Architecture as Language

Article *in* IEEE Software · March 2010

DOI: 10.1109/MS.2010.38 · Source: doi.ieeecomputersociety.org

CITATIONS
12

READS
627

1 author:

Markus Völter
independent/itemis
148 PUBLICATIONS **4,239** CITATIONS

SEE PROFILE

Architecture as Language

Markus Völter – Final Version

Software Architecture Today

Software architecture is a somewhat funny thing. Everybody in software development agrees that we need it in some way, shape or form. However, we cannot agree on a definition, we don't really know how to manage it efficiently in non-trivial projects and we usually don't have ways to express a system's architectural abstractions precisely and concisely. Asking some of my customers to describe a system's architecture, I get responses that include specific technologies, buzzwords (such as AJAX or SOA) or vague notions of "components" (like *publishing*, *catalogue* or *payment*). Some have wallpaper-sized UML diagrams, the meaning of the boxes and lines not being really clear. All of these things are aspects of the architecture of their system. But none of those explanations represent a concise, unambiguous and "formal" description of the core abstractions of a system.

This is not necessarily surprising since we don't have a way of expressing directly the architectural abstractions of a system. We don't have a *language* that directly expresses software architecture. As we all know, if we don't have a language to express something, we have a hard time grasping it. Today's mainstream programming languages don't even have a way to express building blocks larger than classes and the relationships between them, the very basics of software architecture.

To address this issue, in this article I explain the usefulness of using domain specific languages (DSLs) for describing, managing and validating software architecture. The approach centers around the practice of building an architecture/system/platform-specific DSL that captures the core architectural abstractions of the particular architecture/system/platform, and using code generation to implement it consistently.

At the time of writing of this article, I had implemented the approach with various customers from various domains, including embedded control systems, finance and large-scale web applications. I cannot name the customers here, but they are global players in their respective domains. The projects aim at rebuilding core platforms that will be used over at least the next decade – explaining the drive towards a stable, well-defined and technology-agnostic architecture definition – in a decade technology goes through at least 3 hype cycles.

My Notion of Software Architecture

So what is software architecture? Many definitions are available in the industry (a nice collection is at [AD]). My own working definition of software architecture could be seen as a "best of" of many of these

definitions: software architecture is *all those aspects of a system that we want to be consistent throughout the system* for reasons ranging from meeting non-functional requirements to technology best practices to maintainability to developer training. The definition also implies that there are aspects of a system for which there's no need for system-wide consistency. As the person (or team) responsible for architecture, I don't care about how those aspects are implemented, as long as they fulfill stated requirements.

My working definition does not state anything about granularity. There might be certain nitty-gritty details of the locking protocol used in a concurrent system that I consider part of the system architecture, because if they are not implemented consistently, the overall system might encounter resource contention.

Here is how I break down software architecture in practice. There's what I call the conceptual architecture, it defines the concepts used to describe the system on an architectural level, as well as the relationship between these concepts. Example concepts include *component, message queue, interface, async RPC call* or *actor.* Application architecture uses instances of these concepts to define a concrete system. For example, there might be a component C*ustomerManagement* that implements the *CustomerLookup* interface. Once the conceptual architecture is defined, we can use the non-functional requirements to determine a technology mapping for these concepts. For example, components can be mapped to EJBs or WCF components. Finally, we define a programming model, the way how developers express the application architecture – this includes ways (to stick with the examples above) to describe and implement components, an API to create messages and post them to a queue or the actual protocol for acquiring and locking resources.

Of course all of this should be developed incrementally, by down-to-earth members of the team (not ivory-tower architects). However, I do think that even in an agile/iterative/incremental development approach you want to make sure that at any given time the system conforms to whatever is defined as *the architecture*. The notion of *the architecture* evolves over time, but at any relevant point in time (for example an incremental release) you want the system to be consistent. Although this article is not on process, I find this aspect very important to keep in mind.

Architecture DSLs

So how does all of this relate to DSLs? An Architecture DSL (ADSL) is a language that expresses a system's architecture directly. *Directly* means that the language's abstract syntax contains constructs for all the ingredients of the conceptual architecture. The language can thus be used to describe a system on architectural level. 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 additional frameworks.

To enable architecture model checking and validation, and to support meaningful code generation, the Architecture DSL has to be defined formally – meaning that the semantics are unambiguous, and checkers and generators are able to process the models. "Just Pictures" is not enough.

For humans to be able to use the language a suitable concrete syntax must be defined. *Suitable* means that it must efficiently support the abstraction of architectural concerns by developers. It must also integrate well with existing development tools, specifically, source code management systems.

I want to (re)emphasize an important point: I do not advocate the definition of a generic, reusable language such as the various ADLs, or UML (see below). Based on my 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 architectural abstractions – this goes back to the notion above: now that you *have* a language to express architectural aspects of a system, this helps you reason and discuss about the architecture.

An Example

This section contains an example of an Architecture DSL I have implemented for a real system together with a customer from the domain of airport management systems.

The example, as well as the real-world systems mentioned in the first section uses textual syntax to express the architecture. I will argue below why this decision has been made.

So I was with a customer for one of my 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 systems. Often you use Java technology for the backend, and .NET technology for a Windows frontend. My customer 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 I arrived they briefed me 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 turned out quickly 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 over disagreements and misunderstandings all the time. 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 started with the notion of a component. At that point the notion of components is defined relatively loosely. It's simply the smallest architecturally relevant building block, a piece of encapsulated functionality. We also assumed that components can be instantiated, making components the architectural equivalent to classes in OO programming[1]. To enable components to interact, we also introduced 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.

```
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.

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 have not yet been made. We also introduced connectors to define actual communication paths between components (and their ports).

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

At some point it became clear that in order not to get lost in all the components, instances and connectors we need to introduce some kind of namespace. And we'd need to distribute things to different files (the tool support makes sure that *go to definition* and *find references* still works).

---

[1] Coming up with the concrete set of components, their responsibilities, and consequently, their interfaces is not necessarily trivial either. Techniques like CRC cards can help here [CRC].

```
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*.

```
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 one by one. So we introduced dynamic connectors: we specify a query that is executed at runtime against some kind of naming/trader/lookup/registry infrastructure. It is reexecuted every 60 seconds to find the *InfoScreens* that had just come online.

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

A similar approach can be used to realize load balancing or fault tolerance. A static connector can point to a primary as well as a backup instance. Or a dynamic query can be reexecuted when the currently used component becomes unavailable.

To support registration of instances, we add additional syntax to their definition. A *registered* instance 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*.

```
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). We also defined several message interaction patterns, here are examples of *oneway* and *request-reply*:

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

We talked a long time about suitable message interaction patterns. After a while it turned out that a core use case 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 problem 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:*

- 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 communication abstraction in the system, we added this to our Architecture DSL and were able to write something like the following.

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

replicated singleton flights {
  flights: FlightInfo[]
}

component DelayCalculator {
  publishes flights { publication = onchange }
}

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

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).

Data replication 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. The description also much more clearly reflects the actual architectural intent: it expresses better *what* we want to do (replicate data) compared to a lower level description of *how* we want to do it (sending 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 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.

```
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
    }
  }
}
```

Initially, the protocol state machine is in the *new* state. The only valid message is *registerAircraft*. Once 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 then 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. This is why no ports and stuff are declared. For all

intents and purposes, it's the same thing. Just maybe a couple of bugs are fixed.

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

If you really want to evolve a component (i.e. change its external signature you can write it like this:

```
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.

What we did in a nutshell

The approach shown above proposes the definition of a formal language for a system's conceptual architecture. You develop the language as the understanding of your architecture grows, in realtime: the example above has been built during a two-day architecture exploration workshop. The language therefore always resembles the complete understanding about your architecture in a clear and unambiguous way.

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.

As we enhance the language, we also describe the application architecture using that language. We are building Architecture DSLs.

DSLs can be used to specify any aspect of a software system. The big hype is around using a DSL to describe the business functionality (for example, calculation rules in an insurance system). While this is a very worthwhile use of DSLs, it is also worthwhile to use DSLs to describe software architecture.

Benefits

All involved will have clear understanding of the concepts used to describe the system, there is an unambiguous vocabulary to describe applications.

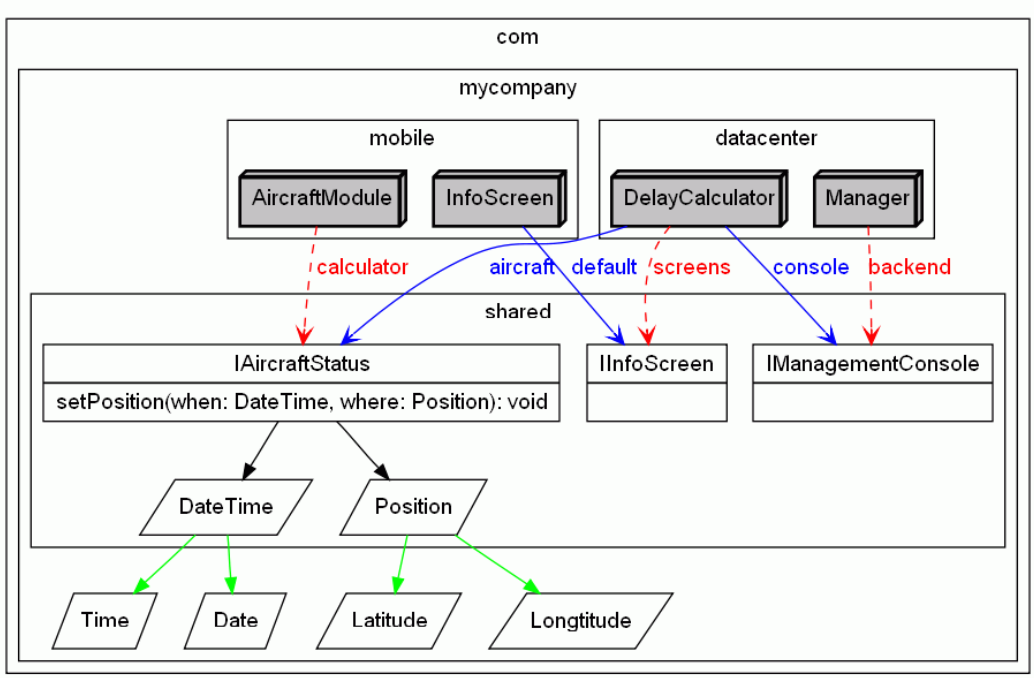
Models can be analyzed and used as a basis for code generation (see below). The architecture is freed from implementation details, or in other words: conceptual architecture and technology decisions are decoupled, making both easier to evolve. We can define a clear programming model based on the conceptual architecture. Last but not least, the architect(s) can contribute directly to the project, by building (or helping to build) the languages and related tools – an artifact that the rest of the team can actually use. In a very real sense, this could be called executable architecture.

Why textual?

Textual DSLs have several advantages. Here are some of them:

- Languages as well as editors are easier to build compared to custom graphical editors (although the validity of this statement depends on the tooling used)
- Textual artifacts integrate much better with existing developer tooling than repository-based models. You can use well-known diff/merge tools, and it is much easier to version/tag/branch *models and code together*. Generally, the tooling is more lightweight.
- Model evolution (i.e. adaptation of models when the DSL evolves over time) is much simpler. While you can use the standard approach – a model-to-model transformation from the old to the new version – you can always use search/replace or grep as a fallback, technologies familiar to everybody.

Where a graphical notation is useful to see relationships between architectural elements, you can use tools like Graphviz [GV] or Prefuse [PF]. Since the model contains all the relevant data, you can easily transform the model into a format those tools can process. The following is an example of a graphviz-generated diagram. It shows *namespaces*, *components*, *interface*s and *datatypes* as well as their relationships. It has been created by transforming the textual model into a dot file, graphviz' way of describing graphs.



Tooling

There are several tools available that support the definition of DSLs – textual or graphical. What those tools have in common is support for the definition of abstract and concrete syntax, as well as for user-friendly editors. In the textual category, you might want to look at Eclipse/TMF Xtext/openArchitectureWare [oAW] the Microsoft Oslo [OSLO] toolkit Jetbrains MPS [MPS], or Intentional's Domain Workbench [IDW]. In the graphical world, take a look at MetaEdit+ [ME].

For reasons of space, the detailed discussion of how to use these tools is out of the scope of this article.

Validating the Model

We need validation rules that constrain the model even further than what we can express with the language grammar. Examples include name-uniqueness, type checks or versioning constraints. To implement such constraints, two preconditions are necessary. The constraint itself must be describable via a formal algorithm (expressed, for example in OCL). And the data needed to evaluate the algorithm must be available in the model. For example, if you want to verify whether a certain deployment scheme is feasible, you might have to put the available network bandwidth and the timing/frequency of messages as well as the size of primitive data types into the model (maybe in a separate file). While capturing this data sounds like a burden, this is actually an advantage: it is core architectural knowledge.

Generating Code

The primary benefit of developing and using the architecture DSL is to better understand concepts by removing any ambiguity and defining them formally. It helps you understand the system and get rid of unwanted technology dependencies. However, ultimately we are not interested in correct models, but in correct running systems. To make this possible, we use code generation. It simplifies and constrains application implementation..

- We generate an API against which the implementation is coded. That API can be non-trivial, for example taking into account the various messaging paradigms or data replication. The generated API allows components to be implemented in a way that does not depend on specific technologies (it does depend on a specific programming language, though). The generated API hides those from the component implementation code.
- We also generate the code that is necessary to run the components (incl. their technology-neutral implementation) on a suitable middleware infrastructure . We call this layer of code the technology mapping code (or glue code). It typically also contains configuration files for the target platform(s). Sometimes this requires additional "mix in models" that specify configuration details for the platform. As an additional benefit, the generators capture best practices in working with the selected technologies.

It is of course 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, for example for local testing). This nicely supports potential multi-platform requirements, and also provides a way to scale or evolve the infrastructure over time.

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, 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 DSLs mentioned earlier.

Be aware that the discussion in this section is only really relevant for application-specific behavior, not for all implementation code. Large amounts of implementation code are related to the technical infrastructure – 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, and 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.

However this *completely misses my point*! I have not experienced much benefit in shoehorning your architecture description into the (typically very limited) 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 case of UML that often requires an architecture-specific profile. Is a profiled UML still *standard*?

Note that I 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. I have done this in several projects. It works, but not as well by far as the approach explained in this paper. There are several reasons, some 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!
- Customizing UML tools (beyond different colors or icons for stereotyped elements) is restricted, cumbersome and tool-specific (i.e. non-standard). A lot of effort must be spent to make profiled UML look and feel as if it was a DSL. And of course, you still have to deal with the complex UML meta model when processing the models…
- 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 "analysis pictures"). Tool integration issues can make developer acceptance very challenging.

A few words on Process and People

Define the language iteratively. Build example models all the time, and keep a reference model (one the uses all parts of the language) current at times. Interleave language definition and generator construction, to verify you have all the data in the models you need for generating meaningful code.

Don't just define a nice language, but also a user-friendly editor for the application developers who will have to use the language. Although language and editor definition with the modern tools mentioned above is quite straight forward and much simpler that what you might know from lex/yacc or antlr, make sure the developers who define the language can "think meta"; for some people this can be a challenge.

Please document how to use the language, editor and code generator based on examples and walk-throughs. Nobody reads "reference manuals". In the documentation, also explain the architectural concepts expressed via the language. People will not understand the language unless they understand what the language describes! And please set aside some time for taking into account feedback (bugs, improvements) from the language users. Throwing something like a custom language "over the fence" and then not supporting it is a death sentence for the approach!

This paper does not contain a consistent methodology. I believe that

- a good idea proven in real projects (as suggested in this in this article)
- together with general software development common sense (such as iterative/incremental development and testing)
- as well as (architectural) experience (which I expect you to bring to the table)
- and of course learning how to use the language engineering tools

is enough to apply this approach successfully in the real world.

Challenges

The approach of course does have a couple of challenges. Let me point out some of them.

Although the tools for defining DSLs, editors and code generators are becoming better, there is some overhead involved in building the language tools. I don't count the analytical part of distilling the architectural concepts and defining the language as overhead, since you'll have to do that in some form anyway. But the construction of the tools is additional work. In my experience it pays off in most settings, but it is an initial investment that needs to be done.

Convincing developers and especially management to abandon the use of standards (UML, ADLs) for architecture description can be a challenge. To overcome it I like to do a small, but meaningful prototype and show the benefits of being in control of your language and not being constrained by a predefined language.

Another challenge is the fact that the semantics of the DSL are usually not formally specified. Typically you write documentation that explains the meaning of language constructs. The code generator also serves as a vehicle for semantics definition, since it maps the language constructs to executable concepts in a target language with known semantics. However, if you generate towards several languages, then you need *one* central semantics definition against which you validate the various implementations. In practice, such a definition is not feasible. To overcome this, you write tests.

Finally, if you have many related, but not identical systems there will be common parts between several architecture DSLs. Consequently, you might want to modularize your DSL into several combinable language modules. This "base language" can then be extended with the concepts that are specific to the system in question. However, because of limitations in today's language engineering tools, such an approach is still a bit of a challenge. I have written about my prototypical implementations here [FAM]. Tools like MPS [MPS] mandate another try, though.

Summary

The article describes how to use DSLs to describe architecture. I want to reemphasize the importance of defining the language specifically for your context, together with the architecture team as the architecture is built – a generic approach does not reap the same benefits. Also, experience with my customers shows one thing consistently: once familiar with the capabilities of today's tools, the approach is considered much more feasible compared to what they had initially thought.

References

[GV] Graphviz Visualozation Toolkit, http://graphviz.net

[PF] Prefuse Visualozation Toolkit, http://prefuse.org

[FAM] Markus Voelter, A family of Languages for Architecture Description, DSM Workshop, OOPSLA 2008, and http://www.voelter.de/data/pub/DSM2008.pdf

[oAW] openArchitectureWare, http://openarchitectureware.org

[OSLO] Microsoft, Oslo Initiative, http://msdn.microsoft.com/en-us/oslo/default.aspx

[MPS] Jetbrains MPS, http://jetbrains.com/MPS

[ME] Metacase, MetaEdit+, http://www.metacase.com

[AD] Carnegie Mellon University, Software Engineering Institute, http://www.sei.cmu.edu/architecture/published_definitions.html

[ADL1] Carnegie Mellon University, ACME, http://www-2.cs.cmu.edu/~acme/

[ADL2] EAST ADL, http://en.wikipedia.org/wiki/EAST-ADL

[ADL3] University of California, Irvine, xADL, http://www.isr.uci.edu/projects/xarchuci/

[ADL4] Institute for Software Intensive Systems, Vanderbilt University, CoSMIC, http://www.dre.vanderbilt.edu/CoSMIC/

[CRC] Beck, Cunningham, A Laboratory For Teaching Object-Oriented Thinking, http://c2.com/doc/oopsla89/paper.html

[IDW] Intentional Software, Intentional Domain Workbench, http://www.intentsoft.com