### 1.10 Example Tools

You could argue that this whole business about DSLs is nothing new. It has long been possible to build custom languages using parser generators such as lex/yacc, ANTLR or JavaCC. And of course you would be right. Martin Fowler’s DSL book4

|  |  |
| --- | --- |
| emphasizes this aspect.  However, I feel that language workbenches, which are tools to efficiently create, integrate and use sets of DSLs in powerful IDEs, make a qualitative difference. DSL developers, as well as the people who use the DSLs, are used to powerful, feature-rich IDEs and tools in general. If you want to establish the use of DSLs and you suggest that your users use **vi** or **notepad.exe**, you won’t get very far with most people. Also, the effort of developing (sets of) DSLs and their IDEs has been reduced significantly by the maturation of language workbenches. This is why I focus on DSL engineering with language workbenches, and emphasize IDE development just as much as language development.  This is not a tutorial book on tools. However, I will show you how to work with different tools, but this should be understood more as representative examples of different tooling |  |
| approaches5. I tried to use diverse tools for the examples, but |  |
| for the most part I stuck to those I happen to know well and that have serious traction in the real world, or the potential to do so: Eclipse Modeling + Xtext, JetBrains MPS, SDF/Stratego/Spoofax, and, to some extent, the Intentional Domain Workbench. All except the last are open source. Here is a brief overview over the tools.  *1.10.1 Eclipse Modeling + Xtext* |  |
| The Eclipse Modeling project is an ecosystem – frameworks and tools – for modeling, DSLs and all that’s needed or useful around it. It would easily merit its own book (or set of books), so I won’t cover it extensively. I have restricted myself to Xtext, the framework for building textual DSLs, Xtend, a Java-like language optimized for code generation, as well as EMF/Ecore, the underlying meta meta model used to represent model data. Xtext may not be as advanced as SDF/Stratego or MPS, but the tooling is very mature and has a huge user community. Also, the surrounding ecosystem provides a huge number of add-ons that support the construction of sophisti- |  |

cated DSL environments. I will briefly look at some of these tools, among them graphical editing frameworks. *1.10.2 JetBrains MPS*

|  |  |
| --- | --- |
| The Meta Programming System (MPS) is a projectional language workbench, which means that no grammar and parser is involved. Instead, editor gestures change the underlying AST directly, which is projected in a way that looks like text. As a consequence, MPS supports mixed notations (textual, symbolic, tabular, graphical) and a wide range of language composition features. MPS is open source under the Apache 2.0 license, and is developed by JetBrains. It is not as widely used as Xtext, but supports many advanced features.  *1.10.3 SDF/Stratego/Spoofax* |  |
| These tools are developed at the University of Delft in Eelco Visser’s group. SDF is a formalism for defining parsers for context-free grammars. Stratego is a term rewriting system used for AST transformations and code generation. Spoofax is an Eclipse-based IDE that provides a nice environment for working with SDF and Stratego. It is also not as widely used as Xtext, but it has a number of advanced features for language modularization and composition.  *1.10.4 Intentional Domain Workbench* |  |
| A few examples will be based on the Intentional Domain Workbench (IDW). Like MPS, it uses the projectional approach to editing. The IDW has been used to build a couple of very interesting systems that can serve well to illustrate the power of DSLs. The tool is a commercial offering of Intentional Software.  Many more tools exist. If you are interested, I suggest you look |  |
| at the Language Workbench Competition6, where a number of language workbenches (13 at the time of writing of this book) are illustrated by implementing the same example DSLs. This provides a good way of comparing the various tools. |  |

### 1.11 Case Studies and Examples

I strove to make this book as accessible and practically relevant as possible, so I provide lots of examples. I decided against a single big, running example because (a) it becomes increasingly complex to follow, and (b) fails to illustrate different approaches to solving the same problem. However, we use a set of case studies to illustrate many issues, especially in Part II, DSL design. These examples are introduced below. These are taken from real-world projects.

#### 1.11.1 Component Architecture

|  |  |  |  |
| --- | --- | --- | --- |
| This language is an architecture DSL used to define the software architecture of a complex, distributed, component-based | | |  |
| system in the transportation domain7. Among other architec- | | |  |
| tural abstractions, the DSL supports the definition of components and interfaces, as well as the definition of systems, which are connected instances of components. The code below shows interfaces and components. An interface is a collection of methods (not shown) or collections of messages. Components then provide and require ports, where each port has a name, an interface and, optionally, a cardinality. | | |  |
| **namespace** com.mycomany { **namespace** datacenter { **component** DelayCalculator { **provides** aircraft: IAircraftStatus **provides** console: IManagementConsole **requires** screens[0..n]: IInfoScreen  } **component** Manager { **requires** backend[1]: IManagementConsole  } **interface** IInfoScreen { **message** expectedAircraftArrivalUpdate( id: ID, time: Time ) **message** flightCancelled( id: ID )  } **interface** IAircraftStatus ... **interface** IManagementConsole ... }  } |

The next piece of code shows how these components can be instantiated and connected.

|  |
| --- |
| **namespace** com.mycomany.test { **system** testSystem { **instance** dc: DelayCalculator **instance** screen1: InfoScreen **instance** screen2: InfoScreen **connect** dc.screens **to** (screen1.default, screen2.default)  }  } |

Code generators generate code that acts as the basis for the implementation of the system, as well as all the code necessary to work with the distributed communication middleware. It is used by software developers and architects and implemented with Eclipse Xtext.

#### 1.11.2 Refrigerator Configuration

This case study describes a set of DSLs for developing cooling algorithms in refrigerators. The customer with whom we have built this language builds hundres of different refrigerators, and coming up with energy-efficient cooling strategies is a big challenge. By using a DSL-based approach, the development and implementation process for the cooling behavior can be streamlined a lot.

Three languages are used. The first describes the logical hardware structure of refrigerators. The second describes cooling algorithms in the refrigerators using a state-based, asynchronous language. Cooling programs refer to hardware features and can access the properties of hardware elements from expressions and commands. The third language is used to test cooling programs. These DSLs are used by thermodynamicists and are implemented with Eclipse Xtext.

The code below shows the hardware structure definition in the refrigerator case study. An appliance represents the refrigerator. It consists mainly of cooling compartments and compressor compartments. A cooling compartment contains various building blocks that are important to the cooling process. A compressor compartment contains the cooling infrastructure itself, e.g. a compressor and a fan.

|  |
| --- |
| **appliance** KIR {  **compressor compartment** cc { **static compressor** c1 **fan** ccfan  } **ambient tempsensor** at  **cooling compartment** RC { **light** rclight **superCoolingMode door** rcdoor **fan** rcfan **evaporator tempsensor** rceva  }  } |

The code below shows a simple cooling algorithm. Cooling algorithms are state-based programs. States can have entry actions and exit actions. Inside a state we check whether specific conditions are true, then change the status of various hardware building blocks, or change the state. It is also possible to express deferred behavior with the **perform ...after** keyword.

**program** Standardcooling **for** KIR { **start**:

|  |
| --- |
| **entry** { **state** noCooling }  **state** noCooling:  **check** ( RC->needsCooling && cc.c1->standstillPeriod > 333 ) { **state** rcCooling  }  **on isDown** ( RC.rcdoor->open ) { **set** RC.rcfan->active = **true set** RC.rclight->active = **false perform** rcFanStopTask **after** 10 { **set** RC.rcfan->active = **false** }  }  **state** rcCooling:  ...  } |

Finally, the following code is a test script to test cooling programs. It essentially stimulates a cooling algorithm by changing hardware properties and then asserting that the algorithm reacts in a certain way.

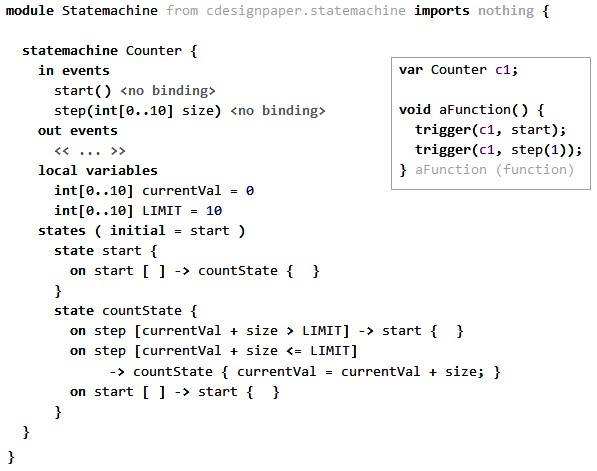
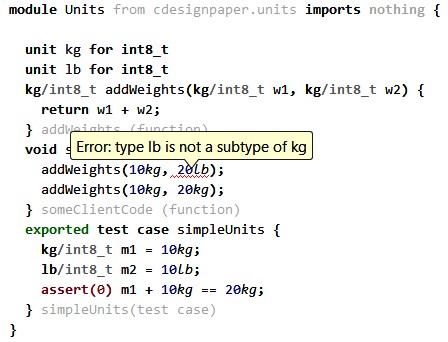
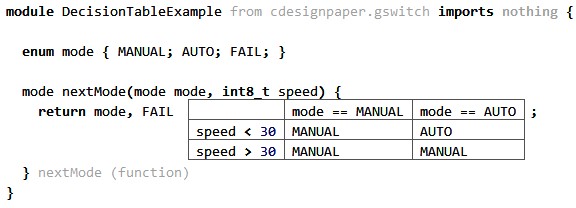
|  |
| --- |
| **cooling** test **for** Standardcooling { **prolog** { **set** cc.c1->standstillPeriod = 0  }  // initially we are not **cooling assert**-**currentstate**-**is** noCooling // then we say that RC needs **cooling**, but // the standstillPeriod **is** still too low.  **mock**: **set** RC->needsCooling = **true step assert**-**currentstate**-**is** noCooling  // now we increase standstillPeriod and **check**  // **if** it now goes to rcCooling **mock**: **set** cc.c1->stehzeit = 400 **step assert**-**currentstate**-**is** rcCooling } |

#### 1.11.3 mbeddr C

This case study covers a set of extensions to the C programming language tailored to embedded programming[[1]](#footnote-1), devel-

oped as part of mbeddr.com[[2]](#footnote-2). Extensions include state machines, physical quantities, tasks, as well as interfaces and components. Higher-level DSLs are added for specific purposes. An example used in a showcase application is the control of a Lego Mindstorms robot. Plain C code is generated and subsequently compiled with GCC or other target device specific compilers. The DSL is intended to be used by embedded software developers and is implemented with MPS.

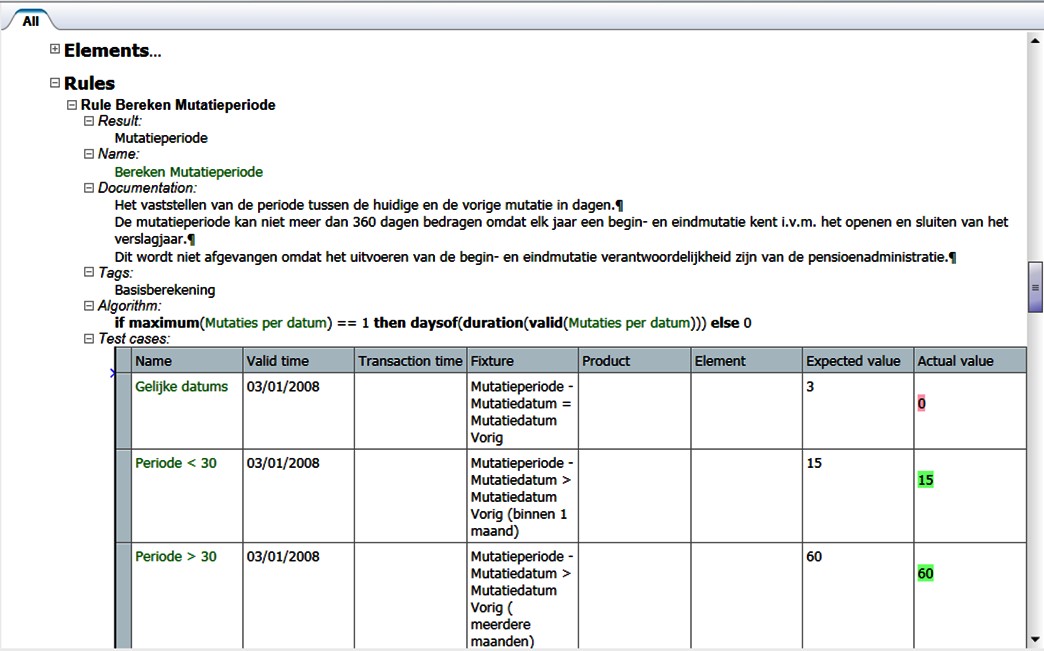
).

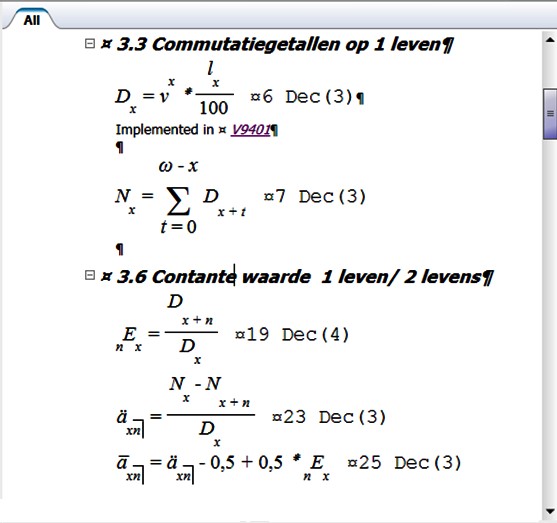


.

#### 1.11.4 Pension Plans

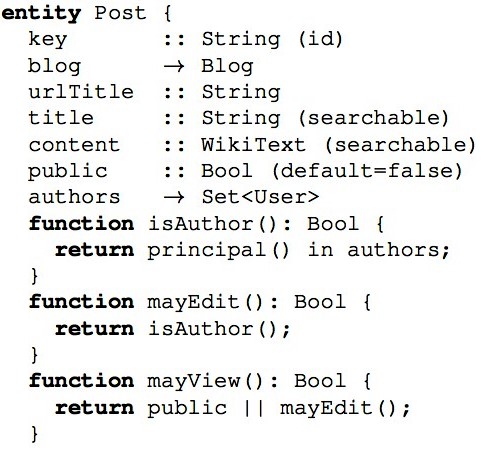
This DSL is used to describe families of pension plans for a large insurance company efficiently. The DSL supports mathematical abstractions and notations to allow insurance mathematicians to express their domain knowledge directly (Fig. 1.5), as well as higher-level pension rules and unit tests using a table notation (Fig. 1.4). A complete Java implementation of the calculation engine is generated. It is intended to be used by insurance mathematicians and pension experts. It has been built by Capgemini with the Intentional Domain Workbench.



.

|  |  |
| --- | --- |
| *1.11.5 WebDSL* |  |
| WebDSL is a language for web programming10 that integrates |  |
| languages to address the different concerns of web programming, including persistent data modeling (**entity**), user in- |  |
| terface templates (**define**), access control11, data validation12, |  |

search and more. The language enforces inter-concern consistency checking, providing early detection of failures[[3]](#footnote-3). The fragments in Fig. 1.6 and Fig. 1.7 show a data model, user interface templates and access control rules for posts in a blogging application. WebDSL is implemented with Spoofax and is used in the researchr digital library14.

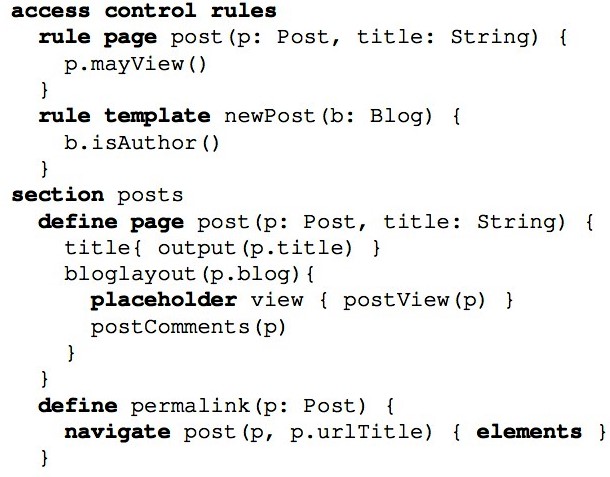


Declarative access control for WebDSL: Combining language integration and separation of concerns. In *ICWE*, pages

175–188, 2008

12 D. Groenewegen and E. Visser. Integration of data validation and user interface concerns in a dsl for web applications. *SoSyM*, 2011

.



1. This system is also used as the ex- [↑](#footnote-ref-1)
2. **mbeddr.com** [↑](#footnote-ref-2)
3. . [↑](#footnote-ref-3)