

1.2 About this Document

This document has a dual purpose. First it gives a tutorial overview of how Rascal can be used to implement DSLs. Second, it presents the documentation for the Rascal entry in the *Language Workbench Competition 2011 (LWC'11)* . In this competition all contestants implement the same DSL (either graphically or textually) in order to evaluate and compare the merits of various language workbench approaches.

The document proceeds by discussing each of the tasks defined by the language workbench challenge. All Rascal source code included in this document is hyper linked to the actual sources in the public Subversion repository of Rascal.

To download and install Rascal, please visit the following website and follow the installation instructions:

- `http://www.rascal-mpl.org/Rascal/EclipseUpdate`

The complete Rascal solution to the LWC'11 challenge can be found at:

- `http://svn.rascal-mpl.org/lwc/trunk/lwc11`

The revision used to prepare this document is 35063 . After installing the Rascal plug-in you may check out this URL as a new project in order to run and explore the submission. Click the "Rascal" menu to start the interactive Rascal documentation system

With respect to the LWC'11 challenge this document is structured as follows:

- Phase 0 (Basics): Chapter 2: Basic Entity Modeling
  - Task 0.1 Section 2.1: Syntax Definition
  - Task 0.2 Section 2.2: Code Generation
  - Task 0.3 Section 2.3: Constraint Checking
  - Task 0.4 Section 2.5: Breaking Down Models
- Phase 1 (Advanced): Chapter 3: Extended Entity Modeling
  - Task 1.1 Section 3.1: Instances
  - Task 1.2 Section 3.2: Type Checking Instances
  - Task 1.3 Section 3.3: Model-to-Model Transformation
  - Task 1.4 Section 3.4: Name spaces
  - Task 1.5 Section 3.5: Integrating Manually Written Code
  - Task 1.6 Section 3.6: Multiple Generators

The full assignment of LWC'11 can be found at:

- `http://www.languageworkbenches.net/LWCTask-1.0.pdf`

Chapter 2

Basic Entity Modeling

2.1 Syntax Definition

2.1.1 Concrete Syntax: Context-Free Grammars

The first step in DSL implementation is defining its syntax. In this section we will define a textual syntax for the entities language. Syntax in RASCAL is defined using context-free grammars. Since RASCAL is backed by generalized parsing technology, you can define arbitrary context-free grammars, without having to restrict your grammar to classes such as LR(k) or LL(k). An important consequence is that grammars can be developed in a modular fashion. We will later see that this is important for extending and combining languages.

Listing 1 shows the main part of the entities syntax definition. First, auxiliary (syntax) modules are extended for defining Layout (white space and comments), Identifiers and Types. Module extension differs from module importing, in the sense that all definitions (functions, productions, data types etc.) are made local to the extending module. An entities file then consists of a sequence of zero or more `Entity`s. An `Entity` starts with the keyword `entity`, followed by a name and a sequence of zero or more `Field`s. Finally, a `Field` consists of a Type (integer, string, boolean or Entity reference) and a name.

Note that the `Entities` non-terminal is defined using `start syntax`: this indicates that it is the top-level syntactic element we will use for parsing files. Note further that productions and symbols in a production are labeled with a name. Production labels are identifiers at the start of a production followed by a colon. The symbols in productions are labeled by a post-fix identifiers.

One final note concerns the `@Foldable` annotation of the `Entity` production. This is used to support folding in editors for this language.

Listing 1 Syntax definition of entities

```
extend lang::entities::syntax::Layout;
extend lang::entities::syntax::Ident;
extend lang::entities::syntax::Types;
start syntax Entities
    = entities: Entity* entities;
syntax Entity
    = @Foldable entity: "entity" Name name "{" Field* "}";
syntax Field
    = field: Type Ident name;
```

Now that we have defined the syntax of our language, how do we use it? In RASCAL, if you

import the standard library `ParseTree` module in addition to all the relevant syntax modules, you get a `parse` function for free. In the RASCAL's REPL[1], we could use our language as follows:

```
> import lang::entities::syntax::Entities;
> import ParseTree;
> pt = parse(#Entities, "entity Person { string name integer age }");
ok
Entities: ‘entity Person { string name integer age }‘
//...
```

The function `parse` receives a reified[2] type argument indicating how the second string argument should be parsed. The result consists of a concrete parse tree, shown after the literal rendering between ‘ (back-ticks). This parse tree value contains *all* information of the source, including white space, comments, keywords, and source locations. Since most of this information is irrelevant to further processing of entities, we will define *abstract* syntax in the next section. Converting a parse tree to an abstract syntax tree (AST) discards irrelevant detail, leaving only essential structure.

2.1.2 Abstract Syntax: Algebraic Data Types

RASCAL is a functional programming language. Like many other functional languages (e.g., ML, Haskell etc.), RASCAL supports algebraic data types (ADTs). ADTs are a perfect vehicle for defining AST types. The complete ADT for defining entity ASTs is shown in Listing 2.

Listing 2 Abstract syntax of entities

```
data Entities = entities(list[Entity] entities);

data Entity = entity(Name name, list[Field] fields);

data Field = field(Type \type, str name);
data Type = primitive(PrimitiveType primitive)
          | reference(Name name);

data Name = name(str name);
data PrimitiveType = string() | date() | integer() | boolean() | currency();
```

ADTs have a similar structure to grammars, except are introduced using the `data` keyword. Each data type has one or more constructors for building values. In this case, for every syntax production in the grammar for entities, there is a corresponding constructor in this ADT. Every constructor has the same number of arguments as the number of symbols in the production (modulo keywords). Lexical tokens correspond to RASCAL primitive types. For instance, the `Ident` symbol in the production for `Field` (cf. Listing 1) is a lexical token, and corresponds to the `str name` argument of the `field` constructor.

Parse trees can be converted to ASTs using the standard library function `implode`. For instance, taking the `pt` parse tree above as starting point:

```
> import lang::entities::ast::Entities;
> ast = implode(#lang::entities::ast::Entities, pt);
ok
Entities: entities([entity(name("Person"),
  [field(primitive(string()), "name"), field(primitive(integer()), "age")])])
```

[1] Read-Eval-Print-Loop

[2] Reified types are value representation of RASCAL types. This kind of type reflection makes it possible to make type-safe (de-)serialization functions in RASCAL.

Again, `implode` gets a reified type to determine what kind of value has be created. It expects a correspondence of ADT type names to grammar non-terminals, and constructor names to production labels [3]. Based on that correspondence it produces an AST as shown[4].

In most language processing tasks, the AST will be the primary data. In the sections below we will assume that source text has been parsed and converted to an AST.

2.2 Code Generation

In the previous section we have defined a concrete and abstract syntax for entities. In this section, our aim is to generate GPL Code from entity models. We will define a generator that produces Java source code from entity ASTs. Listing 3 shows an excerpt of the code.

Listing 3 Functions to generate Java classes

```
public str entity2java(Entity e) {
    return "public␣class␣<e.name.name>␣{
           '<for (f <- e.fields) {>
           '␣␣␣␣<field2java(f)>
           '<}>
           '}";
}
public str field2java(field(typ, n)) {
    <t, cn> = <type2java(typ), capitalize(n)>;
    return "private␣<t>␣<n>;
           'public␣<t>␣get<cn>()␣{
           '␣␣␣␣return␣this.<n>;
           '}
           'public␣void␣set<cn>(<t>␣<n>)␣{
           '␣␣␣␣this.<n>␣=␣<n>;
           '}";
}
```

The code generator is defined using ordinary Rascal functions that produce string values. However, both functions use Rascal's powerful string templates to produce Java code. Although this code is relatively self-explanatory, let's take a closer look at `entity2java`.

The string value returned by `entity2java` uses string interpolation in two ways. First, the name of the `Entity e` is directly put into the string via the interpolated expression `e.name.name` between < and >. Next the body of the class is produced using an interpolated for-loop. This for loop evaluates its body (a string template again) and concatenates the result of each iteration. For each field, the function `field2java` is called to generate a field, and getter and setter declarations. The single quote (') acts as margin: all white space to the left is discarded. Furthermore, every interpolated value is indented automatically relative to this margin. As a result the, output of each consecutive call to `field2java` is nicely indented in the class definition.

The function `field2java` shows a relative particular feature of the Rascal language: pattern-based dispatch. The signatures of Rascal function do not have to consist of a list of typed parameters (as in, e.g., C or Java), but may consist of arbitrary patterns. As a result, such functions act like a certain multi-methods, which are dispatched on the basis of the constructor type of the actual parameters to it. The `field2java` function thus matches its first parameter against the pattern `field(typ, n)`. Later on, we will show that pattern-based dispatch is a powerful tool for modularly extending language implementations.

[3] This is the reason the AST type `Entities` has to be qualified: its name is the same as the `Entities` start non-terminal.

[4] The actual AST is annotated with source location values in order to be able to link back to the original source code (e.g., for error marking).

2.3 Constraint Checking

Constraint checking on the entity language consists of checking for duplicate entity definitions, duplicate fields in an entity definition and undefined entity references in fields. Just like the Java code generator of the previous section, these checks are implemented using ordinary Rascal functions. This functions consume ASTs and produce a list of `Messages`,—a data type for error messages provided by the Rascal standard library. The functions for checking entities are shown in Listing 4.

Listing 4 Functions to check entities for errors

```
public list[Message] check(Entities es) {
    defs = {};
    errors = for (e <- es.entities) {
        if (e.name in defs)
            append error("Redeclared␣entity", e.name@location);
        defs += {e.name};
    }
    return ( errors | it + checkEntity(e, defs) | e <- es.entities );
}
public list[Message] checkEntity(Entity e, set[Name] defs) {
    fs = {};
    return for (f <- e.fields) {
        if (f.name in fs)
            append error("Duplicate␣field", f@location);
        if (reference(Name n) := f.\type, n notin defs)
            append error("Undefined␣entity␣type", n@location);
        fs += {f.name};
    }
}
```

The first function, `check`, loops over all entities in the `Entities` AST `es`. During the loop a set of names of defined entities is maintained in `defs`. If an entity is encountered the name of which is already in `defs` and error is appended to the result of the for loop. The expression `e@location` obtains the source location annotation from the `name` field of the entity. Source locations are a built-in Rascal data type for identifying source fragments. Such location values consist of the file name, character offset, length, begin and end line, and begin and end column of the fragment. The `implode` described above, obtains these values from the concrete parse tree and puts them as annotations on the AST nodes. By putting source locations in the error message, an IDE, for instance, could link errors to offending source fragments.

After redeclared entity errors are collected, the list of errors is extended by checking each entity individually using `checkEntity`. The returned expression is a kind of fold construct[5], well-known from functional programming. The part before the first `|` is the initial value for the fold. The middle expression states what has to be computed in each iteration. In this expression the variable **`it`** has a special meaning: it contains the accumulated result from all previous iterations. The last part, after the second `|`, is a generator expression (like in the for loop) that produces the subsequent values in `es.entities`. As result, a flat list of error messages is returned.

The second function, `checkEntity`, checks for duplicate fields and undefined references. Again, the main part consists of a loop, now iterating over the list of fields in an entity. If a field name has been encountered already, an error is appended. If the type of a field is a reference to an entity, it should be in the set of defined entities (`defs`). If not, it's an error. This last check uses pattern matching to obtain the referenced name of a field type. The type of the

[5] At the time of writing, this construct is still experimental.

(`f.\type`[6]) is matched against the pattern `reference(Name n);` as a result `n` is bound to the referenced name if the match is successful. The result of the for loop, i.e. a list of error messages, is returned as the result of the function.

2.4 An IDE for Entities

RASCAL integrates with Eclipse through the IMP framework[7]. From within RASCAL, you dynamically register call-backs to extend the currently running Eclipse instance with IDE features for your language. These call-backs include functions for:

- Parsing: used for syntax highlighting and folding.
- Error marking: used to highlight errors in the source code.
- Context-menus: to allow the invocation of arbitrary RASCAL code on the contents of the current editor. This is used, for instance, to trigger code generation, formatting, visualization or refactoring.
- Outliners: used for producing source code outlines that are linked to the editor contents using source locations.
- Annotators: used to provide hyper linking between elements of the DSL code. For instance, to jump from a use of a variable to its declaration.

Listing 5 Function to register the entities language into the IDE

```
public str ENTITIES_EXTENSION = "entities";
public str ENTITIES_LANGUAGE = "Entities";
public void registerEntities() {
    registerLanguage(ENTITIES_LANGUAGE, ENTITIES_EXTENSION, parseEntities);
    registerAnnotator(ENTITIES_LANGUAGE, checkAndAnnotatePT);
    registerOutliner(ENTITIES_LANGUAGE, outlineEntities);
    contribs = {popup(menu(ENTITIES_LANGUAGE,[
                        action("Generate␣Java", generateJava),
                        action("Generate␣XML", generateXML),
                        action("Generate␣SQL", generateSQL)]))};
    registerContributions(ENTITIES_LANGUAGE, contribs);
}
```

The essential function for obtaining an IDE for the entities language is shown in Listing 5. The function `registerEntities` registers the necessary call-backs into IMP so that the IDE features listed above become available. First the language is registered with a name, file extension, and parse function (`parseEntities`). Second, we register an annotator. This can be any function that annotates parse trees obtained using the parse function of the previous line. In this case, the second argument is a function which checks entity parse trees and annotates them with errors. Third, we register an outliner. This is a function which projects entity parse trees on to simpler trees that can be rendered as outlines.

Finally, the local variable `contribs` is initialized to a value that represents a context-menu. It contains references to ordinary RASCAL functions (`generateJava`, `generateXML` and `generateSQL`) which form the entry points for code generation. This value is then passed to the function `registerContributions` to install the menu.

[6] The identifier `type` is escaped using \ because **`type`** is a reserved keyword in RASCAL.

[7] `http://www.eclipse.org/imp/`

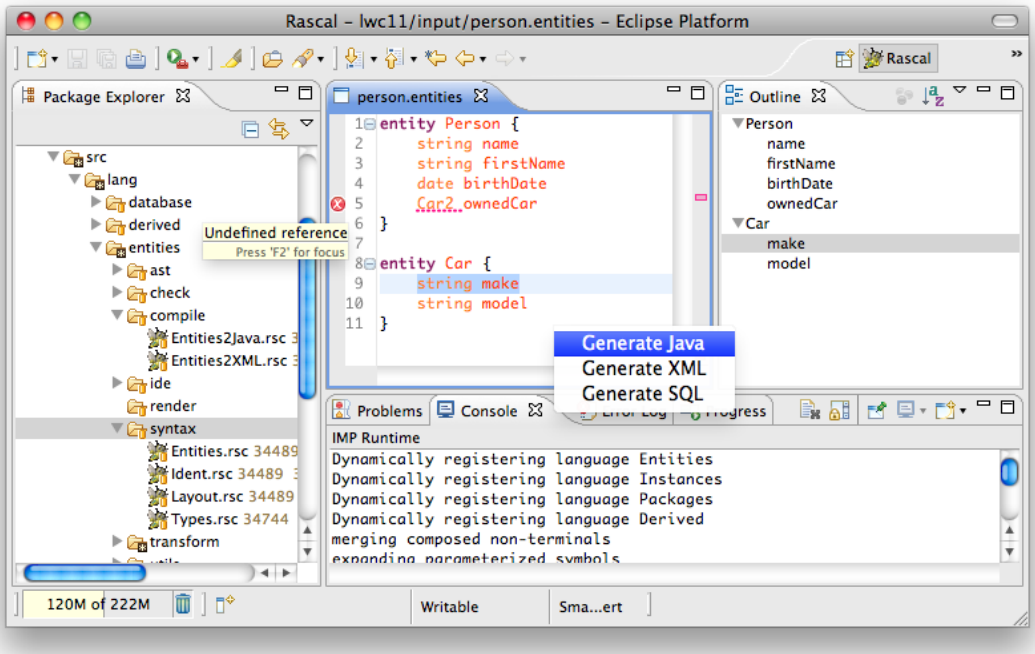

Figure 2.1: Screen-shot of the dynamically generated IDE for the Entities DSL

A screen-shot of the generated IDE for the Entities language is shown in Figure 2.1. The generated IDE actually runs within the RASCAL Eclipse IDE, so the package explorer on the left actually show the source code of the implementation of the Entities DSL. In the middle you see an editor containing a simple entity model. It has syntax highlighting and folding which are both based on the context-free grammar. As you can see, there is an error: entity Person references an undefined entity Car2.

On the right an outline is shown detailing the structure of this entity model. Clicking on an outline element, highlights the corresponding source fragment. Finally, at the bottom of the editor pane, (a fragment of) the context-menu is shown, including entries to invoke various code generators.

2.5 Breaking Down Models

The objective of this section is to show how one could modularize entity definitions, by distributing different entity definitions across different files. It should still be possible for an entity defined in one file, to refer to entities to define in another file. In RASCAL this can be simply achieved by parsing the different files individually, and then merging the resulting ASTs into one AST. This composite AST can then be considered as if it is a single `Entities` definition. As a result, the code generator and consistency checker can be reused as is. The relevant RASCAL code is shown in Listing 6.

The function `merge` takes a variable number of source location (**`loc`**) arguments. Each file is parsed, the result of which is put in a set using a set comprehension. Using a folding comprehension, each list of entities in an `Entities` AST is flattened into a single list, which is then wrapped in an `entities` constructor. The result can be input to further processing. References

Listing 6 Utility functions to merge multiple entity files into a single set of entities

```
public Entities merge(loc files...) {
    ess = { parse(f) | f <- files };
    return entities(( [] | it + es.entities | es <- ess ));
}
```

will be automatically resolved since references are name-based anyhow.

Chapter 3

Extended Entity Modeling

3.1 A Language for Instances

To define the syntax of the instances language, we introduce a new Rascal grammar. The relevant productions are shown in Listing 7. This grammar reuses the identifier and layout syntax of the entities language (not shown).

Listing 7 Syntax definition for the instances language (excerpt)

```
start syntax Instances = instances: Instance* instances;
syntax Instance = @Foldable instance: Name entity Name name "=" "{" Assign* "}";
syntax Assign = assign: Ident name "=" Expression;
syntax Expression = const: Value value | reference: Name name;
```

The start non-terminal (`Instances`) is defined as consisting of zero or more `Instance` definitions. An instance is a (foldable) construct, consisting of an entity type, followed by a name and a list of assignments. An assignment just consists of a name and an assigned `Expression`. `Expressions` include (constant) values and references to other instances. The `Value` non-terminal (not shown) captures dates, strings, integers and floats.

We also have to define an abstract syntax for instances. This code, however, follows the same scheme as the AST types for entities, so we do not show it here.

3.2 Type Checking Instances

Type checking instances is slightly more involved than consistency checking of entities: we now have to combine instances to be checked together with entity definitions to check against. For the rest of this section, we assume that the relevant entity definitions needed for checking instances are somehow provided beforehand. In the actual implementation, instance definitions include simple include directive to indicate the entity context in which the instances are defined. The primary functions of the type checker are shown in Listing 8 and 9.

The first function, `check` (Listing 8), receives two arguments: an `Instances` model and an `Entities` model. In the first line the set of defined entities is put in a map from names to entities. This is done using a map comprehension. Then a map of defined instances, `idefs` is initialized to be the empty map. For each of the instances in `is`, we first check whether the type of the instance corresponds to one of the defined entities. If not it's an error. Second, if the name of the instance is in `idefs` we have duplicate instance definition. At the end of the loop, the instance of the current iteration is recorded in the map `idefs`, together with its type. Finally, the

Listing 8 Type checking of instance definitions against entity definitions

```
public list[Message] check(Instances is, Entities es) {
    edefs = ( e.name: e | e <- es.entities );
    idefs = ();
    errors = for (i <- is.instances) {
        if (i.\type notin edefs)
              append error("Undefined␣type", i@location);
        if (i.name in idefs)
            append error("Duplicate␣instance", i@location);
        idefs[i.name] = i.\type;
    }
    return ( errors | it + check(i, idefs, edefs[idefs[i.name]])
                    | i <- is.instances, i.\type in edefs );
}
```

list of errors is extends with all errors that result from checking instances individually against its defining entity (if it exists).

Listing 9 Type checking of individual instances against an entity type

```
public list[Message] check(Instance i, map[Name, Name] idefs, Entity e) {
    fdefs = ( f.name: f.\type | f <- e.fields );

    list[Message] errors =
        [ error("Undefined␣field", a@location)
            | a <- i.assigns, a.name notin fdefs ]

        + [ error("Undefined␣instance", a@location)
            | a <- i.assigns,  reference(Name n) := a.\value, n notin idefs ]

        + [ error("Missing␣field␣<n>", i@location)
            | n <- domain(fdefs) - { a.name | a <- i.assigns } ];

    return ( errors | it + checkTypes(a, fdefs[a.name], a.\value, idefs)
                    | a <- i.assigns, a.name in fdefs );
}
```

Individual instance are checked in the second `check` function (Listing 9). It receives an instance, a map of defined instances and the entity type. Again, a map is used to store the the defined fields within the entity `e`. The variable `errors` is then filled with three subsequent comprehensions. Each comprehension produces zero or more error messages. Undefined furls are found when assigned names are not in the (domain of the) field definitions map `fdefs`. Undefined instances are found if a name referenced in an assigned value is not in the map of defined instances. Missing fields are found by checking if there are field definitions that are not assigned by instance `i`. Finally, each of the assignments that have corresponding field definitions are checked for type compatibility using the function `checkTypes` (not shown).

An IDE for Instances Just like for the entities, we can use the call-backs into the IDE to dynamically generate IDE support for the instances language. A screen-shot of the IDE is shown in Figure 3.1. Note that this IDE happily coexists with the IDE for entities and the IDE for Rascal in the same Eclipse instance.

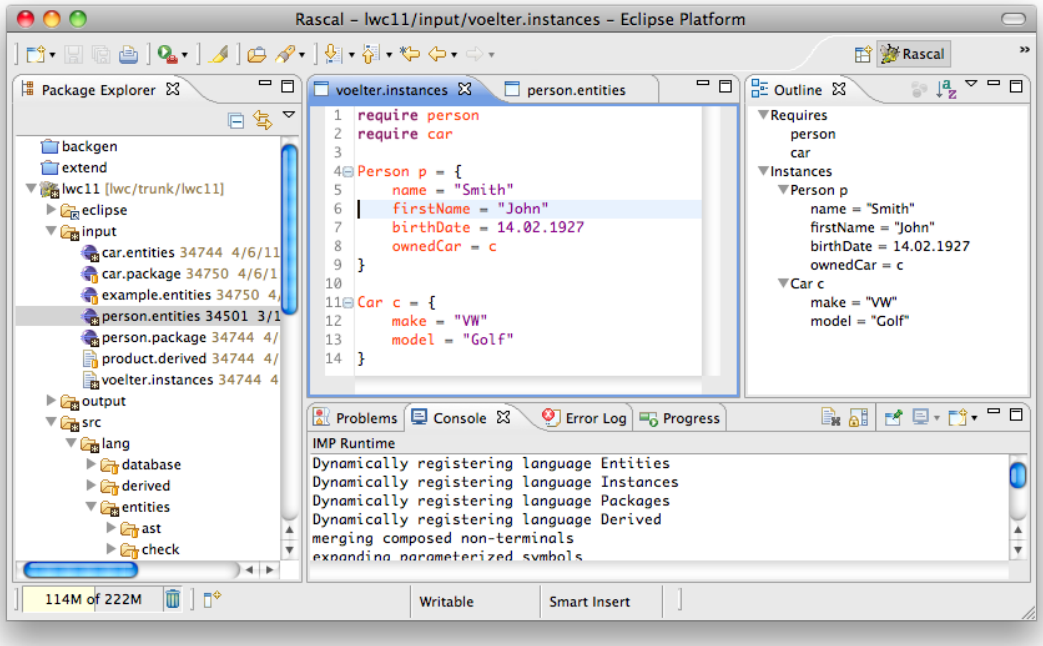

Figure 3.1: Screen-shot of the generated IDE for the instances languages

3.3 Model-to-Model Transformation

In RASCAL, models are *values* and meta-models are *data types*. Model-to-model transformation in RASCAL, thus entails transforming the value of one type to a value of another type. In this section we describe a model-to-model transformation from entity models to a relational database model. In order to do so, I have to define the meta-model, i.e. abstract syntax, for relational database. This is shown in Listing 10.

Listing 10 Abstract syntax for relational databases

```
data Database = database(list[Table] tables);
data Table = table(str name, list[Column] columns);
data Column
     = column(str name, ColumnType \type, list[Constraint] constraints);
data ColumnType = integer() | boolean() | varchar() | date() | text();

data Constraint
     = unique() | key() | notNull() | references(str table, str column);
```

The Database ADT is pretty straight-forward and simplified meta-model for relational databases. Note however that there are only primitive types now and that references between tables are encoded using foreign key constraints.

The resulting database model could be easily transformed to textual SQL code using RASCAL's string templates (cf. the Java code generation above).

Listing 11 Transforming entity models to a database model

```
private str KEY = "_id";
public Database entities2database(Entities es) {
    return database([ entity2table(e) | e <- es.entities ]);
}
public Table entity2table(Entity e) {
    cols = [column(KEY, integer(), [key()])]
         + [ field2column(f) | f <- e.fields ];
    return table(e.name.name, cols);
}
public Column field2column(Field f) {
    t = type2columnType(f.\type);
    cs = [];
    if (reference(name(e)) := f.\type)
        cs = [references(e, KEY)];
    return column(f.name, t, cs);
}
```

3.4 Name spaces

Listing 12 Extension of the entities language to support name spaces (packages)

```
extend lang::entities::syntax::Entities;
start syntax Package
    = package: "package" Ident name "{"
                        Import* imports
                        Entities entities
                "}";
syntax Import = imp: "import" Ident name;
syntax Name = qualified: Ident "." Ident;
```

As models become larger, the need for scoping the use of names becomes more pressing. In this section we will address this problem by adding a name space facility to the existing `Entity` language. Since RASCAL supports modular development of grammars, we can completely reuse the `Entity` grammar and extend it to support name spaces. The source code for the new grammar is shown in Listing 12. We have defined a new top-level element, `Package`, which has a name, and contains import directives and `Entity`s. In order to allow the use of fully qualified names in entities, the `Name` non-terminal is extended with an alternative for qualified names.

3.4.1 Name Resolution

With the syntax for `Package`s in place, we now have to find a way to resolve imports[1]. This process proceeds as follows: first a `WorkingSet` is computed; this data structure is a relation between package names and `LoadResult`s. A `LoadResult` either indicates success—in which case the AST of the found package is included—, or failure (`notFound()`); see Listing 13. The working set is the starting point for further processing.

Listing 13 Helper data structures for loading packages

```
data LoadResult = notFound() | success(loc file, Package package);
alias WorkingSet = rel[str pkgName, LoadResult result];
```

[1] For packages we make the following assumptions: packages are not nested, imports are non-transitive but may be cyclic, entities are declared out of order and may have cyclic dependencies.

Further processing, in this case, consists of checking for errors such as:

- Package could not be found (in import or qualified name)
- Qualified name in an `Entity` declaration does not correspond to the name of the package it is contained in.
- Name collision when two imported packages export the same name.

These checks are implemented in a way similar to the earlier checkers in this document (cf. Listing 4, 8 and 9) so we omit them here.

If a working set is checked to be without errors, all names used in each package can be resolved to fully qualified names. In RASCAL, this is typically performed using a source-to-source transformation. This function is shown in Listing 14. The for loop iterates over all packages that have been successfully loaded. Then the imported packages (including the name of the current package) are put in `imps` (using the helper function `imports` which is not shown). The current package is then traversed using the RASCAL built-in **visit** construct. The **visit** construct traverses any value, and matches the patterns in each of its cases, to the elements visited. If a pattern matches, the corresponding code is executed. In this case, when **visit** encounters an unqualified name, it is replaced (using ⇒) with the fully qualified name. This new name is constructed in the **when** clause based on the name of the imported package `p2` that exports it. The new `qualified` name gets the same source location as the original name, so that type errors may be related to the source code.

Listing 14 Name resolution

```
public WorkingSet resolve(WorkingSet pkgs) {
    for (k <- pkgs, success(l, pkg) := pkgs[k]) {
        imps = imports(pkg) + {pkg.name};
        pkg = visit (pkg) {
            case Name n:name(str x) => qualified(p2.name, x)[@location=n@location]
                 when i <- imps, success(_, p2) := pkgs[i], x in exports(p2)
        }
        pkgs[k] = success(l, pkg);
    }
    return pkgs;
}
```

Now that all packages in the working set have been resolved, the original type checker for entities can be reused. This is a consequence of the fact that the type checker is oblivious to the structure of names. The passing of the location of the old name to the new qualified name in Listing 14 will ensure that errors are correctly highlighted in the source code.

3.4.2 Java Code Generation

In order to generate code for a `Package`, we assume that packages are fully resolved and have no errors. The Java code generator for packages is an extension of the original Java code generator. It is simply the function shown in Listing 15. It maps entity packages to Java packages. For each of the imports, a corresponding Java import directive is generated, importing all elements in the package. Finally, the function `entity2java` from the original code generator (cf. Listing 3) is called in a comprehension to generate code for each entity in the package[2].

In order to deal with qualified names, we have to extend the function that `type2java` in order to deal with the new AST constructor. It simply returns a Java qualified name (bottom of Listing 15).

[2]The function `intercalate` joins a list of elements into a string with the first argument as separator.

Listing 15 Extension of the entities code generator to support packages

```
import lang::packages::ast::Packages;
extend lang::entities::compile::Entities2Java;
public str package2java(Package pkg) {
    return "package␣<pkg.name>;
           '<for (i <- pkg.imports) {>
           'import␣<i.name>.*;
           '<}>
           '<intercalate("\n",  [ entity2java(e) | e <- pkg.entities ])>";
}
public str type2java(qualified(str pkg, str name)) = "<pkg>.<name>";
```

3.5 Integrating Manually Written Code

Listing 16 Extending the entities language to support computed attributes

```
import lang::entities::syntax::Layout;
import lang::entities::syntax::Ident;
import lang::entities::syntax::Types;
import lang::entities::syntax::Entities;
syntax Field
    = derived: Type Ident "=" Expression
    | annotated: Annotation Type Ident ;

syntax Annotation = annotation: "@" Ident "(" String ")";
syntax Expression
    = nat: Natural value
    | field: Ident var
    | bracket Bracket: "(" Expression exp ")"
    | neg: "-" Expression arg
    > left (
          mul: Expression lhs "*" Expression rhs
        | div: Expression lhs "/" Expression rhs
    )
    > left (
          add: Expression lhs "+" Expression rhs
        | sub: Expression lhs "-" Expression rhs
    );
```

The entities languages as described up till now just assume all attributes are actual data attributes. In this sub section, we will extend the entities language with computed attributes. We will cater for computed attributes in two ways:

- An attribute can be given an expression which computes a value in terms of constants and other attributes. For instance, to compute the VAT based on a `price` field of an entity, one could use the following notation:

  ```
  double vat = 0.19 * price
  ```

  A code generator will translate the expression on the right of the equals sign will be directly to host language expressions.

- The other form of computed attributes uses a Java like annotation mechanism to integrate with manually written code. In this, we will use an annotation `@host("...")` where `...`

should point to a fully qualified static Java method which accepts a single parameter of the type this attribute is defined in. For instance:

```
@host("tax.utils.VAT.computeVat")
double vat
```

In this case the code generator assumes that such static methods will be available when the resulting Java code is compiled.

In order to support such computed attributes, we will first extend the syntax of the core language for entities; this is shown in Listing 16. Again, we omit the extension of the abstract syntax since it straightforwardly derives from the grammar extension. The grammar adds two new alternatives for the `Field` non-terminal: `derived` fields and `annotated` fields. For the derived attributes we need an expression language, which is captured by the `Expression` non-terminal.

The syntax definition of the expression language shows two new production attributes (**bracket** and **left**), and a new production combinator (> vs. |). These are used to disambiguate (infix) expressions. The **bracket** keyword indicates that the production should not be included in ASTs derived from these parse trees. Hence, in the ADT defining the abstract syntax of these expressions there will be no constructor corresponding to this production. Bracket productions are used for disambiguation during parsing; abstract syntax is non-ambiguous by definition.

Listing 17 Extending the Java generator to support computed attributes

```
extend lang::entities::compile::Entities2Java;
import lang::derived::ast::Derived;
public str field2java(derived(t, n, exp)) {
    return getter(t, n, exp2java(ex));
}
public str field2java(annotated(a, t, n)) {
    method = substring(a.string, 1, size(a.string) - 1);
    return getter(t, n, "<method>(this)");
}
private str getter(Type t, str n, value exp) {
    <tn, cn> = <type2java(t), capitalize(n)>;
    return "public␣<tn>␣get<cn>()␣{
           '␣␣␣␣return␣<exp>;
           '}";
}
```

The **left** keyword and the > production combinator deal with associativity and precedence of (unary/binary) (prefix/postfix/infix) operators. The **left** keyword, applied to a group of productions, indicates that these productions are *left associative*; there are similar keywords for right associativity (**right**) and non-associativity (**non-assoc**). If two (groups of) productions P and P' are combined using the greater-than sign (>) it means that the productions in P have higher precedence than the productions in P'. In our example, it means that the expression language follows traditional rules for associativity and precedence: unary minus binds stronger that infix operators, multiplication and division are left associative and bind stronger than addition and subtraction, which in turn are both left associative.

Now that we have a language for computed attributes, we have to extend the Java code generator to deal with the new kinds of attributes. The actual extension point is the code for converting fields to Java code: `field2java` in Listing 3. This function was defined using pattern-based dispatch: it matches on the standard field constructors. We can now extend the code generator by adding extra implementations of `field2java` that match the additional field constructors. This is shown in Listing 17.

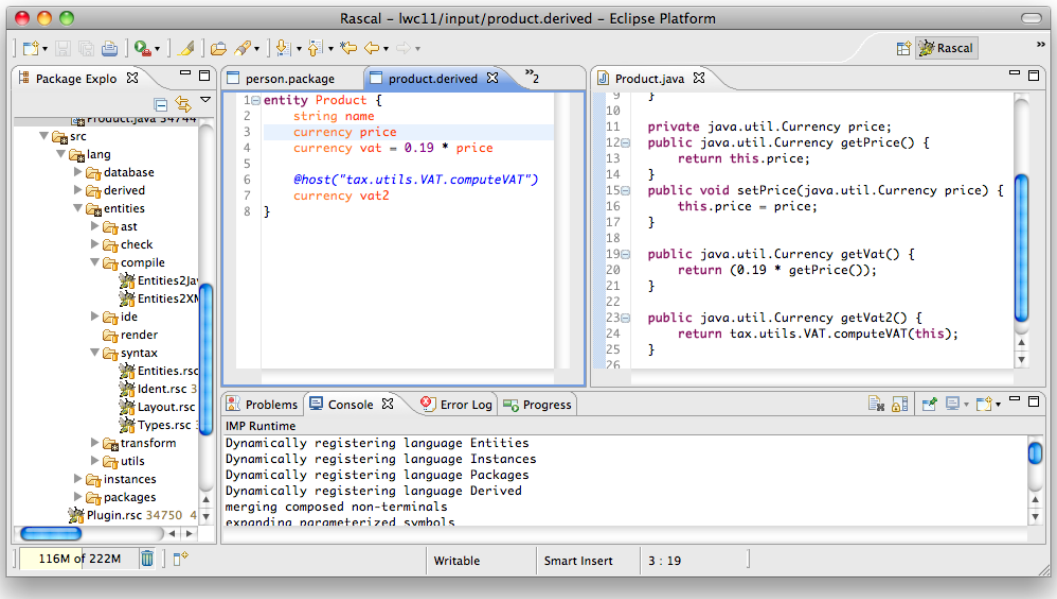

Figure 3.2: Screen-shot of the generated IDE for the entities languages with support for computed attributes

The first extension is concerned with attributes which are accompanied with an expression. The resulting Java code is generated by calling an auxiliary function `getter` which produces a getter-method for a type, field name and Java expression. The Java expression is produced from the `Entity` expression using `exp2java` (not shown) which converts additions, multiplications field references etc. to their Java equivalents.

For annotated attributes, we extract the string of the `host` annotation and construct a getter which returns the result of calling the static method.

Obviously, for computed attributes, it does not make sense to generate setter methods, so this concludes the code generation for entities with derived attributes.

As an example, consider the code of the screen-shot of Figure 3.2. The IDE is generated in the same style as the IDE for entities (cf. Listing 5). The entity in the editor on the left defines a product entity which, for the sake of this example, has two VAT attributes. Both these attributes are computed on the basis of the price of the product. The first (`vat`) uses the expression language, and the second (`vat2`) is annotated with a qualified name of a Java method. The editor on the right shows the relevant part of the generated Java code. The getter of the `vat` attribute consists of an Java expression equivalent to the expression in the entity definition. The getter of `vat2`, on the other hand, consists of a call to the method referenced in the annotation.

3.6 Multiple Generators

To show that it is rather easy to add multiple code generators, I could have easily shown a code generator targeting C# or Ruby, or any similar object-oriented language. However, such code generators will be very similar to the Java code generator(s) shown so far. So, in this sub section we will define a really simple code generator converting entity definitions to XML. The source code for this generator is shown in Listing 18.

Listing 18 Generate XML for entities

```
import XMLDOM;
public Node entities2xml(Entities es) {
    return document(element(none(), "entities",
       [ entity2element(e) | e <- es.entities ]));
}
public Node entity2element(Entity e) {
    a = attribute(none(), "name", e.name.name);
    return element(none(), "entity",
       [a, [ field2element(f) | f <- e.fields ]]);
}
public Node field2element(Field f) {
    attrs = [attribute(none(), "name", f.name)];
    if (primitive(t) := f.\type) {
        attrs += [attribute(none(), "type", getName(t))];
    }
    else {
        attrs += [attribute(none(), "type", "ref"),
                    attribute(none(), "references", f.\type.name.name)];
    }
    return element(none(), "field", attrs);
}
```

Rascal includes basic XML support in its standard library through the `XMLDOM` module. This module contains a simple ADT for XML Document Object Model (DOM) trees. If you convert entities to values over this ADT, they can be automatically serialized as XML documents.

The top level function (`entities2xml`) in Listing 18 creates a document node for the complete set of entities: it contains a single root element (tag "entities") which contains each entity converted to XML as a child. Every entity is converted to XML using the `entity2xml` function. This function, in turn, creates a element node for an entity, with a name (XML) attribute. The XML content for each field is appended to the list of children of this element. Note how the list comprehension containing the `field2element` invocations is spliced in to its surrounding list, next to the name attribute node. This is a feature of Rascal that may be surprising at first sight: list splicing is the default,—if you want to create a list of lists (resp. set of sets, for that matter) you have to use an extra pair of brackets (resp. curly braces). Finally, the `field2element` function creates an XML element node for a field which has a name attribute, a type attribute, and, if the field has a reference type, a "references" attribute. For all created elements, we have used the default XML name space which is indicated by `none()` as the first argument to attribute and element constructors.

Chapter 4

Conclusion

In this article we have introduced the Rascal language workbench by using the LWC'11 challenge as a case study. Let's summarize what we've seen so far.

First, in Chapter 2 we have defined a small DSL for defining Entities. We have defined the concrete syntax using Rascal's context-free grammars, and the abstract syntax using ADTs. We then used Rascal's string templates to implement a code generator, targeting Java. Next, we have shown how a number of simple constraints can be checked by traversing and querying Entity ASTs. All three aspects of DSL implementation have been integrated in a specialized IDE for the DSL, which features syntax highlighting, folding, error marking, context menus and outlines (Section 2.4). Finally, we have indicated how entity definition can be distributed over multiple files by individually parsing each files and merging the resulting set of ASTs into a single composite entities specification. After the merge, the code generator and constraint checker can be reused as is.

In Chapter 3, we extended the basic entities DSL in a number of ways. First, we have defined a language for entity instances. Again this included the definition of concrete and abstract syntax. More importantly, we have shown how instances can be type checked against entity definitions.

The second extension of the entities DSL consists of a Model-to-Model transformation (Section 3.3. We have described how entity models (ASTs) can be transformed to relational database models. These relational database models are again defined using ADTs. Model-to-Model transformation in Rascal thus entails transformation of ASTs.

In order to illustrate modularity and composition of language definitions in Rascal, we have extended the entities DSL to support name spaces (Section 3.4. An additional aspect of this language is name resolution: finding the right entity definition for each name occurrence. The syntax of the new language is again defined as a modular extension of the base language. Because we have implemented name resolution as a transformation that transforms names to qualified names, we have been able to reuse the code generator and constraint checker of the base language to a very large extent.

Finally, in Section 3.5, the base entities language is extended to support computed attributes. First, the syntax of the entities language is extended with an expression language, to directly compute the value of an attribute. And second, we have added field annotations that indicate that an attribute is to be computed using a Java method. For both extensions the Java code generator had to be extended. Again, we were able to do this in a modular and incremental fashion.

For the sake of exposition, the source listings in this document have often been excerpts. It is therefore instructive to have a look at the actual size of the complete solution. Table 4.1 shows an overview of the number of non-comment, non-blank lines of code. The figures are shown per language (Entities, Instances, Packages and Derived), and per aspect (Syntax,

	Syntax	AST	Checker	Generator	IDE	*Total*
ENTITIES	45	23	27	32	72	199
INSTANCES	40	25	58	–	56	179
PACKAGES	+11	+11	114	+28	62	226
DERIVED	+28	+19	–	+32	+36	115
Total	124	78	199	92	226	719

Table 4.1: Non-comment lines-of-code per language, per aspect

AST, Checker, Generator and IDE). A +-sign indicates that the number describes an extension of the base entities language. For the language with name spaces (PACKAGES) loading of files and name resolution is included in the numbers of the Checker aspect. Currently there is no checker for entities with derived attributes, except the one of the base language. Excluded from these numbers are: the model-to-model transformation (Section 3.3), SQL code generation, and the XML generator for entities (Section 3.6), packages, and utility modules for parsing, imploding and merging. The total number of lines of code, including the components omitted from Table 4.1, is 953.