MARKUS VOELTER, VOELTER@ACM.ORG

DSL ENGINEERING

1
DSLs and Requirements

1.1 What is Requirements Engineering

Wikipedia defines a requirements as follows: *a requirement is a singular documented need of what a particular product or service should be or perform.* Wiktionary says: *specifies a verifiable constraint on an implementation that it shall undeniably meet or (a) be deemed unacceptable, or (b) result in implementation failure, or (c) result in system failure.* In our own words we would probably define a requirement as a statement about *what a system should do, and with which quality attributes, without presupposing a specific implementation.* Requirements are supposed to tell the programmers what the system they are about to implement, should do. They are a means of communicating from humans (people who know what the system should do) to other humans (those that have to implement it).

Of course, as well all know, there are a number of challenges in this:

- Those who implement the system may have a different background than those who write them, making misunderstandings likely.
- Those who write the requirements may not actually really know what they want the system to do. Requirements change and learning about these and writing them down is hard.
- Usually requirements are written in plain English (or whatever language you prefer). Writing things down precisely and completely in a non-formal language is next to impossible.

Traditional requirements (documents) are a means of communication between people. However, in the end this is not really true. In an ideal world, the requirements (in the brain of the person who writes them down) should be communicated directly to the computer, without the intermediate programmer. If we look at the problem in this

way, requirements now become a formal, computer-understandable document.

Wikipedia has a nice list of characteristics that requirements should posses:

Cohesive The requirement addresses one and only one thing.

Complete The requirement is fully stated in one place with no missing information.

Consistent The requirement does not contradict any other requirement and is fully consistent with all authoritative external documentation.

Atomic The requirement is atomic, i.e., it does not contain conjunctions. E.g., "The postal code field must validate American and Canadian postal codes" should be written as two separate requirements: (1) "The postal code field must validate American postal codes" and (2) "The postal code field must validate Canadian postal codes".

Traceable The requirement meets all or part of a business need as stated by stakeholders and authoritatively documented.

Current The requirement has not been made obsolete by the passage of time.

Feasible The requirement can be implemented within the constraints of the project.

Unambiguous The requirement is concisely stated without recourse to technical jargon, acronyms (unless defined elsewhere in the Requirements document), or other esoteric verbiage. It expresses objective facts, not subjective opinions. It is subject to one and only one interpretation. Vague subjects, adjectives, prepositions, verbs and subjective phrases are avoided. Negative statements and compound statements are prohibited.

Mandatory The requirement represents a stakeholder-defined characteristic the absence of which will result in a deficiency that cannot be ameliorated. An optional requirement is a contradiction in terms.

Verifiable The implementation of the requirement can be determined through one of four possible methods: inspection, demonstration, test or analysis.

If requirements are written as pure prose text, then making sure all these characteristics are met, boils down mostly to a manual review process. Of course, this is tedious and error prone (often exacerbated

through over-specification even though the prose text isn't a specification at all), so it's often not done rigorous enough or not at all and requirements documents end up in the sorry state we all know.

If you want to get one step better, you use somewhat controlled language: words like "must", "may", or "should" have a well defined meaning and are used consciously. Using tables and, to some extent, state machines, is also a good way to make some of the data a bit more unambiguous. To manage large sets of requirements, tools like DOORS or RequisitePro are used: they allow the unique naming of requirements, as well as the expression of relationships and hierarchies among requirements. However, the requirements themselves are still expressed as plain text, so the fundamental problems mentioned above are not improved significantly.

In this chapter we will give you some ideas and examples on how this situation can be improved with DSLs.

1.2 *Requirements vs. Design vs. Implementation*

Traditionally, we try to establish a clear line between requirements, architecture and design, and implementation. For example, consider the following:

Requirement The system shall be 100% reliable.

Design Use hot-standby and failover to continue service if something breaks.

Implementation ...all the code that is necessary to implement the design above.

We make this distinction because we want to establish different roles in the software engineering process. For example, product management writes the requirements, a systems architect comes up with the architecture and design, and then a programmer writes the actual code. Different organizations might be involved: the OEM writes the requirements, a systems integrator does the architecture, and some cheap outsourcing company does the coding. In such a scenario it is important to draw precise boundaries between the activities, of course.

However, in some sense the boundaries are arbitrary. We could just as well do the following:

Requirement The system shall be 100% reliable by using hot-standby and failover to continue service if something breaks.

Design We use two application servers running on two hosts, using XYZ messaging queue as a replication engine for the hot-standby. We use a watchdog for detecting if the primary machine breaks.

Implementation ...all the code that is necessary to implement the design above.

From software development we know that it is very hard to get requirements right. In the real world, you have to elaborate on the requirements incrementally: you write some requirements, then you write a prototype and check if the requirements make sense, then you refine the requirements, write a (maybe more detailed) prototype, and so on.

In systems engineering this approach is also very well established. For example, when satellites are built, the scientists come up with initial scientific requirements, for example, regarding the resolution a satellite-based radar antenna looking at the earth should have. Let's look at some of the consequences:

- A given resolution requires a specific size of the antenna, and a specific amount of energy being sent out. (Actually, the two influence each other prohibiting a strict hierarchy.)
- A bigger antenna results in a heavier satellite, and more radar energy requires more solar panel area - increasing the size and weight again.
- At some point, the size and weight of the satellite cannot be further increased, because a given launch vehicle reaches its limits - a different launch vehicle might be required.
- A bigger launch vehicle might be much more expensive, or you might have to change the launch provider. For example, you might have to use a Soyuz instead of an Ariane.
- A Soyus launched Baikonur cannot reach the same orbits as an Ariane launched from Courou. As a consequence, the satellite might be "further away" from the area you want to inspect with your radar, neglecting the advantages gained by the bigger antenna.
- Instead of a bigger antenna, you might want to use several smaller ones, maybe even on different satellites.

Now this has just looked at size and weight. Similar problems exist with heat management, pointing accuracy, propulsion, etc. It is plainly impossible to write requirements, throw them over the fence, and have somebody build the satellite.

So what do the systems engineers do? They come up with a model of the satellite. Using mathematic formulas, they describe how the different properties discussed above relate. These might be approximations or based on past experience - after all, the real physics can be quite complex. And then they run a so-called trade-off analysis. In other words, they change the input values until a workable compromise is reached. Usually this is a manual process, but sometimes parts of it can be automated.

This shows three things. First, requirements elicitation is incremental. Second, models can be a big help to precisely specify requirements and then "play" with them. And third, the boundary between requirements and design is blurred, and the two influence each other.

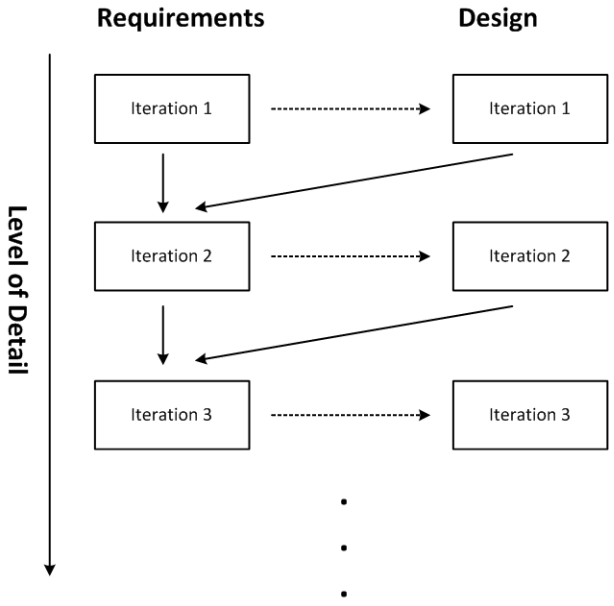


Figure 1.1: Requirements and Design influence each other and are thus best done iteratively, and in parallel

1.3 *Using DSLs for Requirements Engineering*

So here is the approach for using DSLs we suggest: identify a couple of core areas of the to-be-built system that lend themselves to specification with a formal language. Then develop a DSL to express these areas and use them to describe the system. The rest of the system - i.e. the areas for which a DSL-based description makes no sense - is described textually, with the usual tools. We will discuss the integration between the textual requirements and the DSL-based requirements below.

Once a suitable DSL has been found and implemented, those people who have the requirements in mind can directly express them - the lossy human-to-human communication is no longer a problem. Various constraint checks, tests and simulations can be used to have the requirements owners "play" with the requirements models to see if they really express what they had in mind.

At this point you may say: well, that's not really requirements engineering anymore, that's just plain old DSL usage. We agree to some extent. However, these DSLs might not be complete enough to support execution. In many cases they are, but in some cases checking, analysis and/or simulation might be all that's required. More detailed models may be required to make them executable.

Of course there is one significant caveat: we first have to build this DSL. So how do we go about that? We could have somebody write prose requirements and hand them over to the DSL developer ... back to square one!

There is a much better approach, though. Since today's language workbenches support extremely rapid prototyping, you can actually build the DSLs interactively with the requirements owner. Since you're not capturing the specific requirements but rather try to describe how specific requirements are described, you are performing what's called a domain analysis: you try to understand the degrees of freedom in the domain to be able to represent it with the DSL. This is similar to what has been done with analysis models (back in the day ...). However, instead of drawing UML analysis diagrams, you capture the domain into a language definition. These are if you will "executable analysis models", since you can always turn around and have the requirements owner try to express specific requirements with the DSL you're building, verifying the suitability of the DSL.

Here is the process we use:

1. Have the requirements owner explain some particular aspect of the domain.

2. Try to understand that aspect and change your DSL so it can express that aspect.

3. Have the requirements owner try to express a couple of specific, but representative, requirements with the DSL.

4. You'll run into problems, some things cannot be expressed with the DSL.

5. Go back to 1. and reiterate. A complete iteration should take no more than 60 minutes.

6. After half a day, stop working with the requirements owner and clean up/refactor the DSL.

7. Start another of the language design sessions with the requirements owner and iterate – over time, you should get closer to *the* DSL for the domain.

1.4 Integration with Plain Text Requirements

Using DSLs, you will not be able to describe all requirements of a systems. There will always be aspects that cannot be formalized, or that are so specific, that the effort of building a DSL does not pay off. You have to find some way of integrating plain text requirements with DSL code. Here are some ideas of how you can do this.

1.4.1 Embedding DSL code in a requirements tool

As part of the VERDE research project, Eclipse-based tooling for requirements engineering is developed. This includes a "classical" requirements engineering tool in which textual requirements are classified, structured and put into relationships with each other. The requirements structure is represented as an EMF model. In addition to plain text, requirements can have parameters with well-defined types (strings, integers).

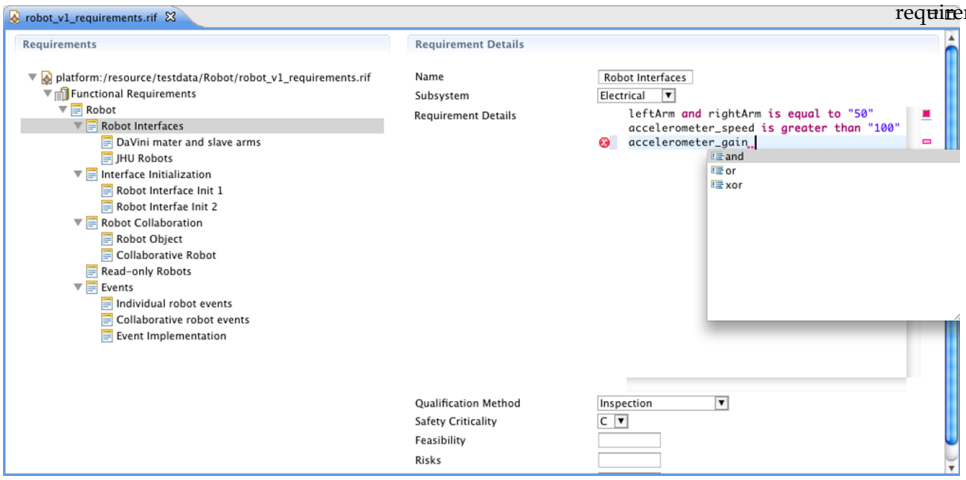


Figure 1.2: An Xtext DSL embedded in a requirements engineering tool

In addition, the type of an attribute can also be a DSL - in other words, it is possible to associate DSL programs with requirements. As

figure 1.2 shows, these DSL fragments are integrated with the UI of the requirements management tool.

1.4.2 *Integrating Plain Text and Models*

If DSLs are used to describe requirements, these can obviously annotated with comments, providing some form of integration with non-formal requirements. However, this is a bad solution for various reasons. In most cases it is better to keep the text and the models separate, and then refer from one to the using actual links (not just text references).

Using the Xdoc extension to Xtext

In MPS, one can use the generic documentation language and embed references

1.5 *Requirements Traceability*

This means: being able to trace how requirements lead to design, implementation and test artifacts. This allows you to see if and how requirements have influenced the design, whether they are implemented and whether tests for the requirements exist. This is especially important in safety-critical environments where you have to "prove" that certain requirements have been taken care of adequately.

Traceability links are basically "pointers" from requirements to other artifacts. In a world where requirements as well as design, implementation and test artifacts are all model-based, establishing these pointers becomes trivial. In mixed environments with many different tools built on many different foundations, this can become arbitrarily complicated.

In this section we want to show you two approaches to traceability.

1.5.1 *Traceability Framework for Eclipse*

The VERDE project mentioned above also develops a traceability framework based on Eclipse. Various types of traceability links can be defined after which they can be used to establish links between arbitrary EMF based models. In addition to the generic EMF-based version, tool adapters can also be plugged in. These provide specific support for links from/into a number of specific tools or formats, for example plain text files, non-EMF UML tools or AUTOSAR models.

The links are kept external to the actual models, so no modifications to existing meta models or languages are required. This is also the reason why non-EMF artifacts can be integrated, albeit requiring the special tool adapter.

1.5.2 *Traceability in MPS*

MPS supports language annotations. Arbitrary models - independent of the language - can be annotated with any other data. One use case of this approach is the annotation with traceability links, as shown in figure 1.3.

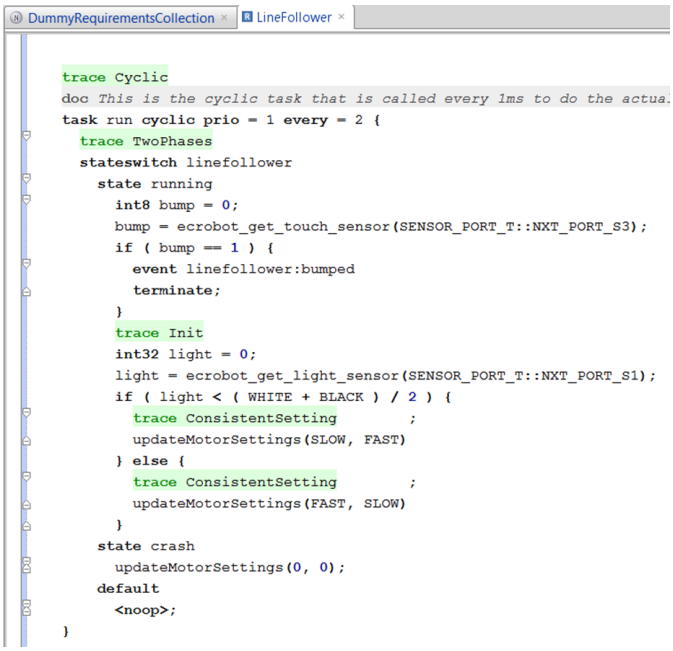


Figure 1.3: Traceability annotations on a C-like language

A context menu action adds a new link to any other model element: Control-Space allows the selection of one or more requirements to which to point. The requirements themselves are imported into an MPS model from a requirements engineering tool via RIF files.

Alternatively it would also be possible to create tooling that points from requirements to implementation (as opposed to the opposite direction, shown above).

1.6 *Summing Up*

This chapter describes how DSLs play into requirements engineering. Let us wrap up by revisiting the desirable characteristics for requirements as outlined by Wikipedia, and see how DSLs can improve the situation. The following lists only those characteristics for which DSLs make a difference.

Consistent Consistency is enforced by the language. If the DSL is

crafted correctly, no inconsistent programs can be expressed.

Traceable As shown in the last section above, traceability is at least simplified.

Feasible Specific requirements are checked for feasibility by being expressible with the DSL - they are within the scope of what the DSL, hence, the domain for which we write the requirements - is intended.

Unambiguous A description of requirements - or application functionality in general - with a DSL always unambiguous, provided the DSL has well-defined semantics.

Verifiable Constraints, tests, verification or simulation can be used to verify that the requirements regarding various properties. Inspection and review is simplified, because DSL programs are less verbose than implementation code, and clearer than prose.

2
DSLs and Software Architecture

2.1 What is Software Architecture

2.1.1 Definitions

Software architecture has many definitions by various groups of people. Here are a few. Wikipedia defines software architecture as *The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships between them.*

This is a classic definition that refers to the structure of a software system, observable by analyzing an existing system. It emphasizes the structure (as opposed to the behavior) and focusses on the coarse grained building blocks found in the system. A definition by Boehm builds on this:

A software system architecture comprises

- *A collection of software and system components, connections, and constraints.*
- *A collection of system stakeholders' need statements.*
- *A rationale which demonstrates that the components, connections, and constraints define a system that, if implemented, would satisfy the collection of system stakeholders' need statements.*

In addition to the structure, this definition also emphasizes the relevance of the stakeholders and their needs, it ties the structure of the system to what the system is required to do.

Hayes-Roth introduce another concern: *The architecture of a complex software system is its style and method of design and construction.* Instead of looking at the structures, they emphasize that there are different architectural styles and they emphasize the "method of design and construction". Eoin Woods takes it one step further: *Software architecture is the set of design decisions which, if made incorrectly, may cause your*

project to be cancelled. He emphasizes the design decisions that lead to a given system. So he doesn't look at the system as a set of structures, but rather considers the architecture as a process - the design decisions - that lead to a given system.

Let us propose another definition: *Software architecture is everything that needs to be consistent throughout a software system*. This definition is useful because it includes structures and behavior, it doesn't say anything about coarse-grained vs. detailed (after all, a locking protocol is possibly a detail, but it is important to implement it consistently) and it implies that there needs to be some kind of process or method to achieve the consistency.

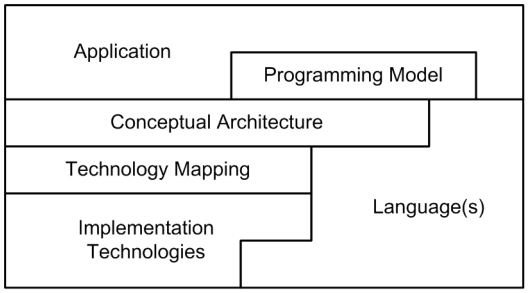


Figure 2.1: Conceptual structure of a software architecture

Figure 2.1 looks at software architecture in more detail. At the center of the diagram you can see the conceptual architecture. It defines architectural concepts from which systems are built - we will come back to this in the next section. Examples could include: task, message, queue, component, port, or replicated data structure. Note that these concepts are independent of specific implementation technologies: it is the technology mapping that defines how the architecture concepts are implemented with specific technologies. Separating the two has the advantage that the conceptual discussions aren't diluted by what certain technologies may or may not be able to do. The decision what technology to map to is then driven by non-functional concerns, i.e. whether and how a given technology can provide the required quality of service.

The programming model is the way how the architectural concepts are implemented in a given programming language. Ideally, the programming model should not change as long as the architectural concepts remain stable - even if the technology mapping or the implementation technologies change. Applications are implemented by instantiating the architectural concepts, and implementing them based on the programming model.

We want to reemphasize the importance of the conceptual architecture. When asking people about the architecture of systems, one often

gets answers like: "it's a Web service architecture", or "it's an XML architecture" or "it's a JEE architecture". Obviously, all this conveys is that a certain technology is used. When talking about architectures per se, we want to talk about architectural concepts, how they relate to each other. In many cases, fundamental architectural concepts can and should be used to explain these. Only in a second step, a mapping to one or more technologies should be discussed.

Here are some of these fundamental architectural concepts:

Modularize Break big things down into smaller things, so they can be understood (and potentially reused) more easily. Examples: Procedures, Classes, Components, Services, User Stories.

Encapsulate Hide the innards of a module so they can be changed without affecting clients. Examples: Private Members, Facade Pattern, Components, Layers/Rings/Levels.

Contracts Describe clearly the external interface of a module. Examples: Interfaces, Pre/Post Conditions, Protocol State Machines, Message Exchange Patterns, Published APIs.

Decoupling Reduce dependencies in time, data structure or contention. Examples: Message Queues, Eventual Consistency, Compensating Transactions.

Isolate Crosscuts Encapsulate handling of cross-cutting concerns. Examples: Aspect Orientation, Interceptors, Application Servers, Exception Handling.

Separation of Concerns Separate different responsibilities into different modules.

... more to be added ... TODO

2.2 *Architecture DSLs*

2.2.1 *Conceptual Architecture and DSLs*

So how does all of this relate to DSLs? An Architecture DSL (ADSL) is a language that expresses a system's architecture directly with "directly" meaning that the language's abstract syntax contains constructs for all the ingredients of the conceptual architecture. The language can hence be used to describe a system on architectural level such that no 3GL/GPL source code is required. Code generation is used to generate representations of the application architecture in the implementation language(s), automating the technology mapping (once the decisions

about the mapping have been made manually, of course). Finally, the programming model is defined with regards to the generated code plus possibly additional frameworks.

We want to (re)emphasize an important point: we do not advocate the definition of a generic, reusable language such as the various ADLs, or UML (see below). Based on our experience, the approach works best if you define the ADSL in real time as you understand, define and evolve the conceptual architecture of a system! The process of defining the language actually helps the architecture/development team to better understand, clarify and refine the architectural abstractions as the language serves as a (formalized) ubiquitous language that lets you reason and discuss about the architecture.

2.2.2 *An Example ADSL*

This section contains an example of an Architecture DSL Markus has implemented for a real system together with a customer from the domain of airport management systems on one of his consulting "gigs".

The customer decided they wanted to build a new flight management system. Airlines use systems like these to track and publish information about whether airplanes have landed at airports, whether they are late, the technical status of the aircraft, etc. The system also populates the online-tracking system on the Web and information monitors at airports. This system is in many ways a typical distributed system: there is a central data center to do some of the heavy number crunching, but there's additional machines distributed over relatively large areas. Consequently you cannot simply shut down the whole system, introducing a requirement to be able to work with different versions of parts of the system at the same time. Different parts of the system will be built with different technologies: Java, C++, C#. This is not an untypical requirement for large distributed systems either. Often you use Java technology for the backend, and .NET technology for a Windows front end. They had decided that the backbone of the system would be a messaging infrastructure and they were evaluating different messaging tools for performance and throughput.

When Markus arrives they briefed him about all the details of the system and the architectural decisions they had already made, and then asked me whether all this made sense. It quickly turned out that, while they knew many of the requirements and had made specific decisions about certain architectural aspects, they didn't have a well-defined conceptual architecture. And it showed: when the team were discussing about their system, they stumbled into disagreements about architectural concepts all the time because they had no language for the architecture. Hence, we started building a language. We would actually build the grammar, some constraints and an editor while we

discussed the architecture in a two-day workshop.

We (i.e., the customer and Markus) started with the notion of a component. At that point the notion of components is defined relatively loosely, simply as the smallest architecturally relevant building block, a piece of encapsulated application functionality. We also assume that components can be instantiated, making components the architectural equivalent to classes in OO programming. To enable components to interact, we also introduce the notion of interfaces, as well as ports, which are named communication endpoints typed with an interface. Ports have a direction (provides, requires) as well as a cardinality.

Figure 2.2: Components, required ports and provided ports

```
component DelayCalculator {
  provides aircraft: IAircraftStatus
  provides managementConsole: IManagementConsole
  requires screens[0..n]: IInfoScreen
}
component Manager {
  requires backend[1]: IManagementConsole
}
component InfoScreen {
  provides default: IInfoScreen
}
component AircraftModule {
  requires calculator[1]: IAircraftStatus
}
```

It is important to not just state which interfaces a component provides, but also which interfaces it requires because we want to be able to understand (and later: analyze with a tool) component dependencies. This is important for any system, but especially important for the versioning requirement. We then looked at instantiation. There are many aircraft, each running an AircraftModule, and there are even more InfoScreens. So we need to express instances of components. Note that these are logical instances. Decisions about pooling and redundant physical instances had not been made yet. We also introduce connectors to define actual communication paths between components (and their ports).

At some point it became clear that in order to not get lost in all the components, instances and connectors, we need to introduce some kind of namespace. It became equally clear that we'd need to distribute things to different files - the tool support ought to make sure that editor functionality such as Go To Definition and Find References

Figure 2.3: Component instances and their connectors

```
instance dc: DelayCalculator
instance screen1: InfoScreen
instance screen2: InfoScreen
connect dc.screens to (screen1.default, screen2.default)
```

still works.

Figure 2.4: TODO

```
namespace com.mycompany {
  namespace datacenter {
    component DelayCalculator { … }
    component Manager { … }
  }
  namespace mobile {
    component InfoScreen { … }
    component AircraftModule { … }
  }
}
```

It is also a good idea to keep component and interface definition (essentially: type definitions) separate from system definitions (connected instances), so we introduced the concept of compositions.

Figure 2.5: TODO

```
namespace com.mycompany.test {
  composition testSystem {
    instance dc: DelayCalculator
    instance screen1: InfoScreen
    instance screen2: InfoScreen
    connect dc.screens to (screen1.default, screen2.default)
  }
}
```

Of course in a real system, the DelayCalculator would have to dynamically discover all the available InfoScreens at runtime. There is not much point in manually describing those connections. So, we specify a query that is executed at runtime against some kind of naming/trader/lookup/registry infrastructure. It is re-executed every 60 seconds to find the InfoScreens that had just come online.

A similar approach can be used to realize load balancing or fault

Figure 2.6: TODO

```
namespace com.mycompany.production {
  instance dc: DelayCalculator
  dynamic connect dc.screens every 60 query {
    type = IInfoScreen
    status = active
  }
}
```

tolerance. A static connector can point to a primary as well as a backup instance. Or a dynamic query can be re-executed when the currently used component becomes unavailable. To support registration of instances, we add additional syntax to their definition. A registered instance automatically registers itself with the registry, using its name (qualified through the namespace) and all provided interfaces. Additional parameters can be specified, the following example registers a primary and a backup instance for the DelayCalculator.

Figure 2.7: TODO

```
namespace com.mycompany.datacenter {
  registered instance dc1: DelayCalculator {
    registration parameters {role = primary}
  }
  registered instance dc2: DelayCalculator {
    registration parameters {role = backup}
  }
}
```

Until now we didn't really define what an interface is. We knew that we'd like to build the system based on a messaging infrastructure. Here's our first idea: an interface is a collection of messages, where each message has a name and a list of typed parameters - this also requires the ability to define data structures, but in the interest of brevity, we won't show that. After discussing this notion of interfaces for a while, we noticed that it was too simplistic. We needed to be able to define the direction of a message: does it flow in or out of the port? More generally, which kinds of message interaction patterns are there? We identified several, here are examples of oneway and request-reply:

We talked a long time about various message interaction patterns. After a while it turned out that one of the core use cases for messages is to push status updates of various assets out to various interested parties. For example, if a flight is delayed because of a technical prob-

Figure 2.8: TODO

```
interface IAircraftStatus {
  oneway message reportPosition(aircraft: ID, pos: Position )
  request-reply message reportProblem {
    request (aircraft: ID, problem: Problem, comment: String)
    reply (repairProcedure: ID)
  }
}
```

lem with an aircraft, then this information has to be pushed out to all the InfoScreens in the system. We prototyped several of the messages necessary for "broadcasting" complete updates, incremental updates and invalidations of a status item. And then it hit us: we were working with the wrong abstraction! While messaging is a suitable transport abstraction for these things, architecturally we're really talking about replicated data structures. It basically works the same way for all of those structures:

- You define a data structure (such as FlightInfo).
- The system then keeps track of a collection of such data structures.
- This collection is updated by a few components and typically read by many other components.
- The update strategies from publisher to receiver always include full update of all items in the collection, incremental updates of just one or a few items, invalidations, etc.

Once we understood that, in addition to messaging, there's this additional core abstraction in the system, we added this to our Architecture DSL and were able to write something like the following. We define data structures and replicated items. Components can then publish or consume those replicated data structures. We state that the publisher publishes the replicated data whenever something changes in the local data structure. However, the InfoScreen only needs an update every 60 seconds (as well as a full load of data when it is started up).

This is much more concise compared to a description based on messages. We can automatically derive the kinds of messages needed for full update, incremental update and invalidation and create these messages in the model using a model transformation. The description also much more clearly reflects the actual architectural intent: it expresses better what we want to do (replicate state) compared to a lower level description of how we want to do it (sending around state

Figure 2.9: TODO

```
struct FlightInfo {
   //  … attributes …
}

replicated singleton flights {
  flights: FlightInfo[]
}

component DelayCalculator {
  publishes flights { publication = onchange }
}

component InfoScreen {
  consumes flights { init = all update = every(60) }
}
```

update messages). While replication is a core concept for data, there's of course still a need for messages, not just as an implementation detail, but also as a way to express your architectural intent. It is useful to add more semantics to an interface, for example, defining valid sequencing of messages. A well-known way to do that is to use protocol state machines. Here is an example that expresses that you can only report positions and problems once the aircraft is registered. In other words, the first thing an aircraft has to do is register itself.

Initially, the protocol state machine is in the new state. Here, the only valid message is `registerAircraft`. If this is received, we transition into the registered state. In registered, you can either `unregisterAircraft` and go back to new, or receive a `reportProblem` or `reportPosition` message in which case you'll remain in the registered state. We mentioned above that the system is distributed geographically. This means it is not feasible to update all part of the systems (e.g. all InfoScreens or all AircraftModules) in one swoop. As a consequence, there might be several versions of the same component running in the system. To make this feasible, many non-trivial things need to be put in place in the runtime. But the basic requirement is this: you have to be able to mark up versions of components, and you have to be able to check them for compatibility with old versions. The following piece of code expresses that the DelayCalculatorV2 is a new implementation of DelayCalculator. `newImplOf` means that no externally visible aspects change which is why no ports and other externally-exposed details of the component are declared. For all intents and purposes, it's the same thing - just maybe a couple of bugs are fixed.

If you really want to evolve a component (i.e., change its external

Figure 2.10: TODO

```
interface IAircraftStatus {
  oneway message registerAircraft(aircraft: ID )
  oneway message unregisterAircraft(aircraft: ID )
  oneway message reportPosition(aircraft: ID, pos: Position )
  request-reply message reportProblem {
    request (aircraft: ID, problem: Problem, comment: String)
    reply (repairProcedure: ID)
  }
  protocol initial = new {
    state new {
      registerAircraft => registered
    }
    state registered {
      unregisterAircraft => new
      reportPosition
      reportProblem
    }
  }
}
```

Figure 2.11: TODO

```
component DelayCalculator {
  publishes flights { publication = onchange }
}
newImplOf component DelayCalculator: DelayCalculatorV2
```

signature) you can write it like this:

Figure 2.12: TODO

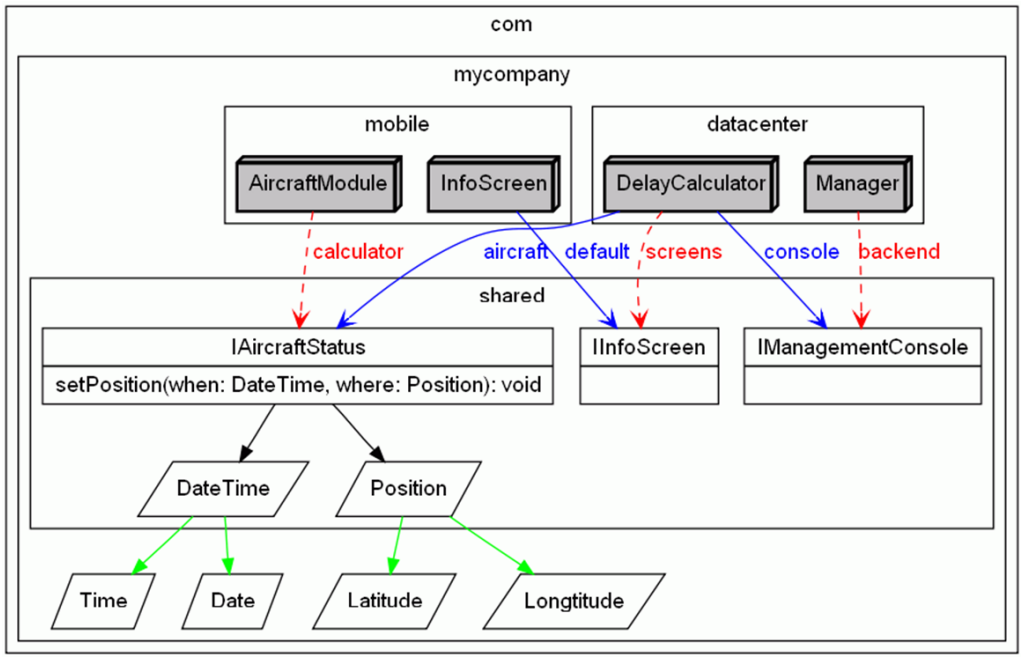
```
component DelayCalculator {
  publishes flights { publication = onchange }
}
newVersionOf component DelayCalculator: DelayCalculatorV3 {
  publishes flights { publication = onchange }
  provides somethingElse: ISomething
}
```

The keyword is `newVersionOf` and now you can provide additional features (like the `somethingElse` port) and you remove required ports. You cannot add additional required ports or remove any of the provided ports since that would destroy the "plug-in compatibility". Constraints make sure that these rules are enforced on model level. Using

the approach shown here, we were able to quickly get a grip towards the overall architecture of the system. All of the above was actually done in the first day of the workshop. We defined the grammar, some important constraints, and a basic editor (without many bells and whistles). We were able to separate what we wanted the system to do from how it would achieve it: all the technology discussions were now merely an implementation detail of the conceptual descriptions given here (albeit of course, a very important implementation detail). We also had a clear and unambiguous definition of what the different terms meant. The nebulous concept of component has a formal, well-defined meaning in the context of this system. It didn't stop there: the next day involved a discussion of how to actually code the implementation for a component and which parts of the system could be automatically generated.

In this project, as well as in many other ones, we have used textual DSLs. We have argued in this book why textual DSLs are superior in many cases, and these arguments apply here as well. However, we did use visualization to show the relationships between the building blocks, and to communicate the architecture to stakeholders who were not willing to dive into the textual models. Figure 2.13 shows an example, created with Graphviz.

Figure 2.13: TODO



2.2.3 *Architecture DSL Concepts*

What we did in a nutshell The approach recommends the definition of a formal language for your project's or system's conceptual architecture. You develop the language as the understanding of your architecture grows. The language therefore always resembles the complete understanding about your architecture in a clear and unambiguous way. As we enhance the language, we also describe the application architecture using that language.

Component Implementation By default, component implementation code is written manually against the generated API code, using well-known composition techniques such as inheritance, delegation or partial classes.

However, there are other alternatives for component implementation that do not use a 3GL/GPL, but instead use formalisms that are specific to certain classes of behavior: state machines, business rules or workflows. You can also define and use a domain-specific language for certain classes of functionality in a specific business domain. Note how this connects the dots to business domain DSLs mentioned below.

Be aware that the discussion in this section is only really relevant for application-specific behavior, not for all implementation code. Huge amounts of implementation code are related to the technical infrastructure (e.g., remoting, persistence, workflow) of an application. It can be derived from the architectural models, and generated automatically.

Standards, ADLs and UML Describing architecture with formal languages is not a new idea. Various communities recommend using Architecture Description Languages (ADLs, for example [ADL1, ADL2, ADL3, ADL4]) or the Unified Modeling Language (UML) to this end. However, all of those approaches advocate using existing, generic languages for specifying architecture, although some of them, including the UML, can be customized to some degree.

Unfortunately, efforts like that completely miss the point. We have not experienced much benefit in shoehorning your architecture description into the (typically very limited as well as too generic) constructs provided by predefined languages - one of the core activities of the approach explained is this paper is the process of actually building your own language to capture your system's conceptual architecture.

So: are standards important? And if so, where?

In order to use any architecture modeling language successfully, people first and foremost have to understand the architectural concepts they are dealing with. Even if UML is used, people will still have to understand the architectural concepts and map them to the language - in the case of UML that often requires an architecture-specific profile.

Is a profiled UML still standard?

Note that we do not propose to ignore standards altogether. The tools are built on MOF/EMOF, which is an OMG standard, just like the UML. It is just on a different meta level.

Of course, you can use the approach explained above with UML profiles. We have done this in several projects and my conclusion is that it doesn't work well in most environments. There are several reasons, three of them are:

- Instead of thinking about architectural concepts, working with UML makes you think more about how to use UML's existing constructs to express your architectural intentions. That is the wrong focus! It also takes a lot of effort just to learn UML well enough to make choices that are sensible (enough). In the end, you'll find that UML has very little actual semantics of its own.
- UML is a huge standard and allows quite a bit of variation in how it's used and implemented, so in order to be able to have *processable* models, you'll have to confine your users and yourself to a certain subset and implementation, which takes additional effort.
- UML tools typically don't integrate very well with existing development infrastructure (editors, CVS/SVN, diff/merge). That is a huge problem now models play the role of source code (as opposed to being mere "analysis pictures").

Code Generation It should have become clear from the paper that the primary benefit of developing the architecture DSL (and using it) is just that: understanding concepts by removing any ambiguity and defining them formally. It helps you understand your system and get rid of unwanted technology interference.

But of course, now that we have a formal model of the conceptual architecture (the language) and also a formal description of system(s) we're building - i.e., the sentences (or models) defined using the language - we might as well use it to do more good:

- We generate an API against which the implementation is coded. That API can be non-trivial, taking into account the various messaging paradigms, replicated state, etc. The generated API allows developers to code the implementation in a way that does not depend on any technological decisions (beyond the precise technical description format of the API): the generated API hides those from the component implementation code. We call this generated API and the set of idioms to use it the programming model.
- Remember that we expect some kind of component container or middleware platform to run the components, so we also generate

> the code that is necessary to run the components (including their technology-neutral implementation) on the implementation technology of choice. We call this layer of code the technology mapping code (or glue code). It typically also contains a whole bunch of configuration files for the various platforms involved. Sometimes this requires addition "mix-in models" (some call these "marking models") that specify configuration details for the platform. As a side effect, the generators capture best practices in working with the technologies you've decided to use.

It is of course completely feasible to generate APIs for several target languages (supporting component implementation in various languages) and/or generating glue code for several target platforms (supporting the execution of the same component on different middleware platforms). This nicely supports potential multi-platform requirements, and also provide a way to scale or evolve the infrastructure over time.

Another point worth making is that you typically generate in several phases: a first phase uses type definitions (components, data structures, interfaces) to generate the API code so you can code the implementation. A second phase generates the glue code and the system configuration code. As a consequence, it is often sensible to separate type definitions from system definitions in models: they are used at different times in the overall process, and also often created, modified and processed by different people. This shows that language (and model) modularization is useful.

In summary, the generated code supports an efficient and technology independent implementation and hides much of the underlying technological complexity, making development more efficient and less error-prone.

The Role of Patterns Patterns are an important part of today's software engineering practice. They are a proven way of capturing working solutions to recurring problems, including their applicability, tradeoffs and consequences. So how do patterns factor into the approach described above?

Architecture Patterns and Pattern Languages describe blueprints for architectures that have been used successfully. They can serve as an inspiration for building you own system's architecture. Once you have decided on using a pattern (and have adapted it to your specific context) you can make concepts defined in the pattern first class citizens of your DSL. In other words, patterns influence the architecture, and hence the grammar of the DSL.

Design Patterns, as their name implies, are more concrete, more

implementation-specific than architectural patterns. It is unlikely that they will end up being central concepts in your architecture DSL. However, when generating code from the models, your code generator will typically generate code that resembles the solution structure of a number of patterns. Note, however, that the generator cannot decide on whether a pattern should be used: this is a tradeoff the (generator) developer has to make manually.

When talking about DSLs, generation and patterns, it is important to mention that you cannot completely automate patterns: a pattern doesn't just consist of the solution's UML diagram nor is it a mere code template. Significant parts of a pattern explain which forces affect the pattern's solution, when a pattern can be applied and when it cannot, as well as the consequences of using the pattern. A pattern often also documents many variations of itself that may all have different advantages and disadvantages. A pattern that has been implemented in the context of a transformation does not account for these aspects - the developer of the transformations must take them into account, assess them and make decisions accordingly.

What needs to be documented? I advertise the above approach as a way to formally describe your system's conceptual and application architecture. So, this means it serves as some kind of documentation, right?

Right, but it does not mean that you don't have to document anything else. Here's a bunch of things you still need to document:

Rationales/Architectural Decisions The DSLs describe what your architecture(s) look like, but it does not explain why. You still need to document the rationales for your architectural and technological decisions. Typically you should refer back to your (non-functional) requirements here. Note that the grammar is a really good baseline. Each of the constructs in your architecture DSL grammar is the result of a number of architectural decisions. So, if you explain for each grammar element why it is there (and why maybe certain other alternatives have not been chosen) you are well on your way to document the important architectural decisions. A similar approach can be used for the application architecture, i.e. the instances of the DSL.

User Guides A language grammar can serve as a well-defined and formal way of capturing an architecture, but it is not a good teaching tool. So you need to create tutorials for your users (i.e., the application programmers) on how to use the architecture. This includes what and how to model (using your DSL) and also how to generate code and how to use the programming model (how to fill in the implementation code into the generated skeletons).

There are more aspects of an architecture that might be worth documenting, but the above two are the most important.

2.3 *Embedding Business Logic via DSLs*

Architecture DSLs as introduced above focus on the description of software architecture. As the section on component implementation explains, we do not concern ourselves much with the question of how the components are implemented, i.e. how business logic is expressed.

Of course this is essential for building the actual software system. As we outline above, one can either implement the application logic with manually written code, or one can use a DSL for that. This leads us to application domain DSLs, to which we have given a separate chapter.

If application domain DSLs are used together with architecture DSLs, then the code generator for the application domain DSL has to act as the "implementor" of the components (or whatever other architectural abstractions are defined in the architecture DSL), and the generated code must fit in. This is an example of DSL cascading, described here TODO.

2.4 *Component Models*

There are many (more or less formal) definitions of what a components is. They range from a building block of software systems to something with explicitly defined context dependencies to something that contains business logic and is run inside a container.

Our understanding (notice we are not saying we have a real definition) is that a component is the smallest architecture building block. When defining a system's architecture, you don't look inside components. Components have to specify all their architecturally relevant properties declaratively (aka in meta data, or models). As a consequence, components become analyzable and composable by tools. Typically they run inside a container that serves as a framework to act on the runtime-relevant parts of the meta data. The component boundary is the level at which the container can provide technical services such as as logging, monitoring, or failover.

We don't have any specific requirements towards what meta data a component actually contains (and hence, which properties are described). We think that the concrete notion of components has to be defined for each (system/platform/product line) architecture separately:

this is exactly what we do with the language approach introduced above.

From our experience in development projects, we find that we almost always start by modeling the component structure of the system to be built. To do that, we start by defining what a component actually is - that is, by defining a meta model for component-based development. Independent of the domain in which the development project resides, these meta models are quite similar across application domains - insurance, e-commerce, embedded, radio astronomy and so on - as opposed to technical domains such as persistence, transaction processing, security. We therefore show parts of these meta models here to give you a head start when defining your own component architecture.

Three Typical Viewpoints It is useful to look at a component-based system from several viewpoints. Three viewpoints form the backbone of the description. These viewpoints should end up being separate models in their own separate DSL, so in the following text we use "model" and "interchangeably".

The **Type** viewpoint describes *component types*, *interfaces*, and *data structures*. A component provides a number of interfaces and references a number of required interfaces. An interface owns a number of operations, each with a return type, parameters, and exceptions. Figure 2.14 shows this.

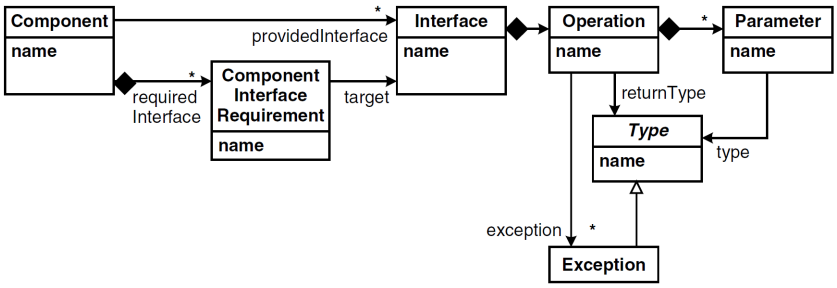


Figure 2.14: TODO

To describe the data structures with which the components work (figure 2.15), we start out with the abstract type Type. We use primitive types as well as complex types. A complex type has a number of named and typed attributes. There are two kinds of complex types. Data transfer objects are simple (C-style) structs that are used to exchange data among components. Entities have a unique ID and can be made persistent (this is not visible from the meta model). Entities can

reference each other and thus build more complex data graphs. Each reference has to specify whether it is navigable in only one or in both dimensions. A reference also specifies the cardinalities of the entities at the respective ends.

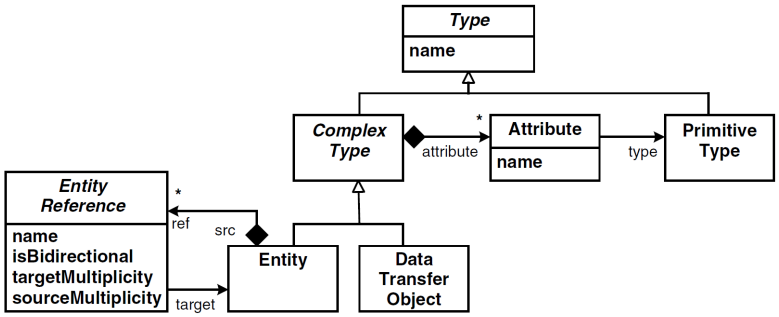


Figure 2.15: TODO

The **Composition** viewpoint, illustrated in figure 2.16, describes component instances and how they are connected. A configuration consists of a number of component instances, each referencing their type (from the Type viewpoint). An instance has a number of wires: a wire is an instance of a component interface requirement. Note the constraints defined in the meta model:

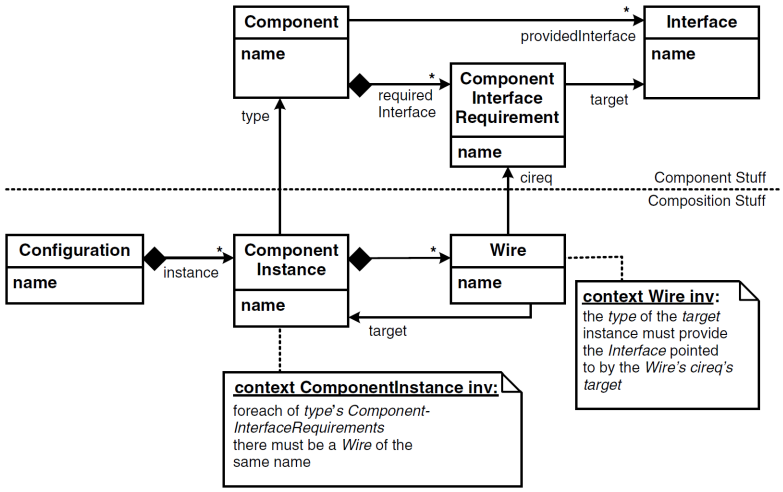


Figure 2.16: TODO

- For each component interface requirement defined in the instance's type, we need to supply a wire.
- The type of the component instance at the target end of a wire needs to provide the interface to which the wire's component interface requirement points.

Using the Type and Composition viewpoints, it is possible to define component types as well as their collaborations. Logical models of applications can be defined. From the Composition viewpoint, you can for example generate or configure a container that instantiates the component instances.

The **System** viewpoint describes the system infrastructure onto which the logical system defined with the two previous viewpoints is deployed (figure 2.17).

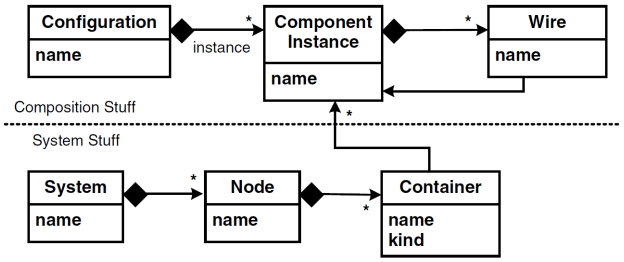


Figure 2.17: TODO

A system consists of a number of nodes, each one hosting containers. A container hosts a number of component instances. Note that a container also defines its kind - this could be things like OSGi, JEE, Eclipse or Spring. Based on this information, you can generate the necessary "glue" code to run the components in that kind of container. The node information, together with the connections defined in the composition model, allows you to generate all kinds of things, from remote communication infrastructure code and configuration to build and packaging scripts.

You may have observed that the dependencies among the viewpoints are well-structured. Since you want to be able to define several compositions using the same components and interfaces, and since you want to be able to run the same compositions on several infrastructures, dependencies are only legal in the directions shown in figure 2.18.

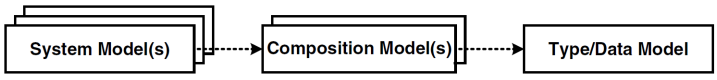


Figure 2.18: TODO

Aspect Models The three viewpoints described above are a good starting point for modeling and building component-based systems. However, in most cases these three models are not enough. Additional aspects of the system have to be described using specific aspect models that are arranged around the three core viewpoint models. The

following aspects are typically handled in separate aspect models:

- Persistence
- Authorization and Authentication (important in enterprise systems)
- Forms, layout, page flow (for Web applications)
- Timing, scheduling and other quality of service aspects (especially in embedded systems)
- Packaging and deployment
- Diagnostics and monitoring

The idea of aspect models is that the information is not added to the three core viewpoints, but rather is described using a separate model with a suitable concrete syntax. Again, the meta model dependencies are important: the aspects may depend on the core viewpoint models and maybe even on one another, but the core viewpoints must not depend on any of the aspect models. Figure 2.19 illustrates a simplified persistence aspect meta model.

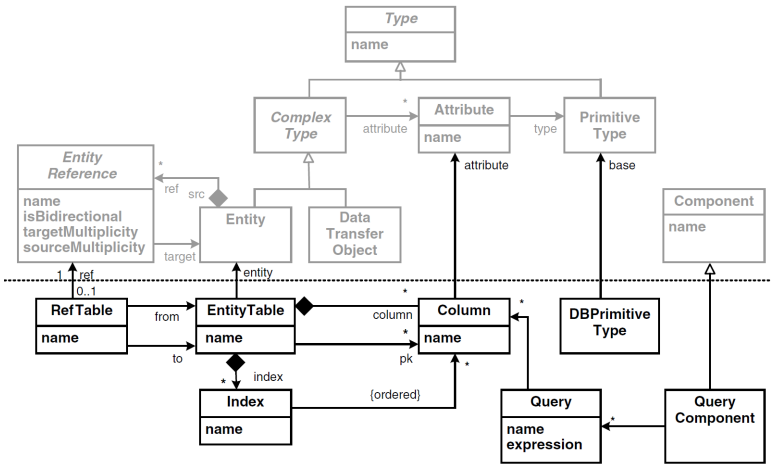


Figure 2.19: TODO

Variations The meta models we describe above cannot be used in exactly this way in every project. Also, in many cases the notion of what constitutes a component needs to be extended. So there are many variations of these meta models. However, judging from practice, even these variations are limited. In this section we want to illustrate some of the variations we've come across.

You might not need to use interfaces to declare *all* of a component's behavior: Operations could be added directly to the components. As

a consequence, of course, you cannot reuse the interface's "contracts" separately, independently of the supplier or consumer components.

Often you'll need different kinds of components, such as domain components, data access (DAO) components, process components, or business rules components. Depending on this component classification you can come up with valid dependency structures between components. You will typically also use different ways of implementing component functionality, depending on the component types.

Another way of managing dependencies is to mark each component with a layer tag, such as domain, service, GUI, or facade, and define constraints on how components in these layers may depend on each other.

Hierarchical components, as illustrated in figure 2.20, are a very powerful tool. Here a component is structured internally as a composition of other component instances. Ports define how components may be connected: a port has an optional protocol definition that allows for port compatibility checks that go beyond simple interface equality. While this approach is powerful, it is also non-trivial, since it blurs the formerly clear distinction between Type and Composition viewpoints.

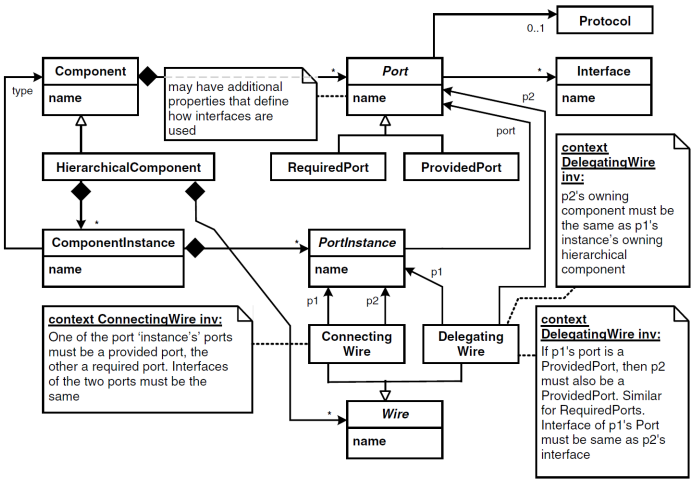


Figure 2.20: TODO

A component might have a number of configuration parameters - comparable to command line arguments in console programs - that help configure the behavior of components. The parameters and their types are defined in the type model, and values for the parameters can be specified later, for example in the models for the Composition or the System viewpoints.

You might want to say something about whether the components are stateless or stateful, whether they are thread-safe or not, and what their lifecycle should look like (for example, whether they are passive

or active, whether they want to be notified of lifecycle events such as activation, and so on).

It is not always enough to use simple synchronous communication. Instead, one of the various asynchronous communication patterns, such as those described in [VKZ04], might be applicable. Because using these patterns affects the APIs of the components, the pattern to be used has to be marked up in the model for the Type viewpoint, as shown in Figure 7.13.

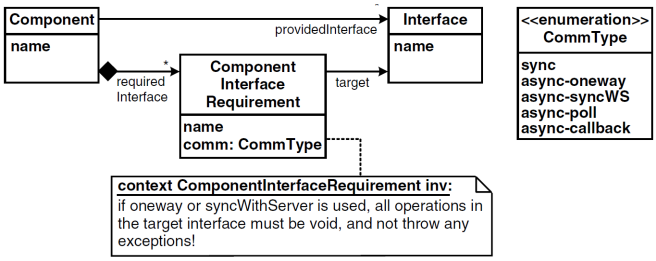


Figure 2.21: TODO

In addition to the communication through interfaces, you might need (asynchronous) events using a static or dynamic publisher/subscriber infrastructure. It is often useful that the "direction of flow" of these events is the opposite of the dependencies discussed above.

The (model for the) Composition viewpoint connects component instances statically. This is not always feasible. If dynamic wiring is necessary, the best way is to embed the information that determines which instance to connect to at runtime into the static wiring model. So, instead of specifying in the model that instance A must be wired to instance B, the model only specifies that A needs to connect to a component with the following properties: needs to provide a certain interface, and for example offer a certain reliability. At runtime, the wire is "deferenced" to a suitable instance using an instance repository. This approach is similar to CORBA's trader service.

Finally, it is often necessary to provide additional means of structuring complex systems. The terms business component or sub system are often used. Such a higher-level structure consists of a number of components. Optionally, constraints define which kinds of components may be contained in a specific kind of higher-level structure. For example, you might want to define that a business component always consists of exactly one facade component and any number of domain components. The approach laid out above works for engineering a suitable DSL for the higher-level situation as well, as it's basically a matter of finding/identifying the "right" abstractions. You might have to come up with a way to parametrize the lower-level structure to a small extent and to reference such parametrized structures from the

higher-level DSL to make this really useful.

2.5 *Summing Up*

3
DSLs and Product Lines

This paper investigates the application of domain-specific languages in product line engineering (PLE). We start by analyzing the limits of expressivity of feature models. Feature models correspond to context-free grammars without recursion, which prevents the expression of multiple instances and references. We then show how domain-specific languages (DSLs) can serve as a middle ground between feature modeling and programming. They can be used in cases where feature models are too limited, while keeping the separation between problem space and solution space provided by feature models. We then categorize useful combinations between configuration with feature model and construction with DSLs and provide an integration of DSLs into the conceptual framework of PLE. Finally we show how use of a consistent, unified formalism for models, code, and configuration can yield important benefits for managing variability and traceability. We illustrate the concepts with several examples from industrial case studies.

3.1 Introduction

The goal of product line engineering (PLE) is to efficiently manage a range of products by factoring out commonalities such that definitions of products can be reduced to a specification of their variable aspects. One way of achieving this is the expression of product configurations on a higher level of abstraction than the actual implementation. An automated mapping transforms the configuration to the implementation. Traditionally this higher level of abstraction is realized with feature models [1] or similar configuration formalisms such as orthogonal variability models [2] or decision models [3]. A feature model defines the

[1] D. Beuche, H. Papajewski, and W. SchrÃűder-Preikschat. Variability management with feature models. *SCP*, 53(3):333–352, 2004

[2]

[3]

set of valid configurations for a product in a product line by capturing all variations points (i.e. features), as well as the constraints between them.

However, since feature models can only describe bounded configuration spaces, their expressivity is limited. This limitation can be reduced to the fact that the feature modeling formalism corresponds to context-free grammars without recursion. By using full context-free grammars, the expressivity of feature modeling can be extended to domain-specific configuration languages with unbounded configuration spaces, without the need to regress to programming in a general-purpose programming language.

In this paper, we investigate the application of domain-specific languages (DSLs) in product line engineering. We first analyze the difference in expressivity between feature models and DSLs, and discuss when to use which approach (Section 3.2). For a given product line, the decision between feature models and DSLs often is not mutually exclusive. We categorize approaches for composing feature models and DSLs, for example, by using feature models to configure DSL programs (Section 3.3). If we express the complete product definition based on the linguistic integration of feature models and DSLs, we can reap a number of additional benefits such as uniform traceability (Section 3.4). To illustrate the approach, we discuss three industrial case studies of DSLs used in product line engineering (Section 3.5). Finally, to put the approach into perspective, we present a mapping of DSLs and their tools to the core concepts and processes of PLE (Section 3.6).

3.2 *From Feature Models to DSLs*

A feature model is a compact representation of the features of the products in a product line, as well as the constraints imposed on configurations. Feature models are an efficient formalism for *configuration*, i.e. for *selecting* a valid combination of features from the feature model. The set of products that can be defined by feature selection is fixed and finite: each valid combination of selected features constitutes a product. This means that all valid products have to be "designed into" the feature model, encoded in the features and the constraints among them. Some typical examples of variation points that can be modeled with feature models are the following:

- Does the communication system support encryption?
- Should the in-car entertainment system support MP3s?
- Should the system be optimized for performance or memory foot-

print?

- Should messages be queued? What is the queue size?

In the simplest case, product lines can be implemented with programming languages only. For example, an object-oriented framework is implemented as part of domain engineering, and it is then customized specifically for each product by writing framework client code. The advantage of this code-only approach is that no special tools are needed and that there is a high degree of flexibility for the implementers. Any kind of variability can be expressed with programming languages, using techniques such as preprocessors, branching, polymorphism, generics or design patterns. However, using these techniques only, there is no distinction between the problem space and the solution space — no higher level representation of the variability in a product line is provided, making it hard to keep track of, and manage the variability. This is especially problematic if the definition of a product requires consistent changes in several places in the implementation artifacts.

Feature models provide an abstraction over the implementation of the product. Based on the feature configuration, mappings derive the necessary adaptations of the underlying implementation artifacts. Because of this abstraction, feature model-based configuration is simple to use — product definition is basically a decision tree. This makes product configuration efficient, and potentially accessible for stakeholders other than software developers.

As described by Batory [4] and Czarnecki [5], a particular advantage of feature models is that a mapping to logic exists. Using SAT solvers, it is possible to check, for example, whether a feature model has valid configurations at all. The technique can also be used to automatically complete partial configurations. This has been shown to work for realistically-sized feature models [6]. Pure::variants (http://pure-systems.com) maps feature models to Prolog to achieve a similar goal, as does the GEMS-based tool described in [7].

[4]
[5]
[6]
[7]

In the rest of this section we will discuss the limitations of feature models. We argue that in cases in which feature models are unsuitable, we should not regress to low-level programming, but use DSLs instead, to avoid losing the differentiation between problem space and solution space. As an example we use a product line of water fountains as found in recreational parks[8]. Fountains can have several basins, pumps and nozzles. Software is used to program the behavior of the pumps and valves to make the sprinkling waters aesthetically pleasing. The feature model in Fig. 3.1 represents valid hardware combinations for a simple water fountain product line. The features correspond to the presence of a hardware component in a particular fountain instal-

[8] This is an anonymized version of an actual project the authors have been working on. The real domain was different, but the example languages presented in this paper have been developed and used for that other domain.

lation.

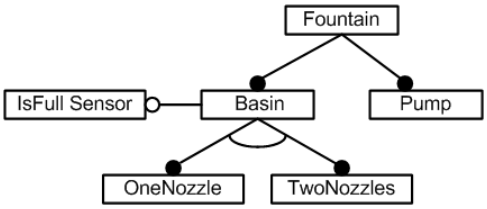


Figure 3.1: Feature model for the simple fountains product line used as the example. Fountains have basins, with one or two nozzles, and an optional full sensor. In addition, fountains have a pump.

The real selling point of water fountains is their *behavior*. A fountain's behaviour determines how much water each pump should pump, at which time, with what power, or how a pump reacts when a certain condition is met, e.g. a basin is full. Expressing the full range of such behaviors is not possible with feature models. Feature models can be used to select among a fixed number of predefined behaviors, but approximating all possible behaviors would lead to unwieldy feature models. Instead we could program the behavior in a general purpose language such as C. However, this means that we lose the abstraction feature models provide, reducing efficiency and understandability, as well as the possibility for direct involvement of non-programmers. Domain-specific languages can serve as a middle ground between the controlled setting of feature model-based configuration and the complete lack of restrictions of general purpose programming languages.

3.2.1 *Feature Models as Grammars*

To understand the limitations of feature models, consider their relation to grammars. Feature models essentially correspond to context-free grammars without recursion [9]. For example, the feature model in Fig. 3.1 is equivalent to the following grammar. We use all caps to represent terminals, and camel-case identifiers as non-terminals:

[9]

```
Fountain -> Basin PUMP
Basin -> ISFULLSENSOR? (ONENOZZLE | TWONOZZLES)
```

This grammar generates a finite number of sentences, i.e there are exactly four possible configurations, which correspond to the finite number of products in the product line. However, this formalism does not make sense for modeling behavior, for which there is typically an infinite range of variability. To accommodate for unbounded variability, the formalism needs to be extended. Allowing recursive grammar productions is sufficient to model unbounded configuration spaces, but for convenience, we consider also attributes and references.

Attributes express properties of features. For example, the *PUMP* could have an integer attribute *rpm*, representing the power setting of

the pump. Some feature modeling tools (e.g. pure::variants) support attributes.

```
Fountain -> Basin PUMP(rpm:int)
Basin -> ISFULLSENSOR? (ONENOZZLE | TWONOZZLES)
```

Recursive grammars can be used to model repetition and nesting. Repetition is also supported by cardinality-based feature models, as described in [10]. Nesting is necessary to model tree structures such as those occurring in expressions. The following grammar extends the fountain feature model with a *Behavior*, which consists of a number of *Rules*. The *Basin* can now have any number of *Nozzles*.

[10]

```
Fountain -> Basin PUMP(rpm:int) Behavior
Basin -> ISFULLSENSOR? NOZZLE*
Behavior -> Rule*
Rule -> CONDITION CONSEQUENCE
```

References allow the creation of context-sensitive relations between parts of generated sentences — or subtrees of the generated derivation trees. For example, by further extending our fountain grammar we can describe a rule whose condition refers to the *full* attribute of the *ISFULLSENSOR* and whose consequence sets a *PUMP*'s *rpm* to 0.

```
Fountain -> Basin id:PUMP(rpm:int)? Behavior
Basin -> id:ISFULLSENSOR(full:boolean)? id:NOZZLE*
Behavior -> Rule*

Rule -> Condition Consequence
Condition -> Expression
Expression -> ATTRREFEXPRESSION | AndExpression |
              GreaterThanExpression | INTLITERAL;

AndExpression -> Expression Expression
GreaterThanExpression -> Expression Expression

Consequence -> ATTRREFEXPRESSION Expression
```

Fig. 3.2 shows a possible rendering of the grammar with an enhanced feature modeling notation. We use cardinalities, as well as references to existing features, the latter are shown as dotted boxes. A valid configuration could be the one shown in Fig. 3.3. It shows a fountain with one basin, two nozzles named *n1* and *n2*, one sensor *s* and a pump *p*. It contains a rule that expresses that if the *full* attribute of *s* is set, and the *rpm* of pump *p* is greater than zero, then the *rpm* should be set to zero.

3.2.2 *Domain-Specific Languages*

While the extended grammar formalism discussed above enables us to cover the full range of behavior variability, the use of trees to instantiate these grammars is not practical. Another interpretation of these

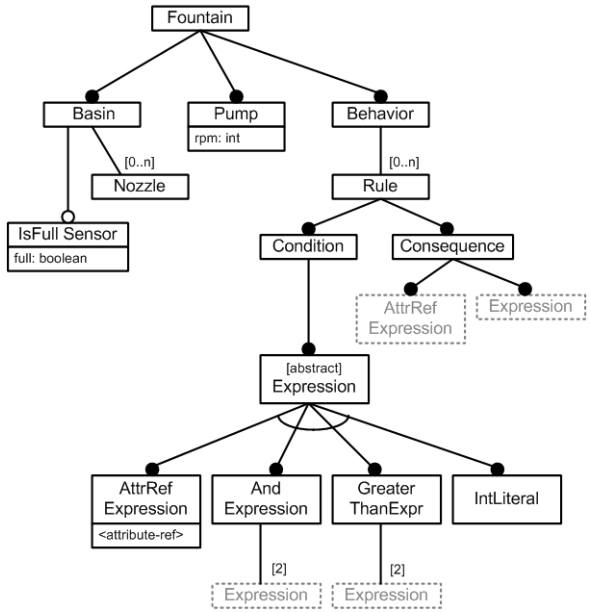


Figure 3.2: An extended feature modeling formalism is used to represent the example feature model with attributes, recursion and and references (the dotted boxes).

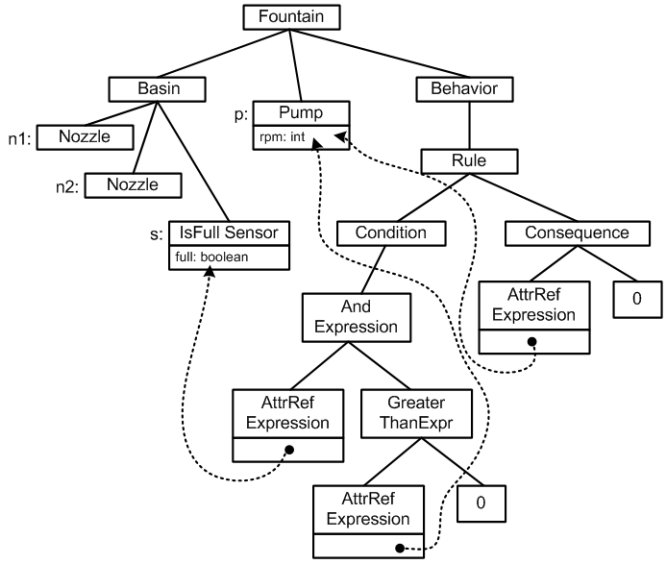


Figure 3.3: Example configuration using a tree notation. Referenceable identities are rendered as labels left of the box. The dotted lines represent references to variables.

grammars is as definition of a *language* with a *textual concrete syntax* — the tree in Fig. 3.3 looks like an abstract tree (AST). To make the language readable we need to add concrete syntax definitions (keywords), as in the following extension of the fountain grammar:

```
Fountain -> "fountain" Basin Pump Behavior
Basin -> "basin" IsFullSensor Nozzle*
Behavior -> Rule*

Rule -> "if" Condition "then" Consequence
Condition -> Expression
Expression -> AttrRefExpression | AndExpression |
```

```
            GreaterThanExpression | IntLiteral;

AndExpression -> Expression "&&" Expression
GreaterThanExpression -> Expression ">" Expression
AttrRefExpression -> <attribute-ref-by-name>
IntLiteral -> (0..9)*

Consequence -> AttrRefExpression "=" Expression

IsFullSensor: "sensor" ID (full:boolean)?
Nozzle: "nozzle" ID
Pump: "pump" ID (rpm:int)?
```

We can now write a program that uses a convenient textual notation, which is especially useful for the expressions in the rules. We have created a *domain-specific language* for configuring the composition *and* behavior of fountains. A complete language definition would also include typing rules and other constraints, but that is beyond the scope of this paper.

```
fountain
  basin sensor s
        nozzle n1
        nozzle n2
  pump p
  if s.full && p.rpm > 0 then p.rpm = 0
```

DSLs fill the gap between feature models and programming languages. They can be more expressive than feature models, but they are not as unrestricted and low-level as programming languages. Like programming languages, DSLs, support *construction*, allowing the composition of an unlimited number of programs. Construction happens by instantiating language concepts, establishing relationships, and defining values for attributes. We do not a-priori know all possible valid programs. In contrast to programming languages, DSLs keep the distinction between problem space and solution space intact since they consist of concepts and notations relevant to the problem domain. Non-programmers can continue to contribute directly to the product development process, without being exposed to implementation details.

DSLs are a good fit when instances of concepts need to be created, when relationships between these instances must be established, or when algorithmic behavior has to be described, e.g. in business rules, calculations, or events. Examples of the application of DSLs include calculating the VAT and other taxes in an invoicing product line, specifying families of pension contracts, and defining communication protocols in embedded systems.

In general, a *domain-specific language* (DSL) is a software language specialized for a particular problem domain. DSLs can use graphical, textual or tabular concrete syntax, or any combination thereof. Like

programming languages, DSLs can either be compiled — typically through code generation to a programming language — or interpreted by an interpreter running on the target environment.

A DSL's concrete and abstract syntax are tailored closely to the domain at hand. Using DSLs only requires knowledge about the problem domain, not about the solution domain. This improves productivity, quality and maintainability. Productivity is improved because a higher level notation is provided, avoiding dealing with implementation details. Quality is improved because transformations or interpreters execute the programs consistently. Maintainability is improved because changes to the program can be done on the level of the DSL program, or by changing the generator or interpreter.

A note on terminology: we use the terms *DSL program*, *DSL code* and *model* interchangably. Strictly speaking, the DSL code is a textual representation of a model, but this distinction is not relevant in this paper. While we focus primarily on textual DSLs in this paper, the discussion is equally valid for DSLs using other notations, as long as the underlying expressivity is not reduced, as for example in purely tabular notations. However, expression-like models can best be expressed using a textual notation.

Supplying convenient editors and other tools along with the DSL increases the usability of the language significantly. The term *language workbench* has been introduced by Martin Fowler in 2004 [11], referring to tools that support the efficient definition, composition and use of DSLs. Open Source examples include Eclipse Xtext (http://eclipse.org/xtext), Spoofax (http://strategoxt.org/Spoofax) and JetBrains MPS (http://www.jetbrains.com/mps/).

[11] M. Fowler. Language workbenches: The killer-app for domain specific languages?, 2005

3.3 *Combining DSLs and Feature Models*

In the previous sections we have explained the difference in expressive power between feature models and DSLs. We have also outlined the benefits and drawbacks of both approaches. In this section we categorize how both approaches can be combined.

3.3.1 *Implementing Components with DSLs*

One way of using feature models is to use them to select between a number of prebuilt, reusable components that are used to customize a framework as part of product definition. Instead of implementing these components in a programming language, they can be implemented using DSLs. This is especially useful if the behavior in these components is highly domain-specific. At the time of product definition using the feature model, the domain expert does not have to know

that the components have been developed using DSLs .

For example, we have worked on a system in which a DSL was used to describe OSGi component structures, generating all the low level OSGi details. Feature model-based configuration was then used to create different products from these components.

3.3.2 *Variation over models*

Feature models can also be used to vary the model in a fine-grained way. Model execution then happens in two steps: first, the model is configured based on the feature configuration, and then the configured model is processed as before via transformation, generation or interpretation.

Similar to configuration of source code, for example via the C preprocessor, the configuration of models can be done in several different ways. The actual implementation may be different depending on the DSL and language workbench:

- *Negative Variability via Removal* DSL program elements can be annotated with *presence conditions*, Boolean expressions over the features of a feature model. When the DSL program is mapped to the solution space, a model transformation removes all those elements whose presence condition is false based on the current feature configuration. Czarnecki and Antkiewicz show this approach applied to UML models [12], including static checks that ensure that every valid feature configuration leads to a structurally correct UML model.
- *Positive Variability via Superimposition* A set of prebuilt model fragments is created. The feature configuration selects a subset of them. The fragments are then merged using some DSL-specific or generic merge operator, resulting in a superimposed model representing the variant. Apel et al discuss the approach in general and demonstrate it for various UML diagrams, among them class diagram, state diagrams and sequence diagrams. [13].
- *Positive Variability via Aspects* A core program is available, together with a set of aspects. The aspects use pointcuts to define where and how they affect the core program. Based on the feature configuration, a subset of these aspects is selected and applied to the core program. The case study in [14] describes this approach applied to EMF models (Eclipse Modeling Framework, http://eclipse.org/emf).

As Fig. 3.4 shows, this approach reduces the numbers of variation points in the artifacts and automatically supports the consistent implementation of several, non-local adaptations. Since the models are transformed into lower-level artifacts automatically, they "expand" the variability to potentially many low-level variation points.

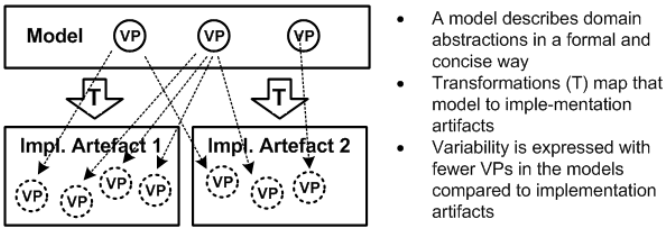


Figure 3.4: Transformations "expand" variation points, thereby reducing the number of variation points that have to be management explicitly.

3.3.3 *Variations in the Transformation or Execution*

When working with DSLs, the execution of models — by transformation, code generation or interpretation — is under the control of the domain engineer. The transformations or the interpreter can also be varied based on a feature model.

- *Negative Variability via Removal* The transformations or the interpreter can be annotated with presence conditions, the configuration happens before the transformations or the interpreter are executed.

- *Branching* The interpreter or the transformations can query over a feature configuration and then branch accordingly at runtime.

- *Positive Variability via Superimposition* Transformations or interpreters can be composed via superposition before execution. For transformations, this is especially feasible if they transformation language is declarative, which means that the order in which the transformations are specified is irrelevant. Interpreters are usually procedural, object-oriented or functional programs, so declarativeness is hard to achieve in those.

- *Positive Variability via Aspects* If the transformation language or the interpreter implementation language support aspect oriented programming, then this can be used to configure the execution environment. For example, the Xpand code generation engine (http://wiki.eclipse.org/Xpand) supports AOP for code generation templates.

Examples for all of these are described by Voelter and Groher [15] based on the openArchitectureWare tool suite[16].

Creating transformations with the help of other transformations or by any of the above variability mechanisms is also referred to as *higher-order transformations* [17]. Note that if a bootstrapped environment is used, the transformations are themselves models created with a transformation DSL. This case then reduces to just variation over models, as described in the previous subsection.

[15]

[16] openArchitectureWare has since been migrated to the Eclipse Xpand and Xtext projects

[17]

3.3.4 *DSLs as Attribute Types*

Some feature modeling tools support feature attributes. Typically, the types of these attributes are primitive (integer, string, float). They could also be typed with a DSL, the set of valid programs expressed in this DSL would be the range of values.

This approach is useful when the primary product definition can be expressed with a feature model. The DSL-typed attributes can be used for those variation points for which selection is not expressive enough.

3.3.5 *Feature Models on Language Elements*

The opposite approach is also possible. The primary product definition is done with DSLs. However, some language elements may have a feature model associated with them for detailed configuration. When the particular language concept is instantiated, a new ("empty") feature configuration is created, and can be configured by the application engineer.

3.3.6 *Merging of the two approaches*

We have described the limitations of the feature modeling approach. The feature modeling community is working on alleviating some of these limitations.

For example, cardinality based feature models [18] support the multiple instantiation of feature subtrees. References between features could be established by using feature attributes typed with another feature — the value range would be the set of instances of this feature. Name references are an approximation of this approach.

[18]

Clafer [19] combines meta modeling and feature modeling. In addition to providing a unified syntax and a semantics based on sets, Clafer also provides a mapping to SAT solvers to support validation of models. The following code is Clafer code (adapted from Michal Antkiewicz' *Concept Modeling Using Clafer* tutorial at http://gsd.uwaterloo.ca/node/310).

[19]

```
abstract Person
  name : String
  firstname : String
  or Gender
     Male
     Female
  xor MaritalStatus
    Single
    Married
    Divorced
  Address
    Street : String
    City : String
    Country : String
    PostalCode : String
    State : String ?

abstract WaitingLine
  participants -> Person *
```

The Clafer code example describes a concept *Person* with the following characteristics:

- a name and a first name of type *String* (similar to attributes)
- a gender which is *Male* or *Female*, or both (similar to or-groups in feature models)

- a marital status which is either *single*, *married* or *divorced* (similar to xor-groups in feature models)
- an *Address* (similar composition is language definitions)
- and an optional *State* attribute on the address (similar to optional features in feature modeling)

The code also shows a reference: a *WaitingLine* refers to any number of *Persons*.

Note, however, that an important ingredient for making DSLs work in practice is the domain-specific concrete syntax. None of the approaches mentioned in this section provide customizable syntax. However, approaches like Clafer are a very interesting backend for DSLs to support analysis, validation and automatic creation of valid programs from partial configurations (Section 3.7).

3.4 *Benefits of Tools*

If all product line artifacts, i.e. program code, models, and transformations, are expressed using a single modeling infrastructure, then only a single approach is needed for varying any of them. In addition, the integration between different models representing different aspects of the overall product configuration becomes simpler. For example, the ability to refer to features in a feature model from an Xtext-based language is a generic, reusable module that can be integrated into arbitrary DSLs. Similar tooling is available for JetBrains MPS. Arbitrary program elements expressed with any language defined in MPS can be annotated with presence conditions (Fig. 3.5). Upon transformation, all those model elements whose presence condition is false at the time of generation are removed, so no lower level code is generated from them. It is also possible to configure the program for a specific variant while it is edited.

A similar approach can be used for expressing traceability, another important cornerstone of PLE [20]. Arbitrary model elements can be annotated with traceability links to a requirements database. For Eclipse, the VERDE project (http://www.itea-verde.org/) develops generic traceability tooling with which any EMF-based model can be related to other models or RIF-based requirements files. In MPS this is possible as well, using the same annotation-based approach that is used for presence conditions. MPS' capability to combine independently developed languages makes the composition of an overall product configuration from program/model fragments expressed in various lan-

[20] W. Jirapanthong and A. Zisman. Supporting product line development through traceability. In *apsec*, pages 506–514, 2005

```
{bumper} int8 bump = 0;
{bumper} bump =
         ecrobot_get_sensor(SENSOR_PORT_T::NXT_PORT_S3);
{bumper} if ( bump == 1 ) {
  {debugOutput &&!bumper} debugString(3, "bump:", "BUMP!");
  event linefollower:bumped
  terminate;
}
{sonar && debugOutput} if ( currentSonar < 150 ) {
  event linefollower:blocked
  terminate;
}
```

Figure 3.5: Presence conditions (Boolean expressions over features) on program elements in the MPS tool. Presence conditions can be attached to any program node and control which nodes are transformed during code generation.

guages almost trivial. Language composition in MPS and the annotation mechanism is explained in detail in [21].

[21] M. Voelter and K. Solomatov. Language modularization and composition with projectional language workbenches illustrated with MPS. In M. van den Brand, B. Malloy, and S. Staab, editors, *Software Language Engineering, Third International Conference, SLE 2010*, Lecture Notes in Computer Science. Springer, 2010

3.5 Examples

This section contains industry examples in which DSLs are used to implement product lines. Note that we cannot reveal the actual companies using the DSL, and in case of the fountains, which we had already introduced in Section 3.2, we even had to move the example into another domain. However the cases and the languages are real world examples.

3.5.1 Alarm System Menus

The company manufactures burglar alarm systems. These systems detect when buildings are compromised. They consist of sensors that detect the burglary, and actuators including sirens, lights, and alarm propagation facilities to the police. These systems also have configuration devices, used by the house owner to configure, among other things, when the system should be active, and which kinds of alarms should be raised under which conditions.

The company sells many different alarm systems, sensors, and actuators. Based on the configuration of an individual system, the menus in the configuration device need to be adapted. Traditionally these menus have been described using Word documents, developers then implemented the menus in C.

A new approach uses a DSL that formally describes the menu structure. Code generation creates the C implementation. Fig. 3.6 shows some example code. The language is purely structural, no behavior is described explicitly. A templating mechanism is provided that supports multiple instantiations of the same template in several locations in the tree, configuring each instance with different values passed into the template instance. Menus can also inherit from other menus to avoid code duplication for related systems.

While the DSL is relatively simple, it plays an important role nonetheless, because product management can directly describe the menus in a formal way, making the overall development process much simpler, faster, and less error prone.

We have used a DSL instead of a feature model for the following reasons: menus require recursion in the underlying formalism to be able to define unlimited numbers of instances of submenus. The ability to define standalone submenus that can be included in other menus is an example of references. Finally, the various elements have many fine

```
menu Normal label "Standardmenue"
    submenu autoLocking label "Automatic Locking"
        item startTime sys(TurnOnAlarm)
            valuerange Time
        item endTime sys(TurnOffAlarm)
            valuerange Time
        template areaSettings [size=15, area=1,
                sw=sys(TurnOnAlarm)] area1Settings
        template areaSettings [size=10, area=2,
                sw=sys(TurnOnAlarm)] area2Settings

    end
end

template [size: int, area: int, sw: swref] areaSettings
    item onOrOff sys(TurnOffAlarm) labelexpr "Autolock "
                                    +area+" on/off"
        bool true = label(size) "On" false = label "Off"
    item test sys(AlarmLevel) label "Test"
        bool
    item alarmLevel sys(AlarmLevel)
        valuerange SoundLevel restrict size..80
end

menu Expert extends Normal
    item master sys(UnlockNow) afterItem unlockNow bool
end
```

Figure 3.6: Example menu definition for the alarm systems. Menus contain submenus and items. Templates can be defined, and template instances can be embedded into a menu tree several times with different parameter values.

grained attributes. A textual notation works much better in these cases than trees.

3.5.2 *Fountains*

This is the example used in section 3.2. Before using DSLs, fountain designers experimented with different arrangements of basins and pumps, writing down, in prose text, interesting configurations and behaviors. Developers wrote the corresponding controller code in C. The domain from which this example is derived is very complex, with ca. 700 different products!

Several DSLs are used. The first one (Fig. 3.7) is used to describe the logical structure of basins, pumps and valves. It uses a form of multiple inheritance similar to classes (here: appliances) and traits (here: features), quite similar to what Scala provides (http://scala-lang.org).

```
feature BasicOnePump
    pump compartment cc1
        static compressor c1

feature AtLeastOneZone extends BasicOnePump
    water compartment comp1
        pumped by c1
        compartment levelsensor ct_f1
        light l_f1

feature[f] SuperPowerCompartment
    water compartment adds to f
        superPowerMode

feature WithAlarm
    level alarm a1

fountain StdFountain extends AtLeastOneZone
```

Figure 3.7: Fountains can be composed from features, who contribute hardware elements. Parametrized features (those with brackets) can be included more than once, binding the parameter differently each time.

A second language (Fig. 3.8) is used to describe the behavior. The behavior model refers to a hardware structure. All the hardware elements have events, properties and commands defined, which can be accessed from the behavior model. A state based, reactive, asynchronous language is used to describe the behavior, driving the activation of the pumps and valves based on sensor input and timing events.

In addition to generating C code, there is also an in-IDE interpreter that can run the pumping programs and execute tests. These tests are also described with a textual language. A simulation engine to "play" with the programs is available as well. This is an example where additional tools make the use of DSLs much more feasible for domain experts.

The behavior language also overlays configuration over the pumping behavior, an example of negative variability of a model (Section 3.3). The behavior can be varied depending on whether certain optional hardware components are installed in the fountain, as shown in Fig. 3.9.

We have used DSLs in this case because algorithmic behavior is

```
pumping program P1 for AtLeastOneZone + WithAlarm +
                       SuperPowerCompartment[f=comp1]  {
    parameter defaultWaterLevel : int
    parameter superWaterLevel: int
    event superPowerTimeout

    init {
        set comp1->targetHeight = defaultWaterLevel
    }

    start:
        on (comp1->needsPower == true) && !(comp1->isPumping) {
            do comp1->pumpOn
        }
        on comp1->enough {
            do comp1->pumpOff
        }
        on comp1.superPumping->turnedOn {
            set comp1->targetHeight = superWaterLevel
            raise event superPowerTimeout after 20
        }
        on comp1.superPumping->turnedOff or superPowerTimeout {
            set comp1->targetHeight = defaultWaterLevel
        }
}
```

Figure 3.8: The fountain behavior is defined with a reactive, asynchronous language. A pumping program is defined for a combination of hardware features. The properties, events and commands of the hardware elements defined by these features can be used in the programs.

described. The expressions used in the language cannot sensibly be represented with feature models. The hardware structure language has to be able to instantiate the same hardware component several times, another reason for using a DSL instead of a feature model. We use negative variability to overlay hardware structure dependencies over the behavior specifications.

```
every 10 {
    variant WithAlarm {
        if ( comp1->currentHeight > alarmLevel ) {
            do a1->ring
            set alarmActive = true
        }
        if alarmActive {
            if comp1->currentHeight > alarmLevel {
                do a1->stop
                set alarmActive = false
            }
}   }   }
```

Figure 3.9: Within a pumping program, *variant* statements can be used to implement negative variability over optional hardware features. The example shows code that is only executed if the *WithAlarm* feature is present.

3.5.3 *Architecture DSLs*

Many product lines are built on a common software architecture, while the application functionality varies from product to product. Architecture DSLs [22] can be used very effectively in these cases. An architecture DSL is a DSL, in which the abstractions of the language correspond to the architectural concepts of the execution platform. They are defined by architects, and used by developers as they develop applications. When developing products in challenging environments such as distributed real-time embedded systems, architecture DSLs can provide the benefit of simulation and automatic optimization as described by Balasubramanian and Schmidt in [23].

[22] ; and

[23]

One project in the transportation industry has used an architecture DSL overlaid with configuration-based variability. The architecture models directly refer to the features defined in the configuration model, a form of presence conditions. Tooling is based on Eclipse Xtext and pure::variants. Fig. 3.10 shows a screenshot. The system also supported AOP: aspects are available to "contribute" additional properties to existing model elements. The article in [24] describes this example, and the general approach, in more detail.

[24]

A DSL was used in this case because architecture definition makes heavy use of identities and references, attributes and recursion. For example, it must be possible to define any number of components, and these than have to be instantiated and connected. So even while no expressions are used, DSLs are still useful in this case.

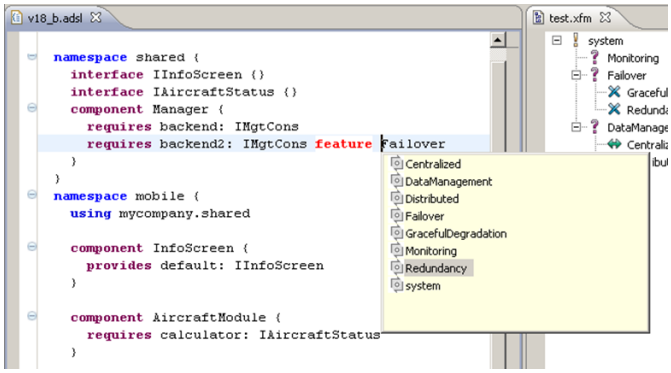


Figure 3.10: Tool support (Eclipse Xtext and pure::variants) for referring to feature models from DSLs, implementing negative variability over DSL code.

3.6 Conceptual Mapping from PLE to DSLs

This section looks at the bigger picture of the relationship between PLE and DSLs. It contains a systematic mapping from the core concepts of PLE to the technical space of DSLs. First we briefly recap the core PLE concepts.

- *Core Assets* designate reusable artifacts that are used in more than one product. As a consequence of their strategic relevance, they are usually high quality and maintained over time. Some of the core assets might have variation points.
- *A Variation Point* is a well-defined location in a core asset where products differ from one another.
- *Kind of Variability* classifies the degrees of freedom one has when binding the variation point. This ranges from setting a simple Boolean flag over specifying a database URL to a DSL program to a Java class hooked into a platform framework.
- *Binding Time* denotes the point in time when the decision is made as to which alternative should be used for a variation point. Typical binding times include source time (changes to the source code are required), load time (bound when the system starts up) and runtime (the decision is made while the program is running).
- *The Platform* are those core assets that actually form a part of the running system. Examples include libraries, frameworks or middleware.
- *Production Tools* are core assets that are not part of the platform, but are used during the possibly automated development of products.
- *Domain Engineering* refers to activities in which the core assets are created. An important part of domain engineering is domain analysis, during which a fundamental understanding of the domain and its commonalities and variability is established.
- *Application Engineering* is the phase in which the domain engineering artifacts are used to create products. Unless variation points use runtime binding, they are bound during this phase.
- *The Problem Space* refers to the application domain in which the product line resides. The concepts found in the problem space are typically meaningful to non-programmers as well.
- *The Solution Space* refers to the technical space that is used to implement the products. In case of *software* product line engineering, this

space is software development. The platform lives in the solution space. The production tools create or adapt artifacts in the solution space based on a specification of a product in the problem space.

In the following sections we now elaborate on how these concepts are realized when DSLs are used.

3.6.1 *Variation Points and Kinds of Variability*

This represents the core of the paper and has been discussed extensively above: DSLs provide more expressivity than feature models, while not being completely unrestricted as programming languages.

3.6.2 *Domain Engineering and Application Engineering*

As we develop an understanding of the domain, we classify the variability. If the variability at a particular variation point is suitable for DSLs, we develop the actual languages together with the IDEs during domain engineering. The abstract syntax of the DSL constitutes a formal model of the variability found at the particular variation point. This is similar to analysis models, with the advantage that DSLs are executable. Users can immediately express exemplary domain structures or behavior and thereby validate the DSL. This should be exploited: language definition should proceed incrementally and iteratively, with user validation after each iteration. The example models created in this way should be kept around, they constitute unit tests for the language.

The combination of several DSLs is often necessary. Different variation points may have different DSLs that must be used together to describe a complete product. In the fountains example, one DSL describes the hardware structure of the fountains, and another one describes the behavior. The behavior DSL refers to elements from the hardware DSL for sensor values and events. In this case, one language merely refers to an element of another language. Deeper integration may also be necessary. For example, we may want to embed a reusable expression language into the fountain behavior DSL. Different language workbenches support language composition to different degrees. References between languages are always possible. Language embedding is not yet mainstream. It supported for example by Spoofax [25] and MPS [26].

Application engineering involves using the DSLs to bind the respective variation points. The language definition, the constraints, and the IDE guide the user along the degrees of freedom supported by the DSL.

[25] L. C. L. Kats and E. Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *OOPSLA*, pages 444–463, 2010

[26] M. Voelter and K. Solomatov. Language modularization and composition with projectional language workbenches illustrated with MPS. In M. van den Brand, B. Malloy, and S. Staab, editors, *Software Language Engineering, Third International Conference, SLE 2010*, Lecture Notes in Computer Science. Springer, 2010

3.6.3 *Problem Space and Solution Space*

DSLs can represent any domain. They can be technical, inspired by a library, framework or middleware, expected to be used by programmers and architects. DSLs can also cover application domains, inspired by the application logic for which the application is built. In this case they are expected to be used by application domain experts. In the case of application DSLs, the DSL resides in the problem space. For execution they are mapped to the solution space by the production tools. Technical DSL can, however, also be part of the solution space. In this case, DSL programs are possibly created by the mapping of an application domain DSL to the solution space. This is an example of cascading [27]: one DSL is executed by mapping it to another DSL. It is also possible that technical DSLs are used by developers as an annotation for the application domain DSLs, controlling the mapping to the solution space or configuring some technical aspect of the solution directly [28].

[27] ; and

[28]

3.6.4 *Binding Time*

DSLs can be executed in two ways. *Transformation* maps a DSL program to another formalism for which an execution infrastructure already exists. If this formalism is another DSL, we speak of model-to-model transformation, or simply transformation. If the target is a programming language, we speak of code generation. Alternatively, the DSL can also be interpreted: a meta program that is part of the platform executes the DSL program directly.

- If we generate source code that has to be compiled, packaged and deployed, the binding time is source. We speak of static variability, or static binding.
- If the DSL programs are interpreted, and the DSL programs can be changed as the system runs, this constitutes runtime binding, and we speak of dynamic variability.
- If we transform the DSL program into another formalism that is then interpreted by the running system, we are in a middle ground. It depends on the details of how and when the result of the transformation is (re-)loaded into the running system whether the variability is load-time or runtime.

When to use transformation vs. interpretation depends on various, usually non-functional concerns. Transformation and code generation is the more mainstream approach because most people find it easier to implement and debug. It has a couple of advantages compared to interpretation, better performance being the most important one,

especially in embedded systems. Interpretation is intriguing because of the fast turnaround time. A detailed discussion of the trade-offs is beyond the scope of this paper.

3.6.5 *Core Assets, Platform and the Production Tools*

DSLs constitute core assets, they are used for many, and often all of the products in the product line. It is not so easy to answer the question whether they are part of the platform or the production tools:

- If the DSL programs are transformed, the transformation code is a production tool. It is used in the production of the products. The DSL or the models are not part of the running system.

- In case of interpretation, the interpreter is part of the platform. Since it directly works with the DSL program, the language definition becomes a part of the platform as well.

- If we can change the DSL programs as the system runs, even the IDE for the DSL is part of the platform.

- If the DSL programs are transformed into another formalism that is in turn interpreted by the platform, then the transformations constitute production tools and the interpreter of the target formalism is a part of the platform.

3.7 *Future Work*

Hybrid solutions such as Clafer and their relationships to DSLs require further investigation. One question is whether Clafer is suitable as a verification backend for DSLs: a DSL's abstract syntax, including possible selection-based parts, could be translated into Clafer, and then Clafer's integration with solvers could be used to verify and check DSL programs. This approach promises a simplification over directly integrating DSLs with solvers, because Clafer provides abstractions that are much closer to DSLs than raw logic. A second, related question is which kinds of languages are suitable for this kind of integration. A third question relating to Clafer is whether it could be directly used as the abstract syntax formalism for DSLs. In this case, Clafer models would be annotated with concrete syntax definitions to render a textual DSL.

3.8 Conclusion

In this paper we have positioned domain-specific languages into the context of PLE. Our practical experience shows that DSLs play an important role in PLE, filling the expressive gap between feature models and programming languages. DSLs can provide problem space-level, formalized descriptions of the core application logic, that is hard to capture with feature models. We show that DSLs fit well with the existing feature model-based PLE approaches and the overall PLE approach. Tool support for DSLs, and for the integration between DSLs and feature models this is coming along. Finally, it is worth pointing out that language workbenches are becoming more and more powerful and user friendly, making the development of DSLs and their tools much less effort than language construction has historically been. The language workbench competition webpage at http://www.languageworkbenches.net/ provides a good overview.