MARKUS VOELTER, VOELTER@ACM.ORG

DSL ENGINEERING

Part I

DSL Design

1
Core Design Dimensions

This document is specially built for the participants of my OOP 2012 talk on DSL design, since I screwed up timing completely :-)

1.1 Completeness

Completeness refers to the degree to which a language L is able to express *complete* programs. Let us introduce a function G ("code generator") that transforms a program p in L_D to a program q in L_{D-1}. For a complete language, p and q have the same semantics, i.e. $OB(p) == OB(G(p)) == OB(q)$. For incomplete languages where $OB(G(p)) \subset OB(p)$ we have to write additional code in L_{D-1}, to obtain a program in D_{-1} that has the same semantics as intended by the original program in L_D. In cases where we use several viewpoints to represent various concerns of D, the set of fragments written for these concerns must be enough for complete D_{-1} generation. General purpose languages at D_0 are by definition complete.

Another way of stating this is that G produces a program in L_{D-1} that is not sufficient for a subsequent transformation (e.g. a compiler), only the manually written L_{D-1} code leads the sufficiency.

Embedded C: The Embedded C language is complete regarding D_{-1}, or even D_{-n} for higher levels of D, since higher levels are always built as extensions of its D_{-1}. Developers can always fall back to D_{-1} to express what is not expressible directly with L_D. Since the users of this system are developers, falling back to D_{-1} or even D_0 is not a problem. ◄

Refrigerators: This DSL uses several viewpoints: one to define the structure of a refrigerator, and one to describe the actual algorithm. When generating the C code for the algorithm, both viewpoints are needed, because the structure determines how exactly some of the algorithms are implemented in the generated C

code. ◀

1.1.1 *Compensating for Incompleteness*

Integrating the L_{D-1} in case of an incomplete L_D language can be done in several ways:

- by calling "black box" code written in L_{D-1}. This requires concepts in L_D for calling D_{-1} foreign functions. No syntactic embedding of D_{-1} code is required.
- by directly embedding L_{D-1} code in the L_D program. This is useful if L_D is an extension of L_{D-1}, or if the tool provide adequate support for embedding the D_{-1} language into L_D programs.

Just "pasting text into a textfield", an approach used by several graphical modeling tools, is not productive, since no syntactic and semantic integration between the languages is provided.

- by using composition mechanisms of L_{D-1} to "plug in" the manually written code into the generated code without actually modifying the generated files (also known as the Generation Gap pattern [1]). Example techniques for realizing this approach include generating a base class with abstract methods (requiring the user to implement them in a manually written subclass) or with empty callback methods which the user can use to customize in a subclass (for example, in user interfaces, you can return a position object for a widget, the default method returns null, default to the generic layout algorithm). You can delegate, implement interfaces, use #include, use reflection tricks, AOP or take a look at the well-known design patterns for inspiration. Some languages provide partial classes, where a class definition can be split over a generated file and a manually written file.

[1]

- or by inserting manually-written L_{D-1} code into the L_{D-1} code generated from the L_D program using protected regions. Protected regions are areas of the code, usually delimited by special comments, whose (manually written) contents are not overwritten during regeneration of the file

We strongly discourage the use of protected regions. You'll run into all kinds of problems: generated code is not a throw-away product anymore, you have to check it in, and you'll run into all kinds of funny situations with your CM system. Also, often you will accumulate a "sediment" of code that has been generated from model elements that are no longer in the model (if you don't use protected regions, you can delete the whole generated source directory from time to time, cleaning up the sediment).

For DSLs used by developers, incomplete DSLs are usually not a problem because they are comfortable with providing the D_{-1} code expressed in a programming language. Specifically, the DSL users are the same people as those who provide the remaining D_{-1} code, so coordination between the two not a problem.

Component Architecture: This DSL is not complete. Only class skeleton and infrastructure integration code is generated from the models. The component implementation has to be implemented manually in Java using the Generation Gap pattern. The DSL is used by developers, so writing code in a subclass of a generated class is not a problem. ◀

For DSLs used by domain experts, the situation is different. Usually, they are not able to write D_{-1} code, so other people (developers) have to fill in the remaining concerns. Alternatively, developers can develop a predefined set of foreign functions that can be called from within the DSL. In effect, developers provide a standard library (cf. Section **??**) which can be invoked as black boxes from DSL programs.

This requires elaborate collaboration schemes, because the domain experts have to communicate the remaining concerns via prose text or verbal communication.

WebDSL: The core of a web application is concerned with persistent data and their presentation. However, web applications need to perform additional duties outside that core, for which often useful libraries exist. WebDSL provides a *native interface* that allows a developer to call into a Java, library by declaring types and functions from the library in a WebDSL program. ◀

Note that a DSL that does not *cover* all of D can still be *complete*: not all of the programs imaginable in a domain may be expressed with a DSL, but those programs that can be expressed can be expressed completely, without any manually written code. Also, the code generated from a DSL program may require a framework written in L_{D-1} to run in. That framework represents aspects of D outside the scope of L_D.

Refrigerators: The cooling DSL only supports reactive, state based systems that make up the core of the cooling algorithm. The drivers used in the lower layers of the system, or the control algorithms controlling the actual compressors, cannot expressed with the DSL. However, these aspects are developed once and can be reused without adaptations, so using DSLs is not sensible. These parts are implemented manually in C. ◀

Controlling D_{-1} code Allowing users to manually write D_{-1} code, and especially, if it is actually a GPL in D_0, comes with two additional challenges though. Consider the following example: the generator generates an abstract class from some model element. The developer is expected to subclass the generated class and implement a couple of abstract methods. The manually written subclass needs to conform to a specific naming convention so some other generated code can instantiate the manually written subclass. The generator, however, just generates the base class and stops. How can you make sure developers actually do write that subclass, using the correct name?

Of course, if the constructor of the concrete subclass is called from another location of the generated code, and/or if the abstract methods are invoked, you'll get compiler errors. By their nature, they are on the abstraction level of the implementation code, however. It is not always obvious what the developer has to do in terms of the model or domain to get rid of these errors.

To address this issue, make sure there is there a way to make those conventions and idioms interactive. One way to do this is to generate checks/constraints *against the code base* and have them evaluated by the IDE, for example using Findbugs or similar code checking tools. qIf one fails, an error message is reported to the developer. That error message can be worded by the developer of the DSL, helping the devloper understand what exactly has to be done to solve the problem

TODO: cite

in the code.

Broken Promises As part of the definition of a DSL you will implement constraints that validate the DSL program in order to ensure some property of the resulting system (see Section **??**).q For example, you might check dependencies between components in an architecture model to ensure components can be exchanged in the actual system. Of course such a validation is only useful if the manually written code does not introduce dependencies that are not present in the model. In that case the "green light" from the constraint check does not help much.

To ensure that promises made by the models are kept by the (manually written) code, use on of the following two approaches. First, generate code that does not allow violation of model promises. For example, don't expose a factory that allows components to look up and use any other component (creating dependencies), but rather use dependency injection to supply objects for the valid dependencies expressed in the model. Second, use architecture analysis tools (dependency checkers) to validate manually written code. You can easily generate the checking rules for those architecture analysis tools from the models.

Component Architecture: The code generator to Java generates component implementation classes that use dependency injection to supply the targets for required ports. This way, the implementation class will have access to exactly those interfaces specified in the model. An alternative approach would be to simply hand some kind of factory or registry where a component implementation can look up instances of components that provide the interfaces specified by the required ports of the current component. However, this way it would be much harder to make sure that only those dependencies are accessed that are expressed in the model. Using dependency injection *enforces* this constraint in the implementation code. ◄

1.1.2 *Roundtrip Transformation*

Roundtrip transformation means that an L_D program can be recovered from a program in L_{D-1} (written from scratch, or changed manually after generation from a previous iteration of the L_D program). This is challenging, because it requires reconstituting the semantics of the L_D program from idioms or patterns used in the L_{D-1} code. This is the general reverse engineering problem and is not generally possible, although progress has been made over recent years (see for example[2]).

Note that for complete languages roundtripping is generally not

[2] ; ; and
Notice that the problem of "understanding" the semantics of a program written at a too-low abstraction level is the reason for DSLs in the first place: by providing linguistic abstractions for the relevant semantics, no "recovery" is necessary for meaningful analysis and transformation.

useful, because the complete program can be written on L_D in the first place. Even if recovery of the semantics is possible it may not be practical: if the DSL provides significant abstraction over the L_{D-1} program, then the generated L_{D-1} program is so complicated, that manually changing the D_{-1} code in a consistent and correct way is tedious and error-prone.

Roundtripping has traditionally been used with respect to UML models and generated class skeletons. In that case, the abstractions were similar (classes), the tool basically just provides a different concrete syntax. This similarity of abstractions in the code and the model made roundtripping possible to some extent. However, it also made the models relatively useless, because they did *not* provide a significant benefit in terms of abstraction over code details.

We generally recommend to avoid (the attempt of building support for) roundtripping.

Embedded C: This language does not support roundtripping, but since all DSLs are extensions of C, one can always add C code to the programs, alleviating the need for roundtripping in the first place. ◀

Refrigerators: Roundtripping is not required here, since the DSL is complete. The code generators are quite sophisticated, and nobody would want to manually change the generated C code. Since the DSL has proven to provide good coverage, the need to "tweak" the generated code has not come up. ◀

Component Architecture: Roundtripping is not supported. Changes to the interfaces, operation signatures or components have to be performed in the models. This has not been reported as a problem by the users, since both the implementation code and the DSL "look and feel" the same way — they are both Eclipse-based textual editors — and generation of the derived low level code happens automatically on saving a changed model. The workflow is seamless. ◀

Pension Plans: This is a typical application domain DSL where the users never see the generated Java code. Consequently, the language has to be complete and roundtripping is not useful and would not fit into the development process. ◀

1.2 *Fundamental Paradigms*

Every DSL is different. It is driven by the domain to which it applies. However, as it turns out, there are also a number of commonalities between DSLs. These can be handled by modularizing and reusing

(parts of) DSLs as discussed in the *next* section. In *this* section we look at common paradigms for describing DSL structure and behaviour.

1.2.1 Structure

Languages have to provide means of structuring large programs in order to keep them manageable. Such means include modularization and encapsulation, specification vs. implementation, specialization, types and instances as well as partitioning.

The language design alternatives described in this section are usually not driven directly by the domain, or the domain experts guiding the design of the language. Rather, they are often brought in by the language designer or the consumers of the DSL as a means of managing overall complexity. For this reason they may be hard to "sell" to domain experts.

Modularization and Visibility DSL often provide some kind of logical unit structure, such as namespaces or modules. Visibility of symbols may be restricted to the same unit, or in referenced ("imported") units. Symbols may be declared as public or private, the latter making them changeable without consequences for using modules. Some form of namespaces and visibility is necessary in almost any DSL. Often there are domain concepts that can play the role of the module, possibly oriented towards the structure of the organization in which the DSL is used.

Most contemporary programming languages use some form of namespaces and visibility restriction as their top level structure.

Embedded C: As a fundamental extension to C, this DSL contains modules with visibility specifications and imports (Fig. 1.1). Functions, state machines, tasks and all other top-level concepts reside in modules. Header files (which are effectively a poor way of managing symbol visibility) are only used in the generated low level code. ◀

```
module Module1 from HPL.main imports Module2 {

  exported var int aReallyGlobalVar;

  struct aLocallyVisibleStruct {
    int x;
    int y;
  };

  exported int anExportedFunction() {
    return anImportedFunction/Module2();
  } anExportedFunction (function)
}
```

Figure 1.1: Modules are the top-level contruct in this C implementation. Module contents can be exported, which means they are visible to importing modules.

Component Architecture: Components and interfaces live in namespaces. Components are implementation units, and are always private. Interfaces and data types may be public or private. Namespaces can import each other, making the public elements of the imported namespace visible to the importing namespace. The

OSGi generator creates two different bundles: an interface bundle that contains the public artifacts, and an implementation bundle with the components. In case of a distributed system, only the interface bundle is deployed on the client. ◀

Pension Plans: Pension plans constitute namespaces. They are grouped into more coarse-grained packages that are aligned with the structure of the pension insurance business. ◀

Partitioning Partitioning refers to the breaking down of programs into several physical units such as files. These physical units do not have to correspond to the logical modularization of the models within the partitions. Typically each model fragment is stored in its own partition. For example, in Java a public class has to live in a file of the same name (logical module == physical partition), whereas in C# there is no relationship between namespace, class names and the physical file and directory structure. A similar relationship exists between partitions and viewpoints, although in most cases, different viewpoints are stored in different partitions.

If a repository-based tool is used, the importance of paritioning is greatly reduced. Although even in that case, there may be a set of federated and distributed repositories that can be considered partitions

Partitioning may have consequences for language design. Consider a DSL where an concept A contains a list of instances of concept B. The B instances then have to be physically nested within an instance of A in the concrete syntax. If there are many instances of B in a given model, they cannot be split into several files. If such a split should be possible, this has to be designed into the language.

Component Architecture: A variant of this DSL that was used in another project had to be changed to allow a namespaces to be spread over several files for reasons of scalability and version-control granularity. In the initial version, namespaces actually *contained* the components and interfaces. In the revised version, components and interfaces were owned by no other element, but model files (partitions) had a namespace declaration at the top, logically putting all the contained interfaces and components into this namespace. Since there was no technical containment relationship between namespaces and its elements, several files could now declare the same namespace. Changing this design decision lead to a significant reimplementation effort because all kinds of naming and scoping strategies changed. ◀

Other concerns influence the design of a partitioning strategy:

Change Impact which partition changes as a consequence of a particular change of the model (changing an element name might require changes to all references to that element from other partitions)

Link Storage where are links stored (are they always stored in the model

that logically "points to" another one)?, and if not, how/where/when to control reference/link storage.

Model Organzation Partitions may be used as a way of organizing the overall model. This is particularly important if the tool does not provide a good means of showing the overall logical structure of models and finding elements by name and type. Organizing files with meaningful names in directory structures is a workable alternative.

Tool Chain Integration integration with existing, file based tool chains. Files may be the unit of checkin/checkout, versioning, branching or permission checking.

Another driver for using partitions is the scalability of the DSL tool. Beyond a certain file size, the editor may become sluggish.

It is often useful to ensure that each partition is processable separately to reduce processing times. An alternative approach supports the explict definition of those partitions that should be processed in a given processor run (or at least a search path, a set of directories, to find the partitions, like an include path in C compilers). You might even consider a separate build step to combine the results created from the separate processing steps of the various partitions (again like a C compiler: it compiles every file separately into an object file, and then the linker handles overall symbol/reference resolution and binding).

The partitioning scheme may also influence users' team collaboration when editing models. There are two major collaboration models: real-time and commit-based. In real-time collaboration, a user sees his model change when another user changes that same model. Change propagation is immediate. A database-backed repository is often a good choice regarding storage, since the granularity tracked by the repository is the model element. In this case, the partitioning may not be visible to the end user, since they just work "on the repository". This approach is often (at least initially) preferred by non-programmer DSL users.

The other collaboration mode is commit-based where a user's changes only make it to the repository if he performs a *commit*, and incoming changes are only visible after a user has performed an *update*. While this approach can be used with database-backed repositories, it is most often used with file-based storage. In this case, the partitioning scheme is visible to DSL users, because it is those files they commit or update. This approach tends to be preferred by developers, maybe because well-known versioning tools have used the approach for a long time.

Specification vs. Implementation Separating specification and implementation supports plugging in different implementations for the same specification and hence provides a way to "decouple the outside from

the inside". This supports the exchange of several implementations behind a single interface. This is often required as a consequence of the development process: one stakeholder defines the specification and a client, whereas another stakeholder provides one or more implementations.

Interfaces, pure abstract classes, traits or function signatures are a realization of this concept in programming languages.

Embedded C: This DSL adds interfaces and components to C. Components provide or use one or more interfaces. Different components can be plugged in behind the same interface. In contrast to C++, no runtime polymorphism is supported, the translation to plain C maps method invocation to flat function calls. ◀

Refrigerators: Cooling programs can refer to entities defined as part of the refrigerator hardware as a means of accessing hardware elements (compressors, fans, valves). To enable cooling programs to run with different, but similar hardware configurations, the hardware structure can use "trait inheritance", where a hardware trait defines a set of hardware elements, acting as a kind of interface. Other hardware configurations can inherit these traits. As long as cooling programs are only written against traits, they work with any refrigerator that implements the particular set of traits against which the program is written. ◀

Specialization Specialization enables one entity to be a more specific variant of another one. Typically, the more specific one can be used in all contexts where the more general one is expected (the Liskov substitution principle[3]). The more general one may be incomplete, requiring the specialized ones to "fill in the holes". Specialization in the context of DSLs can be used for implementing variants or of evolving a program over time.

[3] In GPLs, we know this approach from class inheritance. "Leaving holes" is realized by abstract methods.

Pension Plans: The customer using this DSL had the challenge of creating a huge set of pension plans, implementing changes in relevant law over time, or implementing related plans for different customer groups. Copying complete plans and then making adaptations was not feasible for obvious reasons. Hence the DSL provides a way for pension plans to inherit from one another. Calculation rules can be marked *abstract* (requiring overwriting in sub-plans), *final* rules are not overwritable. Visibility modifiers control which rules are considered "implementation details". ◀

Refrigerators: A similar approach is used in the refrigerator DSL. Cooling programs can specialize other cooling programs. Since the programs are fundamentally state-based, we had to define what exactly it means to override a pumping program. ◀

Types and Instances Types and instances refers to the ability to define a structure that can be parametrized when it is instantiated.

In programming languages we know this from classes and objects (where constructor parameters are used for parametrization) or from components (where different instances can be connected differently to other instances).

Embedded C: Apart from C's `structs` (which are instantiatable data structures) and components (which can be instantiated and connected), state machines can be instantiated as well. Each instance can be in a different state at any given time. ◄

Superposition and Aspects Superposition refers to the ability to merge several model fragments according to some DSL-specific merge operator. Aspects provide a way of "pointing to" several locations in a program based on a pointcut operator (essentially a query over a program or its execution), adapting the model in ways specified by the aspect. Both approaches support the compositional creation of many different model variants from the same set of model fragments.

This is especially important in the context of product line engineering and is discussed in and in section of this book. **TODO**: ref

Component Architecture: This DSL provides a way of advising component definitions, for example to introduce additional ports from an aspect (Fig. 1.2). An aspect may introduce a port `provided port mon: IMonitoring` that allows a central monitoring component to query the adviced components via the `IMonitoring` interface. ◄

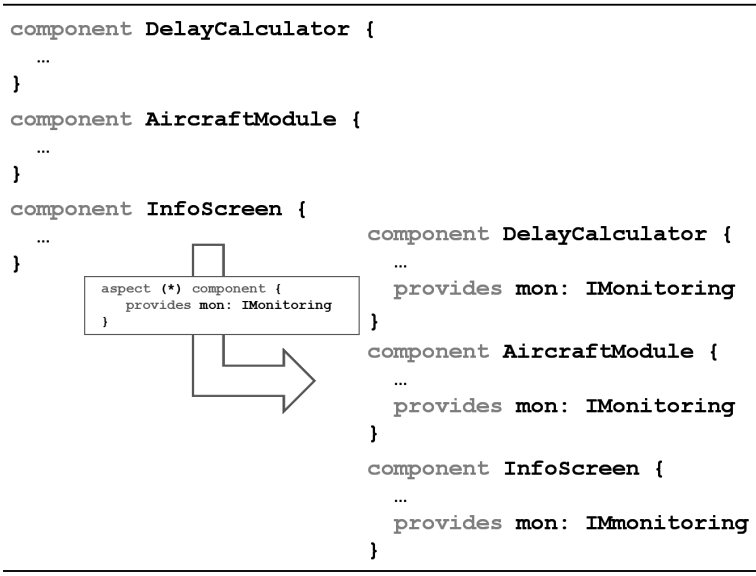

Figure 1.2: The aspect component contributes an additional required port to each of the other components defined in the system.

WebDSL: Entity declarations can be *extended* in separate modules. This makes it possible to declare in one module all data model declarations of a particular feature. For example, in the *researchr* application, a `Publication` can be `Tagged`, which requires an ex-

tension of the `Publication` entity. This extension is defined in the `tag` module, together with the definition of the `Tag` entity. ◀

Versioning Often, artifacts within DSL programs have to be tracked over time. One alternative is to simply version the model files using existing version control systems, or the version control mechanism built into the language workbench. However, this requires users to interact with often complex version control systems and prevents domain-specific adaptations of the version control strategy.

The other alternative is to make versioning and tracking over time a part of the language. For example, model elements can be tagged with version numbers or specify a revision chain by pointing to a previous revision, and enforcing compatibility constraints between those revisions. Instead of declaring explicit versions, business data is often time-dependent, where different revisions of a business rule apply to different periods of time. Support for these approaches can be built directly into the DSL, with various levels of tool support.

Embedded C: No versioning is defined into the DSL. Users work with MPS' integration with popular version control systems. ◀

Component Architecture: Interfaces can specify a `new version of` reference to another interface. If they do so, a constraint enforces that the new version provides at least the same operations as the old version, plus optional additional ones. `new version of` can also be used between components. In this case, the new version has to have the same (or additional) provided ports with the same interfaces or new versions of these interfaces. It must have the same or fewer required ports. Effectively, this means that the new version of something must be replacement-compabible with the old version (the Liskov substitution principle again). ◀

Pension Plans: In the pension workbench, calculation declare have applicability periods. This supports the evolution of calculation rules over time, while retaining reproducability for calculations performed at an earlier point in time. Since the Intentional Domain Workbench is a projectional tool, pension plans can be shown with only the version of a rule valid for a given point in time. ◀

1.2.2 Behavior

The behavior expressed with a DSL must of course be aligned with the needs of the domain. However, in many cases, the behavior required for a domain can be derived from well-known behavioral paradigms, with slight adaptations or enhancements, or simply interacting with

domain-specific structures or data.

Note that there are two kinds of DSLs that don't make use of these kinds of behavior descriptions. Some DSLs really just specify structures. Examples include data definition languages or component description languages (although both of them often use expressions for derived data, data validation or pre- and post-conditions). Other DSLs only specify the kind of behavior expected, and the generator creates the algorithmic implementation. For example, a DSL may specify, simply with a tag such as *async*, that the communication between two components shall be asynchronous. The generator then maps this to an implementation that behaves according to this specification.

Component Architecture: The component architecture DSL is an example of a structure-only DSL, since it only describes black box components and their interfaces and relationships. It uses the specification-only approach to specify whether a component port is intended for synchronous or asynchronous communication. ◀

Embedded C: The component extension to provides a similar notion of interfaces, ports and components as in the previous example. However, since here they are directly integrated with C, C expression can be used for pre- and post-conditions of interface operations (Fig. 1.3). ◀

```
c/s interface IDriver {
  int setDriverValue(int addr, int value)
    pre value > 0
}
```

Figure 1.3: Precondiditions can be added to interface operations. They are executed whenever any runnable that implements the particular operation is executed.

This section described some of the most well-known behavioural paradigms that can serve as useful starting points for behaviour descriptions in DSLs.

This is only an overview over a few paradigms; many more exist. I refer to the excellent Wikipedia entry on *Programming Paradigms* and to the book

Imperative Imperative programs consist of a sequence of statements, or instructions, that change the state of a program. This state may be local to some kind of module (e.g. a procedure or an object), global (as in global variables) or external (when talking to periphery). Procedural and object-oriented programming are both imperative, using different means for structuring and (in case of OO) specialization. Because of aliasing and side effects imperative programs are expensive to analyse. Debugging imperative programs is straight forward and involves stepping through the instructions and watching the state change.

For many people, often including domain experts, this approach is most obvious. Hence it is often a good starting point for DSLs.

Embedded C: Since C is used as a base language, this language is fundamentally imperative. Some of the DSLs on top of it use

other paradigms. ◄

Refrigerators: The cooling language uses various paradigms, but contains sequences of statements to implement aspects of the overall cooling behaviour. ◄

Functional Functional programming uses functions as the core abstraction. A function's return value only depends on the values of its arguments. Functions cannot access global state, no side effects are allowed. Calling the same function several times with the same argument values has to return the same value (that value may even be cached!). No aliasing (through mutable memory cells) is supported, because values are immutable once they are created. Since all dependencies of a computed value are local to a function (the arguments), various kinds of analyses are possible. If assignment to variables is supported, then it uses a form where a variable can only be assigned once.

To create real-world programs, a purely functional language is usually not sufficient, because it cannot affect the environment (after all, this is a side effect). Since there is no state to watch change as the program steps through instructions, debugging can be done by simply showing all intermediate results of all function calls, basically "inspecting" the state of the calculation. This makes building debuggers much simpler. Functional programming is often used for certain parts ("calculation core") of a more complex system.

Pension Plans: The calculation core of pension rules is functional. Consequently, a debugger has been implemented that, for a given set of input data, shows the rules as a tree where all intermediate results of each function call are shown (Fig. 1.4). No "step through" debugger is necessary. ◄

A relevant subset of functional programming is pure expressions (as in `3*2+7 > i`. Instead of calling functions, operators are used. However, operators are just inline-notations for function calls. Usually the operators are hard wired into the language and it is not possible for users to define their own functional abstractions. This is the main differentiator to functional programming in general.

Embedded C: We use expressions in the guard conditions of the state machine extension as well as in pre- and post-conditions for interface operations. In both cases it is not possible to define or call external functions. Of course, C's expression language is reused here. ◄

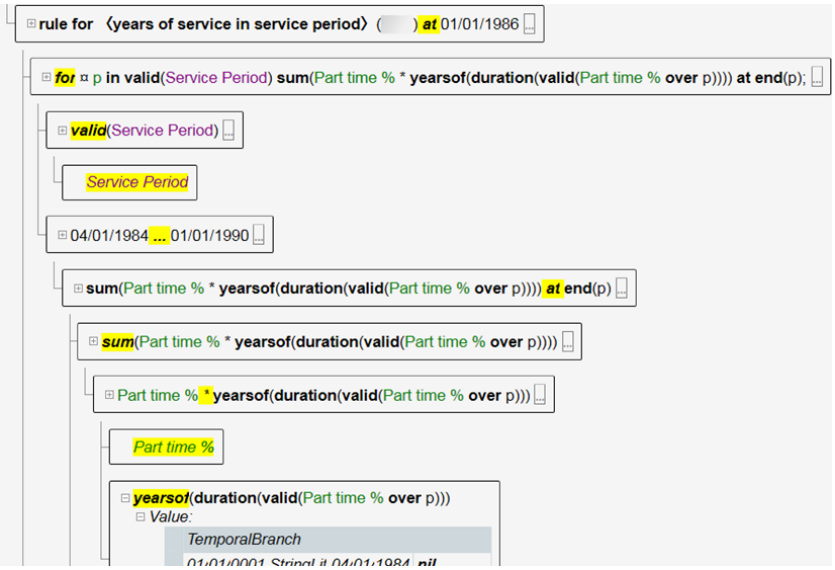

Figure 1.4: Debugging functional programs can be done by showing the state of the calculation, for example as a tree.

Declarative Declarative programming can be considered the opposite of imperative programming (and, to some extent, functional programming). A declarative program does not specify any control flow. It does not specify a sequence of steps of a calculation. Instead, a declarative program only specifies what the program should accomplish, not how. This is often done by specifying a set of properties, equations, relationships or constraints. Some kind of evaluation engine then tries to find solutions. The particular advantage of this approach is that it is not predefined how a solution is found, the evaluation engine has a lot of freedom in doing so, possibly using different solutions in different environments, or evolving the approach over time[4]. This large degree of freedom often makes finding the solution expensive — trial and error, backtracking or exhaustive search may be used. Debugging declarative programs can be hard since the solution algorithm may be very complex and possibly not even be know to the user of the language.

Declarative programming has many important subgroups and use cases. For *concurrent programs*, a declarative approach allows the efficient execution of the same program on different parallel hardware. The compiler or runtime system can allocate the program to available computational resources. In *constraint programming*, the programmer specifies constraints between a set of variables. The engine tries to find values for these variables that satisfy all constraints. Solving mathematical equation systems is an example, as is solving sets of boolean logic formulas.

Component Architecture: This DSL specifies timing and resource

[4] For example, the strategies for implementing SAT solvers have evolved quite a bit over time. SAT solvers are much more scalable today. However, the formalism for describing the logic formulas that are processed by SAT solvers have not changed

characteristics for component and interface operations. Based on this data, one could run an algorithm which allocates the component instances to computing hardware so that the hardware is used as efficiently as possible, while at the same time reducing the amount of bus traffic. This is an example of constraint solving. ◀

Embedded C: This DSL supports presence conditions for product line engineering. A presence condition is a boolean expression over a set of configuration features that determines whether the associated piece of code is present for a given combination of feature selections (Fig. 1.5). To verify the structural integrity of programs in the face of varying feature combination, constraint programming is used (to ensure that there is no configuration of the program where a reference to a symbol is included, but the referenced symbol is not). From the program, the presence conditions and the feature model a set of boolean equations is generated. A solver then makes sure they are consistent by trying to find an example solution that violates the boolean equations. ◀

TODO: cite: refer to DSL impl chapter, and to literature (Czarnecki)

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

Figure 1.5: Code affected by a presence condition is highlighted in blue. The presence condition is rendered on the left of the affected code. It is a boolean expression over a set of (predefined) configuration parameters.

Example: The Yakindu DAMOS block diagram editor supports custom block implementation based on the Mscript language. Mscript (Section 1.6) supports declarative specification of equations between input and output parameters of a block. A solver comes up with an closed, sequential solution that efficiently calculates the output of an overall block diagram. ◀

```
synchronous blockType org::eclipselabs::damos::library::base::_discrete::DiscreteDerivative

input u
output y

parameter initialCondition = 0
parameter gain = 1(s) // normalized

behavior {

        stateful func main<initialCondition, gain, fs>(u) -> y {
                check<0, 1(s), 1(1/s)>(real) -> real

                static assert u is real() :
                        error "Input value must be numeric"

                static assert initialCondition is real() :
                        error "Initial condition must be numeric"

                static assert initialCondition is real() && u is real() => unit(initialCondition) == unit(u) :
                        error "Initial condition and input value must have same unit"

                static assert gain is real() :
                        error "Gain value must be numeric"

                eq u{-1} = initialCondition
                eq y{n} = fs * gain * (u{n} - u{n-1})
        }

}
```

Figure 1.6: An Mscript block specifies input and output arguments of a block (u and v) as well as configuration parameters (*initialCondition* and *gain*). The assertions specify constraints on the data the blocks works with. The *eq* statements specify how the output values are calculated from the input values. Stateful behaviours are supported, where the value for the n-th step depends on values from previous steps (e.g. $n-1$).

Example: Another example for declarative programming is the type system DSL used by MPS itself. Language developers specify a set of type equations containing free type variables, among other things. A unification engine tries to solve the set of equations by assigning actual types to the free type variables so that the set of equations is consistent. We describe this approach in detail in section ◀

TODO: ref

Logic programming is another subparadigm of declarative programming where users specify logic clauses (facts and relations) as well as queries. A theorem prover tries to solve the queries.

The Prolog language works this way

Reactive/Event-based/Agent In this paradigm, behavior is triggered based on events received by some entity. Events may be created by another entity or by the environment (through a device driver). Reactions are expressed by the production of other events. Events may be globally visible or explicitly routed between entities, possibly using filters and/or using priority queues. This approach is often used in embedded systems that have to interact with the real world, where the real world produces events as it changes. A variant of this approach queries input signals at intervals controlled by a scheduler and considers changes in input signals as the events.

Refrigerators: The cooling algorithms are reactive programs that

control the cooling hardware based on environment events. Such events include the opening of a refrigerator door, the crossing of a temperature threshold, or a timeout that triggers defrosting of a cooling compartment. Events are queued, and the queues are processed in intervals determined by a scheduler. ◄

Debugging is simple if the timing/frequency of input events can be controlled. Visualizing incoming events and the code that is triggered as a reaction is relatively simple. If the timing of input events cannot be controlled, then debugging can be almost impossible, because humans are way too slow to fit "in between" events that may be generated by the environment in rapid succession. For this reason, various kinds of simulators are used to debug the behaviour of reactive systems, and sophisticated diagnostics regarding event frequencies or queue filling levels may have to be integrated into the programs as they run in the real environment.

Refrigerators: The cooling language comes with a simulator (Fig. 1.7) based on an interpreter where the behaviour of a cooling algorithm can be debugged. Events are explicitly created by the user, on a time scale that is compatible with the debugging process. ◄

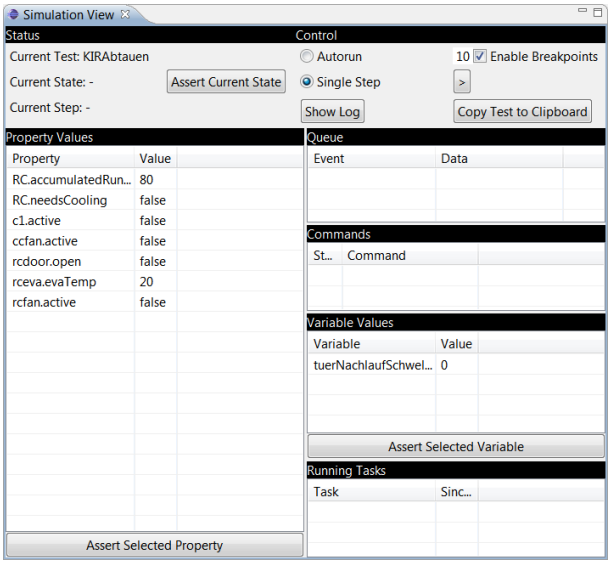

Figure 1.7: The simulator for the cooling language shows the state of the system (commands, event queue, value of hardware properties, variables and tasks). The program can be single-stepped. The user can change the value of variables or hardware properties as a means of interacting with the program.

Dataflow The dataflow paradigm is centered around variables with dependencies (relationships in terms of calculation rules) among them. As a variable changes, those variables that depend on the changing variable are recalculated. We know this approach mainly from two use cases. One is spreadsheets: cell formulas express dependencies to

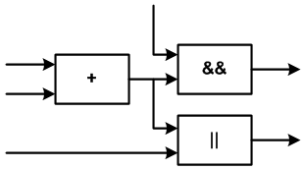

Figure 1.8: Graphical Notation for Flow

other cells. As the values in these other cells change, the dependent cells are updated. The other use case is data flow (or block) diagrams (Fig. 1.8, used in embedded software, ETL systems and enterprise messaging and complex event processing. There, the calculations are encapsulated in the blocks, and the lines represent dependencies — the output of one blocks "flows" into input slot of another block. There are three different execution modes:

- The first one considers the data values as continuous signals. At the time one of the inputs changes, all dependent values are recalculated, the change triggers the recalculation, and the recalculation ripples through the dependency graph. This is the model used in spreadsheets.

- The second one considers the the data values quantized, unique messages. Only if a message is available for all inputs, a new output message is calculated. The recalculation synchronizes on the availability of a message at each input, and upon recalculation, these messages are consumed. This approach is often used in ETL and CEP systems.

- The third approach is time triggered. Once again, the inputs are understood to be continuous signals, and a scheduler determines when a new calculation is performed. It also makes sure that the calculation "ripples through from left to right" in the correct order. This model is typically used in embedded systems.

Debugging these kinds of systems is relatively straight forward because the calculation is always in a distinct state. Dependencies and data flow, or the currently active block and the available messages can easily be visualized in a block diagram notation. Note that the calculation rules themselves are considered black boxes here, whose inside may be built from any other paradigm, often functional. Integrating debuggers for the inside of boxes as well is a more challenging task.

State-based The state-based paradigm describes a system's behaviour in terms of the states the system can be in, the transitions between these states as well as events that trigger these transitions and actions that are executed as states change. State machines are useful for systematically organizing the behavior of an entity. It can also be used to describe valid sequences of events, messages or procedure calls. State machines can be used in an event-driven mode where incoming events actually trigger transitions and the associated actions. Alternatively a state machine can be run in a timed mode, where a scheduler determines when event queues are checked and processed. Except for pos-

sible real-time issues, state machines are easy to debug by highlighting the contents of event queues and the current state.

Embedded C: As mentioned before, this language provides an extension that supports directly working with state machines. Events can be passed into a state machine from regular C code or by mapping incoming messages in components to events in state machines that reside in components. Actions can contain arbitrary C code, unless the state machine shall be verifiable in which case actions may only create outgoing events or change statemachine-local variables. ◄

Refrigerators: The behaviour of cooling programs is fundamentally state driven. A scheduler is used to execute the state machine in regular intervals. Transitions are triggered either by incoming, queued events or by changing property values of hardware building blocks. Note that this language is an example where a behavioral paradigm is used without significant alterations, but working with domain-specific data structures: refrigerator hardware and their properties. ◄

As mentioned before, state-based behavior description is also interesting because it support model checking: a state chart is verified against a set of specifications. The model checker either determines that the state chart conforms to the specifications or provides a counter example. Specifications express something about sequences of states such as: "it is not possible that two traffic lights show green at the same time" or "whenever a pedestrian presses the *request* button, the pedestrian lights eventually will show green". marginnote[-2cm]A good introduction to model checking can be found in [5]. We elaborate on model checking in section.

[5] **TODO**: ref

1.2.3 Combinations

Many DSLs use combinations of various behavioural and structural paradigms described in this section[6]. A couple of combinations are very typical:

[6] Note how this observation leads to the desire to better modularize and reuse some of the above paradigms. Room for research :-)

- a data flow language often uses a functional, imperative or declarative language to describe the calculation rules that express the dependencies between the variables (the contents of the boxes in data flow diagrams or of cells in spreadsheets). Fig. 1.29 shows an example block diagram, and Fig. 1.6 shows an example implementation.
- state machines use expressions as transition guard conditions, as well as typically an imperative language for expressing the actions that are executed as a state is entered or left, or when a transition is executed. An example can be seen in Fig. 1.31

- reactive programming, where "black boxes" react to events, often use data flow or state-based programming to implement the behaviour that determines the reactions.

- in purely structural languages, for example, those for expressing components and their dependencies, a functional/expression language is often used to express pre- and postconditions for operations. A state-based language is often used for protocol state machines, which determines the valid order of incoming events or operation calls.

Note that these combinations can be used to make well-established paradigms domain specific. For example, in the Yakindu State Chart Tools (Fig. 1.31), a custom DSL can be plugged into an existing, reusable state machine language and edtor. Some of the case studies used in this section of the book also use combinations of several paradigms.

Pension Plans: The pension language uses functional abstractions with mathematical symbols for the core actuary mathematics. A functional language with a normal textual syntax is used for the higher-level pension calculation rules. A spreadsheet/data flow language is used for expressing unit tests for pension rules. Various nesting levels of namespaces are used to organize the rules, the most important of which is the pension plan. A plan contains calculation rules as well as test cases for those rules. Pension plans can specialize other plans as a means of expressing variants. Rules in a sub-plan can override rules in the plan from which the sub-plan inherits. Plans can be declared to be abstract, with abstract rules that have to be implemented in sub-plans. Rules are versioned over time, and the actual calculation formula is part of the version. Thus, a pension plan's behaviour can be made to be different for different points in time. ◄

Refrigerators: The cooling behavior description is described as a reactive system. Events are produced by hardware elements (and their drivers). A state machine constitutes the top level structure. Within it, an imperative language is used. Programs can inherit from another program, overwriting states defined in the base program: new transitions can be added, and the existing transitions can be overridden as a way for an extended program to "plug into" the base program. ◄

1.3 *Language Modularity*

Reuse of modularized parts makes software development more efficient, since similar functionality does not have to be developed over and over again. A similar argument can be made for languages. Being able to reuse languages in new contexts make designing DSLs more efficient.

We have identified the following four composition strategies: referencing, extension, reuse and embedding. We distinguish them regarding fragment structure and language dependencies, as illustrated in Fig. 1.9. Fig. 1.10 shows the relationships between fragments and languages in these cases. We consider these two criteria to be essential for the following reasons.

Language modularization and reuse is often not driven by end user or domain requirements, but rather, by a desire of the language designers and implementers for consistency and avoidance of duplicate implementation work.

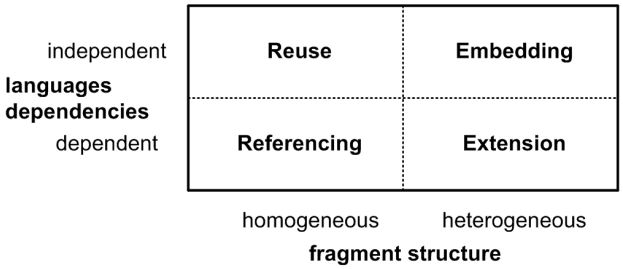

Figure 1.9: We distinguish the four modularization and composition approaches regarding their consequences for fragment structure and language dependencies.

Language dependencies capture whether a language has to be designed with knowledge about a particular composition partner in mind in order to be composable with that partner. It is desirable in many scenarios that languages be composable *without* previous knowledge about all possible composition partners. *Fragment Structure* captures whether the two composed languages can be syntactically mixed. Since modular concrete syntax can be a challenge, this is not always that possible, though often desirable.

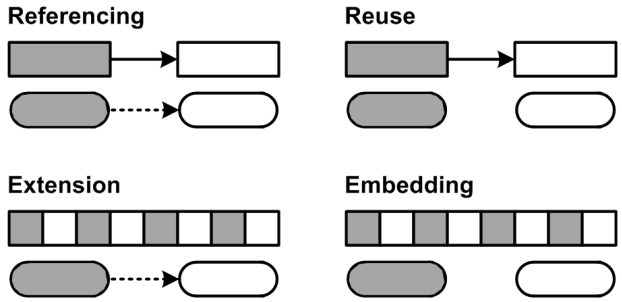

Figure 1.10: The relationships between fragments and languages in the four composition approaches. Boxes represent fragments, rounded boxes are languages. Dotted lines are dependencies, solid lines references/associations. The shading of the boxes represent the two different languages.

1.3.1 *Language Referencing*

Language referencing (Fig. 1.11) enables *homogeneous* fragments with cross-references among them, using *dependent* languages.

A fragment f_2 depends on f_1. f_2 and f_1 are expressed with different languages l_2 and l_1. The referencing language l_2 depends on the referenced language l_1 because at least one concept in the l_2 references a concept from l_1. We call l_2 the *referencing* language, and l_1 the *referenced* language. While equations (2) and (3) continue to hold, (1) does not. Instead

$$\forall r \in Refs_{l_2} \mid lo(r.from) = l_2 \ \wedge \ lo(r.to) = (l_1 \vee l_2) \tag{1.1}$$

(we use $x = (a \vee b)$ as a shorthand for $x = a \vee x = b$).

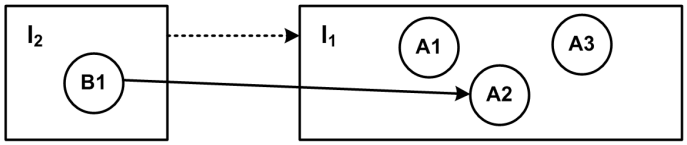

Figure 1.11: Referencing: Language l_2 depends on l_1, because concepts in l_2 reference concepts in l_1. (We use rectangles for languages, circles for language concepts, and UML syntax for the lines: dotted = dependency, normal arrows = associations, hollow-triangle-arrow for inheritance.)

Viewpoints As we have discussed before in Section **??**, a domain D can be composed from different concerns. One way of dealing with this is to define a separate concern-specific DSLs, each addressing one or more of the domain's concerns. A program then consists of a set of concern-specific fragments, that relate to each other in a well-defined way using language referencing. The latter approach has the advantage that different stakeholders can modify "their" concern independent of others. It also allows reuse of the independent fragments and languages with different referencing languages. The obvious drawback is that for tightly integrated concerns the separation into separate fragments can be a usability problem.

Refrigerators: As an example, consider the domain of refrigerator configuration. The domain consists of three concerns. The first concern H describes the hardware structure of refrigerators appliances including compartments, compressors, fans, vents and thermometers. The second concern A describes the cooling algorithm using a state-based, asynchronous language. Cooling programs refer to hardware building blocks and access the their properties in expressions and commands. The third concern is testing T. A cooling test can test and simulate cooling programs. The dependencies are as follows: $A \rightarrow H, T \rightarrow A$. Each of these concerns are implemented as a separate language with references between them. H and A are separated because H is defined by

product management, whereas A is defined by thermodynamicists. Also, several algorithms for the same hardware must be supported, which makes separate fragments for H and A useful. T is separate from A because tests are not strictly part of the product definition and may be enhanced after a product has been released. These languages have been built as part of a single project, so the dependencies between them are not a problem. ◀

Referencing implies knowledge about the relationships of the languages as they are designed. Viewpoints are the classical case for this. The dependent languages *cannot* be reused, because of the dependency on the other language.

Progressive Refinement Progressive refinement, also introduced earlier (Section **??**), also makes use of language referencing.

1.3.2 Language Extension

Language extension Fig. 1.11 enables *heterogeneous* fragments with *dependent* languages. A language l_2 extending l_1 adds additional language concepts to those of l_1. We call l_2 the *extending* language, and l_1 the *base* language. To allow the new concepts to be used in the context provided by l_1, some of them extend concepts in l_1. So, while l_1 remains independent, l_2 becomes dependent on l_1 since

$$\exists i \in Inh(l_2) \mid i.sub = l_2 \ \wedge \ i.super = l_1 \tag{1.2}$$

Consequently, a fragment f contains language concepts from both l_1 and l_2:

$$\forall e \in E_f \mid lo(e) = (l_1 \vee l_2) \tag{1.3}$$

In other words, $C_f \subset (C_{l_1} \cup C_{l_2})$, so f is *heterogeneous*. For heterogeneous fragments (3) does not hold anymore, since

$$\forall c \in Cdn_f \mid lo(co(c.parent)) = (l_1 \vee l_2) \wedge \\ lo(co(c.child)) = (l_1 \vee l_2) \tag{1.4}$$

Note that copying a Language definition and changing it does not constitute a case of language extension, because the extension is not modular, it is invasive. Also, a native interfaces that support calling one language from another one (like calling C from Perl or Java) is not language extension; rather it is a form of language referencing. The fragments remain homogeneous.

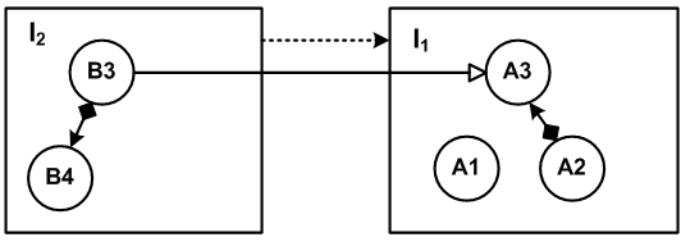

Figure 1.12: Extension: l_2 extends l_1. It provides additional concepts $B3$ and $B4$. $B3$ extends $A3$, so it can be used as a child of $A2$, plugging l_2 into the context provided by l_1. Consequently, l_2 depends on l_2.

Language extension fits well with the hierarchical domains introduced in Section **??**: a language L_B for a domain D may extend a language L_A for D_{-1}. L_B contains concepts specific to D, making analysis and transformation of those concepts possible without pattern matching and semantics recovery. As explained in the introduction, the new concepts are often reified from the idioms and patterns used when using an L_A for D. Language semantics are typically defined by mapping the new abstractions to just these idioms (see Section **??**) *inline*. This process, also known as *assimilation*, transforms a heterogeneous fragment (expressed in L_D and L_{D+1}) into a homogeneous fragment expressed only with L_D.

Language extension is especially interesting if D_0 languages are extended, making a DSL an extension of a general purpose language.

Embedded C: As an example consider embedded programming. The C programming language is typically used as the GPL for D_0in this case. Extensions for embedded programming include state machines, tasks or data types with physical units. Language extensions for the subdomain of real-time systems may include ways of specifying deterministic scheduling and worst-case execution time. For the avionics subdomain support for remote communication using some of the bus systems used in avionics could be added. ◄

TODO: cite Models Paper

Defining a D languages as an extension of a D_{-1} language can also have drawbacks. The language is tightly bound to the D_{-1} language it is extended from. While it is possible for a standalone DSL in D to generate implementations for different D_{-1} languages, this is not easily possible for DSLs that are extensions of a D_{-1} language. Also, interaction with the D_{-1} language may make meaningful semantic analysis of complete programs (using L_D and L_{D-1} concepts) hard. This problem can be limited if isolated L_D sections are used, in which interaction with L_{D-1} concepts is limited and well-defined.

Extension is especially useful for bottom-up domains. The common patterns and idioms identified for a domain can be reified directly into linguistic abstractions, and used directly in the language from which they have been embedded. Incomplete languages are not a problem, since users can easily fall back to D_{-1} to implement the rest. Since DSL users see the D_{-1} code all the time anyway, they will be comfortable falling back to D_{-1} in exceptional cases. This makes extensions suitable only for DSLs used by developers. Domain expert DSLs are typically not implemented as extensions.

Restriction Sometimes language extension is also used to *restrict* the set of language constructs available in the subdomain. For example, the real-time extensions for C may restrict the use of dynamic memory allocation, the extension for safety-critical systems may prevent the use

of void pointers and certain casts. Although the extending language is in some sense smaller than the extended one, we still consider this a case of language extension, for two reasons. First, the restrictions are often implemented by *adding additional* constraints that report errors if the restricted language constructs are used. Second, a marker concept may be added to the base language. The restriction rules are then enforced for children of these marker concepts (e.g. in a module marked as "safe", one cannot use void pointers and the prohibited casts).

Embedded C: Modules can be marked as *MISRA-compliant*, which prevents the use of those C constructs that are not allowed in MISRA . Prohibited concepts are reported as errors directly in the program. ◀

TODO: cite

1.3.3 *Language Reuse*

Language reuse (Fig. 1.13) enables *homogenous* fragments with *independent* languages. Given are two independent languages l_2 and l_1 and two fragment f_2 and f_1. f_2 depends on f_1, so that

$$\exists r \in Refs_{f_2} \mid fo(r.from) = f_2 \wedge fo(r.to) = (f_1 \vee f_2) \tag{1.5}$$

One could argue that in this case reuse is just a clever combination of referencing and extension. While this is true from an implementation perspective, it is worth describing as a separate approach, because it enables the combination of two *independent languages* by adding an adapter *after the fact*, so no pre-planning during the design of l_1 and l_2 is necessary.

Since l_2 is independent, it cannot directly reference concepts in l_1. This makes l_2 reusable with different languages (in contrast to language referencing, where concepts in l_2 reference concepts in l_1). We call l_2 the *context* language and l_1 the *reused* language.

One way of realizing dependent fragments while retaining independent languages is using an adapter language l_A where l_A *extends* l_2 and

$$\exists r \in Refs_{l_A} \mid lo(r.from) = l_A \ \wedge \ lo(r.to) = l_1 \tag{1.6}$$

While language referencing supports reuse of the referenced language, language reuse supports the reuse of the *referencing* language as well. This makes sense for concern DSLs that have the potential to be reused in many domains, with minor adjustments. Examples include role-based access control, relational database mappings and UI specification.

Example: Consider as examples a language for describing user interfaces. It provides language concepts for various widgets, layout definition and disable/enable strategies. It also supports data binding, where data structures are associated with widgets, to enable two-way synchronization between the UI and the data. Using language reuse, the same UI language can be used with different data description languages. Referencing is not enough because

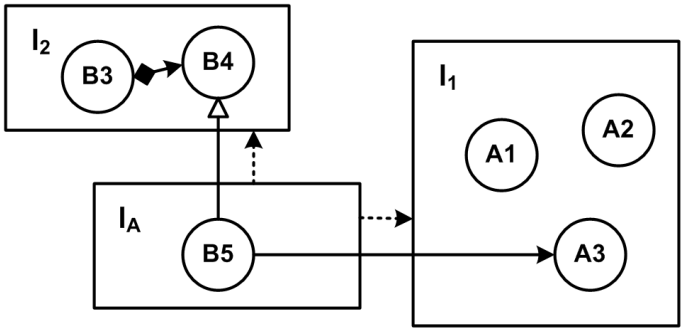

Figure 1.13: Reuse: l_1 and l_2 are independent languages. Within an l_2 fragment, we still want to be able to reference concepts in another fragment expressed with l_1. To do this, an adapter language l_A is added that depends on both l_1 and l_2, using inheritance and referencing to adapt l_1 to l_2.

the UI language would have a direct dependency on a particular data description language. Changing the dependency direction to *data* → *ui* doesn't solve the problem either, because this would go against the generally accepted idiom that UI has dependencies to the data, but not vice versa (cf. the MVC pattern). ◀

Generally, the referencing language is built with the knowledge that it will be reused with other languages, so hooks may be provided for adapter languages to plug in.

Example: The UI language thus may define an abstract concept `DataMapping` which is then extended by various adapter languages. ◀

1.3.4 *Language Embedding*

Language embedding (Fig. 1.14) enables *heterogeneous* fragments with *independent* languages. It is similar to reuse in that there are two independent languages l_1 and l_2, but instead of establishing references between two homogeneous fragments, we now embed instances of concepts from l_2 in a fragment f expressed with l_1, so

$$\forall c \in Cdn_f \mid lo(co(c.parent)) = l_1 \wedge lo(co(c.child)) = (l_1 \vee l_2) \tag{1.7}$$

Unlike language extension, where l_2 depends on l_1 because concepts in l_2 extends concepts in l_1, there is no such dependency in this case. Both languages are independent. We call l_2 the *embedded* language and l_1 the *host* language. Again, an adapter language l_A that extends l_1 can be used to achieve this, where

$$\exists c \in Cdn_{l_A} \mid lo(c.parent) = l_A \wedge lo(c.child) = l_1 \tag{1.8}$$

Embedding supports syntactic composition of independently developed languages. As an example, consider a state machine language that can be combined with any number of programming languages such as Java or C. If the state machine language is used together with

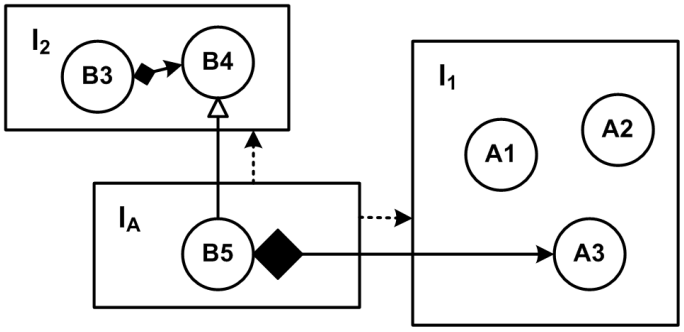

Figure 1.14: Embedding: l_1 and l_2 are independent languages. However, we still want to use them in the same fragment. To enable this, an adapter language l_A is added. It depends on both l_1 and l_2, and uses inheritance and composition to adapt l_1 to l_2 (this is the almost the same structure as in the case of reuse; the difference is that $B5$ now contains $A3$, instead of just referencing it.)

Java, then the guard conditions used in the transitions should be Java expressions. If it is used with C, then the expressions should be C expressions. The two expression languages, or in fact, any other one, must be embeddable in the transitions. So the state machine language cannot depend on any particular expression language, and the expression languages of C or Java obviously cannot be designed with knowledge about the state machine language. Both have to remain independent, and have to be embedded using an adapter language.

When embedding a language, the embedded language must often be extended as well. In the above example, new kinds of expressions must be added to support referencing event parameters. These additional expressions will typically reside in the adapter language as well.

Note that if the state machine language is specifically built to "embed" C expressions, then this is a case of Language Extension, since the state machine language depends on the C expression language.

WebDSL: In order to support queries over persistent data, WebDSL embeds the Hibernate Query Language (HQL) such that HQL queries can be used as expressions. Queries can refer to entity declarations in the program and to variables in the scope of the query. ◄

Pension Plans: The pension workbench DSL embeds a spreadsheet language for expressing unit tests for pension plan calculation rules. ◄

Cross-Cutting Embedding, Metadata A special case of embedding is handling meta data. We define meta data as program elements that are not essential to the semantics of the program, and are typically not handled by the primary model processor. Nonetheless these data need to related to program elements, and, at least from a user's perspective, they often need to be embedded in programs. Since most of them are rather generic, embedding is the right composition mechanism: no dependency to any specific language should be necessary, and the meta data should be embeddable in any language. Example meta data includes:

Documentation should be attachable to any program element, and in the documentation text, other program elements should be referencable.

Traces capture typed relationships between program elements or between program elements and requirements or other documentation ("this element *implements* that requirement")

Presence Conditions in product line engineering describe whether a program element should be available in the program for a given product configuration ("this procedure is only in the program in the *international* variant of the product").

In projectional editors, this meta data can be stored in the program tree, and shown only if switched on. In textual editors, meta data is often stored in separate files, using pointers to refer to the respective model elements. The data may be shown in hovers or views adjacent to the editor itself.

Embedded C: The system supports various kinds of meta data, including traces to requirements and documentation. They are implemented with MPS' *attribute* mechanism which is discussed in part on MPS in section Section **??**. As a consequence of how MPS attributes work, these meta data can be applied to program elements defined in any arbitrary language. ◄

1.3.5 Implementation Challenges and Solutions

Syntax Referencing and Reuse keeps fragments homogeneous. Mixing of concrete syntax is not required. A reference between fragments is usually simply an identifier and does not have its own internal structure for which a grammar would be required. The name resolution phase can then create the actual cross-reference between abstract syntax objects.

Refrigerators: The algorithm language contains cross-references into the hardware language. Those references are simple, dotted names (*a.b.c*). ◄

Example: In the UI example, the adapter language simply introduces dotted names to refer to fields of data structures. ◄

Extension and Embedding requires modular concrete syntax definitions because additional language elements must be "mixed" with programs written with the base language.

Embedded C: State machines are hosted in regular C programs. This works because the C language's `Module` construct contains a collection of `ModuleContents`, and the `StateMachine` concept ex-

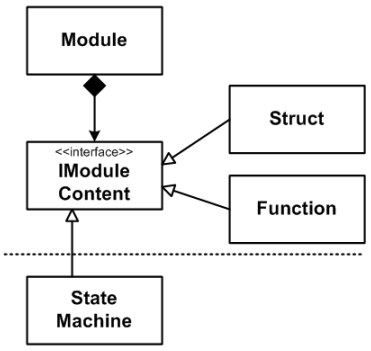

Figure 1.15: The core language (above the dotted line) defines an interface `IModuleContent`. Anything that should be hosted inside a `Module` has to implement this interface, typically from another language. `StateMachines` are an example.

tends the `ModuleContent` concept. This state machine language is designed specifically for being embedded into C, so it can access and extend the `ModuleContent` concept (Fig. 1.15). If the state machine language were reusable with any host language in addition to C, then an adapter language would provide a language concept that adapts C's `IModuleContent` to `StateMachine`, because `StateMachine` cannot directly extend `IModuleContent` — it does now depend on the C language. ◀

Type Systems For referencing, the type system rules and constraints of the referencing language typically have to take into account the referenced language. Since the referenced language is known when developing the referencing language, the type system can be implemented with the referenced language in mind as well.

Refrigerators: In the refrigerator example, the algorithm language defines typing rules for hardware elements (from the hardware language), because these types are used to determine which properties can be accessed on the hardware elements (e.g. a compressor has a property `active` that controls if it is turned on or off). ◀

In case of extension, the type systems of the base language must be designed in a way that allows adding new typing rules in language extensions. For example, if the base language defines typing rules for binary operators, and the extension language defines new types, then those typing rules may have to be overridden to allow the use of existing operators with the new types.

Embedded C: A language extension provides types with physical units (as in 100 kg). Additional typing rules are needed to override the typing rules for C's basic operators (+, -, *, /, etc.). ◀

For reuse and embedding, the typing rules that affect the interplay between the two languages reside in the adapter language.

Example: In the UI example the adapter language will have to adapt the data types of the fields in the data description to the types the UI widgets expect. For example, a combo box widget can only be bound to fields that have some kind of text or enum data type. Since the specific types are specific to the data description language (which is unknown at the time of creation of the UI language), a mapping must be provided in the adapter language. ◀

Transformation In this section we use the terms *transformation* and *generation* interchangably. In general, the term transformation is used if one tree of program elements is mapped to another tree, while gener-

ation describes the case of creating text from program trees. However, for the discussions in this section, this distinction is generally not relevant.

Three cases have to be considered for referencing. The first one (Fig. 1.16) propagates the referencing structure to the target fragments. We call these two transformations single-sourced, since each of them only uses a single, homogeneous fragment as input and creates a single, homogeneous fragment, perhaps with references between them. Since the referencing language is created with the knowledge about the referenced language, the generator for the referencing language can be written with knowledge about the names of the elements that have to be referenced in the other fragment. If a generator for the referenced language already exists, it can be reused unchanged. The two generators basically share naming information.

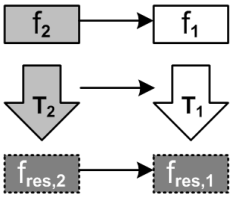

Figure 1.16: Referencing: Two separate, dependent, single-source transformations

Component Architecture: In the types viewpoint, interfaces and components are defined. The types viewpoint is independent, and it is sufficient for the generation of the code necessary for implementing component behaviour: Java base classes are generated that act as the component implementations (expected to be extended by manually written subclasses). A second, dependent viewpoints describes component instances and their connections (it depends on the types viewpoint). A third one describes the deployment of the instances to execution nodes (servers, essentially). The generator for the deployment viewpoint generates code that actually instantiates the classes that implement components, so it has to know the names of those generated (and hand-written) classes. ◄

The second case (Fig. 1.17) is a multi-sourced transformation that creates one single homogeneous fragment. This typically occurs if the referencing fragment is used to guide the transformation of the referenced fragment, for example by specifying target transformation strategies. In this case, a new transformation has to be written that takes the referencing fragment into account. The possibly existing generator for the referenced language cannot be reused as is.

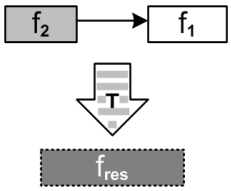

Figure 1.17: A single multi-sourced transformation.

Refrigerators: The refrigerator example uses this case. The code generator that generates the C code that implements the cooling algorithm takes into account the information from the hardware description model. A single fragment is generated from the two input models. The generated code is C-only, so the fragment remains homogeneous. ◄

The third case, an alternative to rewriting the generator, is the use of a preprocessing transformation (Fig. 1.18), that changes the referenced fragment in a way consistent with what the referenced fragment pre-

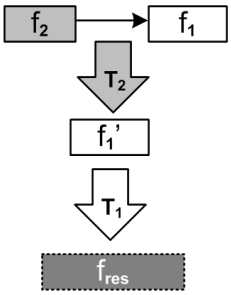

Figure 1.18: A preprocessing transformation that changes the referenced fragment in a way specified by the referencing fragment

scribes. The existing transformations for the referenced fragment can then be reused.

As we have discussed above, language extensions are usually created by defining linguistic abstractions for common idioms of a domain D. A generator for the new language concepts can simply recreate those idioms when mapping L_D to L_{D-1}, a process called assimilation. In other words, transformations for language extensions map a heterogeneous fragment (containing L_{D-1} and L_D code) to a homogeneous fragment that contains only L_{D-1} code (Fig. 1.19). In some cases additional files may be generated, often configuration files.

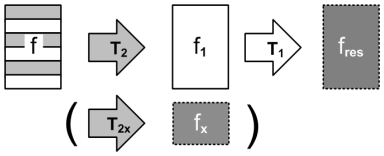

Figure 1.19: Extension: transformation usually happens by assimiliation, i.e. generating code in the host language from code expressed in the extension language. Optionally, additional files are generated, often some configuration files.

Embedded C: State machines are generated down to a function that contains a switch/case statement, as well as enums for states and events. ◀

Sometimes a language extension requires rewriting transformations defined by the base language.

Embedded C: In the data-types-with-physical-units example, the language also provides range checking and overflow detection. So if two such quantities are added, the addition is transformed into a call to a special `add` function instead of using the regular plus operator. This function performs overflow checking and addition. ◀

Language Extension introduces the risk of semantic interactions. The transformations associated with several independently developed extensions of the same base language may interact with each other.

Example: Consider the (somewhat constructed) example of two extensions to Java that each define a new statement. When assimilated to pure Java, both new statements require the surrounding Java class to extend a specific, but different base class. This won't work because a Java class can only extend one base class. ◀

Interactions may also be more subtle and affect memory usage or execution performance. Note that this problem is not specific to languages, it can occur whenever several independent extensions of a base concept can be used together, ad hoc. To avoid the problem, transformations should be built in a way so that they do not "consume scarce resources" such as inheritance links. A more thorough discussion of the problem of semantic interactions is beyond the scope of this paper, and we refer to for details.

TODO: cite

In the reuse scenario, it is likely that both the reused and the context language already come with their own generators. If these generators transform to different, incompatible target languages, no

Figure 1.20: Reuse: Reuse of existing transformations for both fragments plus generation of adapter code

reuse is possible. If they transform to a common target languages (such as Java or C) then the potential for reusing previously existing transformations exists.

There are three cases to consider. The first one, illustrated in Fig. 1.20, describes the case where there is an existing transformation for the reused fragment and an existing transformation for the context fragment — the latter being written with the knowledge that later extension will be necessary. In this case, the generator for the adapter language may "fill in the holes" left by the reusable generator for the referencing language For example, the generator of the context language may generate a class with abstract methods; the adapter may generate a subclass and implement these abstract methods.

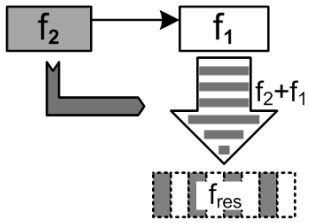

Figure 1.21: Reuse: composing transformations

In the second case, Fig. 1.21, the existing generator for the reused fragment has to be enhanced with transformation code specific to the context language. In this case, a mechanism for composing transformations is needed.

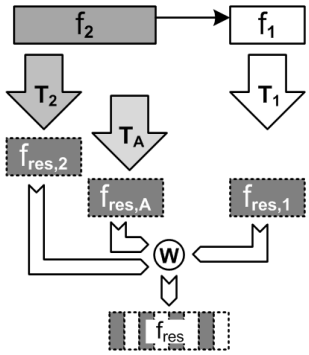

Figure 1.22: Reuse: generating separate artifacts plus a weaving specification

The third case, Fig. 1.22, leaves composition to the target languages. We generate three different independent, homogeneous fragments, and a some kind of weaver composes them into one final, heterogeneous artifact. Often, the weaving specification is the intermediate result generated from the adapter language. An example implementation could use AspectJ.

Embedding An embeddable language may not come with its own generator, since, at the time of implementing the embeddable language, one cannot know what to generate. In that case, when embedding the language, a suitable generator has to be developed. It will typically either generate host language code (similar to generators in the case of language extension) or directly generate to the same target language that is generated to by the host language.

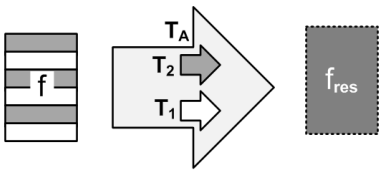

Figure 1.23: In transforming embedded languages, a new transformation has to be written if the embedded language does not come with a transformation for the target language of the host language transformation. Otherwise the adapter language can coordinate the transformations for the host and for the emebedded languages.

If the embeddable language comes with a generator that transforms to the same target language as the embedding language, then the generator for the adapter language can coordinate the two, and make sure a single, consistent fragment is generated. Fig. 1.23 illustrates this case.

Just a language extension, language embedding may also lead to semantic interactions if multiple languages are embedded into the same host language.

1.4 *Concrete Syntax*

A good choice of concrete syntax is important for DSLs to be accepted by the intended user community, particularly if the users are not pro-

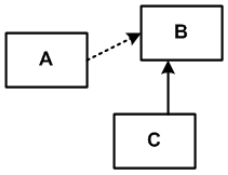

Figure 1.24: Graphical Notation for Relationships

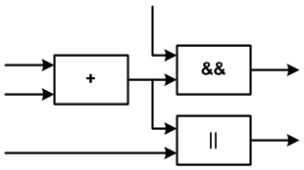

Figure 1.25: Graphical Notation for Flow

grammers. Especially (but not exclusively) in business domains, a DSL will only be successful if and when it uses notations that directly fit the domain — there might even be existing, established notations that should be reused. A good notation makes expression of common concerns simple and concise and provides sensible defaults. It is ok for less common concerns to require a bit more verbosity in the notation.

There are a couple of major classes for DSL concrete syntax[7]: *Textual* DSLs use traditional ASCII or Unicode text. They basically look and feel like traditional programming languages. *Graphical* DSLs use graphical shapes. An important subgroup is represented by those that use box-and-line diagrams that look and feel like UML class diagrams or state machines. *Symbolic* DSLs are textual DSLs with an extended set of symbols, such as fraction bars, mathematical symbols or subscript and superscript. However, there are more options for graphical notations, such as those illustrated by UML timing diagrams or sequence diagrams. *Tables and Matrices* are a powerful way to represent certain kinds of data and can play an important part for DSLs.

[7] Sometimes form-based GUIs or trees views are considered DSLs. We disagree, because this would make any GUI application a DSL.

The perfect DSL tool should support freely combining and integrating the various classes of concrete syntax, and be able to show (aspects of) the same model in different notations. As a consequence of tool limitations, this is not always possible, however. The requirements for concrete syntax are a major driver in tool selection.

When to use which form We do not want to make this section a complete discussion between graphical and textual DSLs — a discussion, that is often heavily biased by previous experience, prejudice and tool capabilities. Here are some rules of thumb. Purely textual DSLs integrate very well with existing development infrastructures, making their adoption relatively easy. They are very well suited for detailed descriptions, anything that is algorithmic or generally resembles (traditional) program source code. Symbolic notations can be considered "better textual", and lend themselves to domains that make heavy use of symbols and special notations. tables are very useful for collections of similarly structured data items, or for expressing how two independent dimensions of data relate. Finally, graphical notations are very good for describing relationships (Fig. 1.24), flow (Fig. 1.25) or timing and causal relationships (Fig. 1.26).

Pension Plans: The pension DSL uses mathematical symbols and notations to express insurance mathematics (Fig. 1.27). A table notation is embedded to express unit tests for the pension plan calculation rules. A graphical projection shows dependencies and specialization relationships between plans. ◄

Embedded C: The core DSLs use a textual notation with some

The Fortress programming language is close to this.

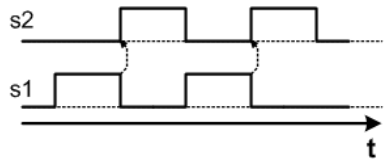

Figure 1.26: Graphical Notation for Causality and Timing

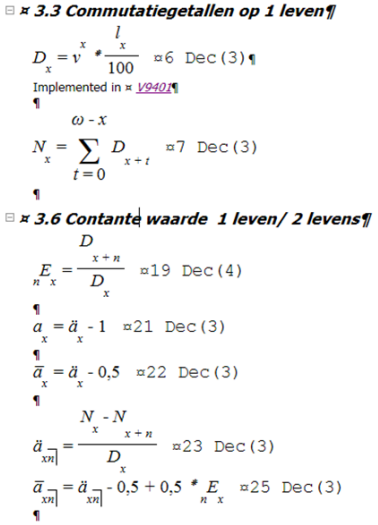

Figure 1.27: Mathematical notations used to express insurance math in the pension workbench.

tabular enhancements, for example, for decision tables (Fig. 1.28). However, as MPS' capability for handling graphical notations will increase, we will represent state machines as diagrams. ◄

```
c/s interface Decider {
  int decide(int x, int y) pre
}

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

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

```
    }
}
```

Figure 1.28: Decision tables use a tabular notation. It is embedded seemlessly into a C program.

Selection of concrete syntax is simple for domain user DSLs if there is an established notation in the domain. The challenge then is to replicate it as closely as possible with the DSL, while cleaning up possible inconsistencies in the notation (since it had not been used formally before). I like to use the term "strongly typed word" in this case[8].

[8] In some cases it is useful to come up with a better notation than the one used historically. This is especially true if the historic notation is Excel :-)

For DSLs targeted at developers, a textual notation is usually a good starting point, since developers are used working with text, and very productive. Tree views, and specific visualizations are often useful, but for editing, a textual notation is a good starting point. It also integrates well with existing development infrastructures.

Embedded C: This was the original reason for developing this system as a set of extensions to C. C is the baseline for embedded systems, and everybody is familiar with it. A textual notation is useful for many concerns in embedded systems. Note that several languages create visualizations on the fly, for example for module dependencies, component dependencies and component instance wirings. The graphviz tool is used here since it provides decent auto-layout. ◄

TODO: ref

There are very few DSLs where a purely graphical notation makes sense. In most cases, textual languages are embedded in the diagrams or tables: state machines have expressions embedded the guards and statements in the actions (Fig. 1.31); component diagrams use text for specifications of operations in interfaces, maybe even with expressions for preconditions; block diagrams use a textual syntax for the implementation/parametrization of the blocks (Fig. 1.29); tables may embed textual notations in the cells (Fig. 1.30). .

A text box where textual code can be entered without language support should only be used as a last resort. Instead, a textual notation, with additional graphical visualizations should be used

In my consulting practice, I almost always start with a textual notation and try to stabilize language abstractions based on this notation. Only then will I engage in a discussion about whether a graphical notations on top of the textual one is necessary. Often it is not, and if it is, we have avoided iterating the implementation of the graphical editor implementation, which, depending on the tooling, can be a lot of work.

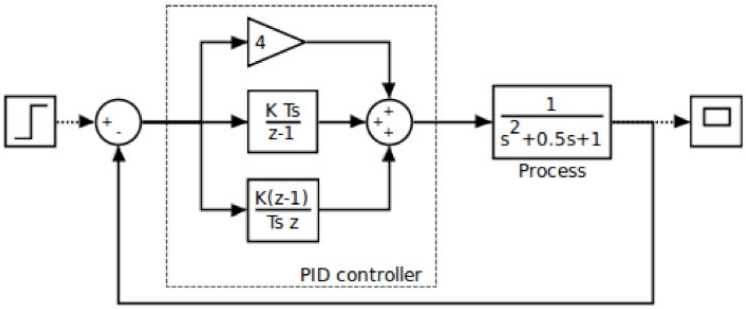

Figure 1.29: Block diagrams built with the Yakindu modeling tools. A textual DSL is used to implement the behaviour in the blocks. While the textual DSL is not technically integrated with the graphical notation (separate viewponts), semantic integration is provided.

Also note that initially, domain users prefer a graphical notation, because of the perception that things that are described graphically are simple(r) to comprehend Not really. However, what is most important regarding comprehensibility is the alignment of the domain concepts with the abstractions in the language. A well-designed textual notation can go a long way. Also, textual languages are more productive once the learning curve has been overcome.

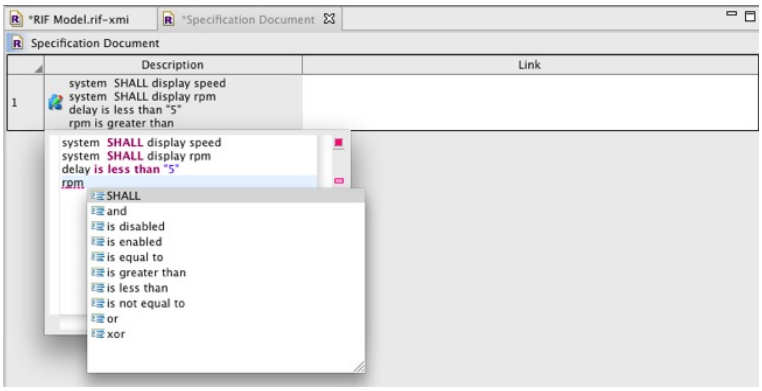

Figure 1.30: The Yakindu Requirements tools integrates a textual DSL for formal requirements specification. An Xtext DSL is syntactically integrated into a table view. Semantic integration is provided as well.

Multiple Notations For projectional editors it is possible to define several notations for the same abstract syntax. By changing the projection rules, existing programs can be shown in a different way. This removes some of the burden of getting it right initially, because the notation can be adapted after the fact. In general, for the concrete syntax of a DSL writability is often more important than readability, because additional read-only representations can always be derived automatically.

Embedded C: For state machines, the primary syntax is textual. However, a tabular notation is supported as well. The projection can be changed as the program is edited, rendering the same state machine textually or as a table. As mentioned above, a graphical notation will be added in the future. ◀

Refrigerators: The refrigerator DSL uses graphical visualizations to render diagrams of the hardware structure, as well as a graphical state charts representing the underlying state machine. ◀

Relationship to Hierarchical Domains Domains at low D are most likely best expressed with a textual or symbolic concrete syntax. Obvious examples include programming languages at D_0. Mathematical expressions, which are also very dense and algorithmic, use a symbolic notation. As we progress to higher Ds, the concepts become more and more abstract, and as state machines and block diagrams illustrate,

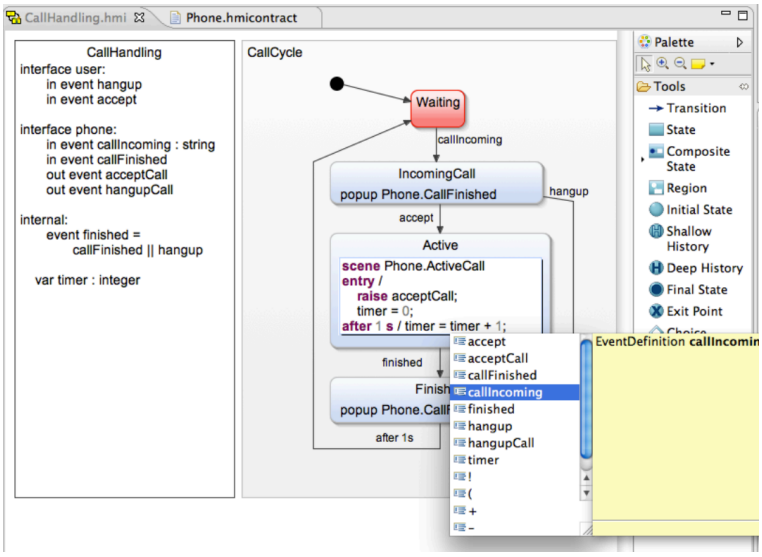

Figure 1.31: The Yakindu State Chart Tools support the use of Xtext DSLs in actions and guard conditions of state machines, mixing textual and graphical notations. The DSL can even be exchanged, to support domain specific action languages.

graphical notations become useful. However, these two notations are also a good example of language embedding since both of them require expressions: state machines in guards and actions (Fig. 1.31), and block diagrams as the implementation of blocks (Fig. 1.29 and Fig. 1.6). Reusable expression languages should be embedded into the graphical notations. In case this is not supported by the tool, viewpoints may be an option. One viewpoint could use a graphical notation to define coarse-grained structures, and a second viewpoint uses a textual notation to provide "implementation details" for the structures defined by the graphical viewpoint[9].

Embedded C: As the graphical notation for state machines will become available, C expression language that is used in guard conditions for transitions will be usable as labels on the transition arrows. In the table notation for state machines, C expressions can be embedded in the cells as well. ◄

[9] Not every tool can support every (combination of) form of concrete syntax, so this aspect is limited by the tool, or drives tool selection.

2
Process Issues

Software development with DSLs requires a compatible development process. A lot of what's required is similar to what's required for any other reusable artifact such as a framework. A workable process must be established between those who build the reusable artifact and those who use it. Requirements have to flow in one direction, and a finished, stable, tested and document tool in the other. Also, using DSLs is a bit of a change for all involved, especially the domain experts. In this chapter we provide some guidelines.

2.1 DSL Development

2.1.1 Requirements for the Language

How do you find out what your DSL should express? What are the relevant abstractions and notations? This is a non-trivial issue, in fact, it is one of the key issues in using DSLs. It requires a lot of domain expertise, thought and iteration. The core problem is that you're trying to not just understand one problem, but rather a *class* of problems. Understanding and defining the extent and nature of this class of problems can be non-trivial. There are several typical ways of how to get started.

If you're building a technical DSL, the source for a language is often an existing framework, library, architecture or architectural pattern. The knowledge often already exists, and building the DSL is mainly about formalizing the knowledge: defining a notation, putting it into a formal language, and building generators to generate parts of the (potentially complex) implementation code. In the process, you often also want to put in place reasonable defaults for some of the framework features, thereby increasing the level of abstraction and making framework use easier.

In case of business domain DSLs, you can often mine the existing (tacit) knowledge of domain experts. In domains like insurance, science or logistics, domain experts are absolutely capable of precisely expressing domain knowledge. They do it all the time, often using Excel or Word. They often have a "language" to express domain concerns, although it is usually not formal, and there's no tool support. In this context, your job is to provide formality and tooling. Similar to domain knowledge, other domain artifacts can also be exploited: for example, hardware structures or device features are good candidates for abstractions in the respective domains.

In both previous cases, it is pretty clear how the DSL is going to look like; discussions are about details, notation, how to formalize things, viewpoints, partitioning and the like (note that those things can be pretty non-trivial, too!).

One of my most successful approaches in this case is to build strawmen: trying to understand something, factor it into some kind of regular structure, and then re-explain that structure back to the stakeholders.

However, in the remaining third case, however, we are not so lucky. If no domain knowledge is easily available, we have to do an actual domain analysis, digging our way through requirements, stakeholder "war stories" and existing applications. Co-evolving language and concepts (see below) is a successful technique especially in this case.

For your first DSL, try to catch case one or two. Ideally, start with case one, since the people who build the DSLs and supporting tools are often the same ones as the domain experts — software architects and developers.

2.1.2 *Iterative Development*

Some people use DSLs as an excuse to do waterfall again. They spend months and months developing languages, tools, and frameworks. Needless to say, this is not a very successful approach. You need to iterate when developing the language.

Start by developing some deep understanding of a small part of the domain for which you build the DSL. Then build a little bit of language, build a little bit of generator and develop a small example model to verify what you just did. Ideally, implement all aspects of the language and processor for each new domain requirement before focusing on new requirements.

Especially newbies to DSLs tend to get languages and meta models wrong because they are not used to think meta. You can avoid this pitfall by immediately trying out your new language feature by building an example model and developing a compatible generator.

It is important that the language approaches some kind of stable state over time (Fig. 2.1). As you iterate, you will encounter the following situation: domain experts express requirements that may sound inconsistent. You add all kinds of exceptions and corner cases to the language. You language grows in size and complexity. After a num-

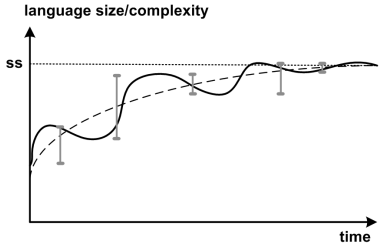

Figure 2.1: Iterating towards a stable language over time

ber of these exceptions and corner cases, ideally the language designer will spot the systematic nature behind these and refactor the language to reflect this deeper understanding of the domain. Language size and complexity is reduced. Over time, the amplitude of these changes in language size and complexity (error bars in Fig. 2.1) should become smaller, and the language size and complexity should approach a stable level (*ss* in Fig. 2.1).

2.1.3 Co-evolve concepts and language

In cases where you do a real domain analysis, i.e. when you have to find out which concepts the language shall contain, make sure you evolve the language in real time as you discuss the concepts.

Defining a language requires formalization. It requires becoming very clear — formal! — about the concepts that go into the language. In fact, building the language, because of the need for formalization, helps you become clear about the concepts in the first place. Language construction acts as a catalyst for understanding the domain! I recommend actually building a language in real time as you analyze your domain.

To make this feasible, your DSL tool needs to be lightweight enough so support language evolution during domain analysis workshops. Turnaround time should be minimal to avoid overhead.

2.1.4 Let people do what they are good at

DSLs offers a chance to let everybody do what they are good at. There are several clearly defined roles, or tasks, that need to be done. Let met point out two, specifically.

Experts in a specific target technology can dig deep into the details of how to efficiently implement, configure and operate that technology. They can spend a lot of time testing, digging and tuning. Once they found out what works best, they can put their knowledge into generator templates, efficiently spreading the knowledge across the team. For the latter task, they will collaborate with generator experts and language designer — our second example role.

The language designer works with domain experts to define abstractions, notations and constraints to accurately capture domain knowledge. The language designer also works with the architect and the platform experts in defining code generators or interpreters. For the role of the language designer, be aware that there needs to be some kind of predisposition in the people who do it: not everybody is good at "thinking meta", some people are simply more skewed towards concrete work. Make sure you use "meta people" to do the "meta work".

There's also a flip side here: you have to make sure you actually do have people on your team who are good at language design, know

about the domain and understand target platforms. Otherwise the MD* approach will not deliver on its promises.

2.1.5 Domain Users vs. Domain Experts

When building business DSLs, people from the domain can play two different roles. They can participate in the domain analysis and the definition of the DSL itself. On the other hand, they can use the DSL to express specific domain knowledge.

It is useful to distinguish these two roles explicitly. The first role (language definition) must be filled by a domain expert. These are people who have typically been working in the domain for a long time, maybe in different roles, who have a deep understanding of the relevant concepts and they are able to express them precisely, and maybe formally.

The second group of people are the domain users. They are of course familiar with the domain, but they are typically not as experienced as the domain experts

This distinction is relevant because you typically work with the domain experts when defining the language, but you want the domain users to actually work with the language. If the experts are too far ahead of the users, the users might not be able to "follow" along, and you will not be able to roll out the language to the actual target audience.

Hence, make sure that when defining the language, you actually cross-check with real domain users whether they are able to work with the language.

2.1.6 DSL as a Product

The language, constraints, interpreters and generators are usually developed by one (smaller) group of people and used by another (larger) group of people. To make this work, consider the metaware a product developed by one group for use by another. Make sure there's a well defined release schedule, development happens in short increments, requirements and issues are reported and tracked, errors are fixed reasonably quickly, there is ample documentation (examples, examples, examples!) and there's support staff available to help with problems and the unavoidable learning curve. These things are critical for acceptance.

A specific best practice is to exchange people: from time to time, make application developers part of the generator team to appreciate the challenges of "meta", and make meta people participate in actual application development to make sure they understand if and how their metaware suits the people who do the real application development.

2.1.7 *Documentation is still necessary*

Building DSLs and model processors is not enough to make the approach successful. You have to communicate to the users how the DSL and the processors work. Specifically, here's what you have to document: the language structure and syntax, how to use the editors and the generators, how and where to write manual code and how to integrate it as well as platform/framework decisions (if applicable).

Please keep in mind that there are other media than paper. Screencasts, videos that show flipchart discussions, or even a regular podcast that talks about how the tools change are good choices, too.

Also keep in mind that hardly anybody reads reference documentation. If you want to be successful, make sure the majority of your documentation is example-driven or task-based.

2.2 *Using DSLs*

2.2.1 *Reviews*

A DSL limits the user's freedom in some respect: they can only express things that are within the limits of DSLs. Specifically, low-level implementation decisions are not under a DSL user's control because they are handled by the model generator or interpreter.

However, even with the nicest DSL, users can still make mistakes, the DSL users can still misuse the DSL (the more expressive the DSL, the bigger this risk). So, as as part of your development process, make sure you do regular model reviews. This is critical especially to the adoption phase when people are still learning the language and the overall approach.

Reviews are easier on DSL level than on code level. Since DSL programs are more concise than their equivalent specification in GPL code, reviews become more efficient.

If you notice recurring mistakes, things that people do in a "wrong" way regularly, you can either add a constraint check that detects the problem automatically, or (maybe even better) consider this as input to your language designers: maybe what the users expect is actually correct, and the language needs to be adapted.

2.2.2 *Compatible Organization*

Done right, MD* requires a lot of cross-project work. In many settings the same metaware will be used in several projects or contexts. While this is of course a big plus, it also requires, that the organization is able to organize, staff, schedule and pay for cross-cutting work. A strictly

project-focused organization has a very hard time finding resources for these kinds of activities. MD* is very hard to do effectively in such environments.

Make sure that the organizational structure, and the way project cost is handled, is compatible with cross-cutting activities. You might want to take a look at the Open Source communities to get inspirations of how to do this.

2.2.3 *Domain Users Programming?*

Domain users aren't programmers. But they are still able to formally and precisely describe domain knowledge. Can they actually do this alone?

In many domains, usually those that have a scientific or mathematical touch, they can. In other domains you might want to shoot for a somewhat lesser goal. Instead of expecting domain users and experts to independently specify domain knowledge using a DSL, you might want to pair a developer and a domain expert. The developer can help the domain expert to be precise enough to "feed" the DSL. Because the notation is free of implementation clutter, the domain expert feels much more at home than when staring at GPL source code.

Initially, you might even want to reduce your aspirations to the point where the developer does the DSL coding based on discussions with domain users, but then showing them the resulting model and asking confirming or disproving questions about it. Putting knowledge into formal models helps you point out decisions that need to be made, or language extensions that might be necessary.

Executing the program, by generating code or running some kind simulator can also help domain users understand better what has been expressed with the DSL.

If you're not able to teach a business domain DSL to the domain users, it might not necessarily be the domain users' fault. Maybe your language isn't really suitable to the domain. If you encounter this problem, take it as a warning sign and take a close look at your language.