AN OPEN PLATFORM FOR SYSTEMS AND BUSINESS ENGINEERING TOOLS: COLLABORATIVE MODELING AND ANALYSIS AT SCALE

VERSION 1.8 ▪ DECEMBER 2, 2019 ▪ MARKUS VOELTER ▪ VOELTER@ACM.ORG

AS THE SIZE AND COMPLEXITY OF SYSTEMS AND BUSINESS ENGINEERING PROJECTS GROWS, THE TOOLS TO BUILD THE UNDERLYING MODELS HAVE TO BECOME MORE POWERFUL. WHILE ADEQUATE ALM, PDM AND SCM TOOLS EXIST, THE CORE MODELING INFRASTRUCTURES FALL SHORT OF WHAT IS NEEDED. THESE TOOLS HAVE TO SUPPORT A WIDE RANGE OF EXPRESSIVE LANGUAGES TO DESCRIBE THE MODELS, MUST ENABLE A DIVERSE RANGE OF ANALYSES, SCALE OUT IN TERMS OF MODEL SIZE AND USER COMMUNITY, SUPPORT VARIOUS FORMS OF COLLABORATION AND, FINALLY, PLAY WELL WITH TODAY'S WEB AND CLOUD INFRASTRUCTURE. THIS PAPER OUTLINES A PLATFORM FOR SUCH TOOLS IN THE HOPE OF ASSEMBLING A COMMUNITY OF STAKEHOLDERS WHO MIGHT DEVELOP, FINANCE AND ULTIMATELY USE SUCH A TOOL.

Acknowledgements: I have been sending this paper around to a lots of people in the MDE and SLE communities in order to collect feedback and improve the paper. While I am the author who tries to bring it all together, the following people have provided feedback on earlier versions of this paper and thus added significant content: Sascha Lisson, Daniel Ratiu, Sergej Kosczejev, Federico Tomassetti, Meinte Boersma, Tamas Szabo, Diederik Dulfer, Benoit Combemale, Bran Selic, Andreas Wortmann (RWTH), Andreas Wortmann (OHB), Ruth Breu, Steffen Zschaler. Thank you very much!

WHAT ARE SYSTEMS AND BUSINESS ENGINEERING TOOLS

Systems engineering focuses on the design of complex (systems of) systems such as aircraft, spacecraft, automobiles or powerplants. The main stakeholders are engineers of various disciplines. They rely on a wide range of modeling languages, the most widespread is (subsets of) SysML, Simulink or Modellica. SysML allows the definition of components, interfaces and their connections, the interactions of those components and the aspects of their behavior. However, more domain-specific aspects of a system must be described as well, for example the security analysis in a car, the realtime and resource consumption of a medical appliance or the thermal properties of a satellite. Here, SysML is not suitable out-of-the-box—it is too generic, both in terms of syntax, but also in terms of semantics.

Similar challenges occur in the business domain, which focuses on the design and implementation of complicated (computational) systems in law, government, finance or other business domains. The main stakeholders are business analysts who are responsible for implementing the domain logic that eventually ends up in cloud applications. Once they move away from writing prose in Word, they rely on a set of DSLs that capture the core abstractions of the domain, often based on a functional or declarative computational paradigm or a rule engine. Examples include the implementation of a country's tax rules, public benefits calculations, medical algorithms or pricing rules in a telecom company.

While these two fields don't seem to have much in common at first sight, they in fact do:

- more and more artifacts are (or should be) rigorously well-defined so that they can qualify as models
- models are large and will certainly grow even more in the future
- the systems are always defined collaboratively by many users
- they are complex and multi-disciplinary, so very different means of description are required by the different roles, disciplines or communities that contribute
- requirements or specifications in the form of (more or less unstructured) text are no longer suitable; a machine processable representation is crucial to perform analyses, simulations or to run them.

Sometimes technical and business aspects should even be modeled together: think about a complex technical system and the contracts under which it is leased to future users, or quality-of-service contracts for cloud systems.

Over the next years, we expect other domains to require similar kinds of tools: big data, the training of machine learning algorithms, design space exploration, decision support and tools in the context of digital twins. In terms of tool infrastructure, they all have the same requirements: non-programmers specify data, transformations and analyses on large amounts of structured data.

A FOCUS ON (MODELING) LANGUAGES

To build models effectively, users need suitable languages. For some aspects of a system, standard languages are suitable; for example, SysML is useful for defining the structural backbone of a technical system. And a functional programming language is a good core for a tool that defines business systems. However, *limiting the users to one particular, generic language is a dead end!* Every non-trivial domain has lots of specifics that must be captured in the models. If users are forced to use a generic language, they will express the aspects that are specific to the domain with generic language constructs: patterns, naming conventions, comments. This leads to bloated modes that are hard to analyze and process by tools. Because of the relatively weak semantics of generic languages like SysML it is easy to "express the wrong thing".

A better solution is to allow the users to *also* use domain-specific languages, languages that capture the essentials of whatever domain they work in with first-class concepts; idioms and conventions are not required, the semantics become clear.

This leads to the following consequence:

A platform for developing systems and business engineering tools must be unbiased regarding the languages used to define the models and the analyses that run on them. Instead it should provide the infrastructure for defining languages, for working effectively with large, multi-paradigm *models in a collaborative manner, and for integrating arbitrary model analysis and transformation services.*

The argument regarding the genericity of the platform might sound like a contradiction to my negative view of generic languages. However, it is not. I am arguing for *specific languages* that integrate syntactically and semantically on a generic platform that handles the non-functional requirements. As it happens, this has also been the idea behind my PhD thesis; it is titled Generic Tools, Specific Languages.

REQUIREMENTS TOWARDS THE PLATFORM

I split the requirements into two classes; those that apply from the perspective of the end user and those that consider the development or integration of additional languages. Let's start with those for the end users:

- **Scalable:** To be able to model whole aircraft, powerplants or the laws of a country and their evolution over time, the platform must scale to very large models, millions and billions of model elements; the large size of such models might also be because they are imported from other systems (for example CAD) or because they represent data collected from a running system (monitoring data). This size is similar to the scale of web applications like those from Google and Amazon. Current modeling tools have a hard time dealing with such scale.
- **Collaboration and Versioning:** The first level of collaboration features should resemble Google Docs because this is what every non-programmer knows and expects: joint editing of a shared document with immediate visibility of the collaborators' edits, selections and cursor movements. In addition, once users understand the need, advanced features such as locking, merging and branching plus some degree of transactional isolation (think: commit/pull in git) should be made available to users. For reasons of reproducibility, specific states of the model must be taggable in a unique and trustable way, with guarantees that this state may not be changed in the future. I do not think offline work (and later, deferred synchronization) is required; networks are becoming more and more ubiquituous.
- **Migration Support:** In an agile development environment, languages will evolve. Potentially a new iteration of a language will change the structure of models in a way that is incompatible with those of the previous iteration; existing models must be migrated. Robust support for (semi-)automated migrations is crucially important.
- **Roles, Views and Contexts:** In any non-trivial use of such a tool, different stakeholders will work with the model. They will care about different aspects or parts of the system, prefer different notations, and perform different tasks at different times. At the minimum, it must be possible to define user groups and permission sets to control who is allowed to view and modify which parts. This is especially crucial if the tool is to be used over supply chains or by different disciplines/departments in an organization. But beyond that, the user's role and usage context should also guide the notations, the actions in the UI, the workflow and the recommendations the tool proactively provides, as well as which parts of the overall model are modifiable by which tool.
- **Native to the Web:** Obviously, the editors must run in the browser. However, there is more to make the system a native citizen of the web: it must be possible to use URLs to interact with the data in the system in a meaningful way.

- **IDE Features:** We expect all the features known from modern software IDEs such as syntax highlighting, code completion, go to definition, find usages, error highlighting, refactorings and debugging or tracing.
- **Liveness:** Users expect error checking and analyses to be immediate, interactive. We do not want a "Rebuild Model" button where the tool goes away for 30 seconds and does its thing. But in addition, for models that represent executable systems, users also want to execute the program (or its test cases) immediately in the tool, a property called liveness. Liveness is major reason why everybody uses spreadsheets—a spreadsheet is always "live", all dependent values are updated when any one is changed by the user.
- **Growable:** The UI complexity of the tool must be able to grow with the users and the complexity of the language. It is not useful to force a 500-button/menu item UI on a user who uses only a very simple DSL. It is also very useful if an editor that modifies a part of the model can be embedded into a traditional web application; this will allow users who only occasionally contribute to the models (or just review them) to work directly with the modeling tool instead of with derived documents and reports. They will never install The Big Tool™ for this occasional use.
- **Continuous Integration:** In many scenarios the models are ultimately transformed into artifacts that become part of a (particular version of) the final system. This means that the platform has to play well with CI servers. For example, models have to be exportable so they can be shipped as part of the product, analyses and tests might have to be run as part of the product build process. It must also be possible to run (non-incrementally) certain analyses on a particular version of the model as part of a reproducible build; services must be triggerable by the CI server. More generally, the repository and its services must be accessible to external tools by modern APIs.

From the language engineering perspective, the system must fulfil the following requirements[1].

- **Multi-Paradigm:** For any given domain, a wide range of languages must be supportable. This includes different styles: prose-style languages for high-level requirements, declarative languages for structures and certain kinds of behavior, but also full-blown behavioral languages to express math, logic or control.
- **Multi-Notation:** The past couple of years (of my own experience but also I think of the community) have made it abundantly clear that *only* graphical/diagrammatic or *only* textual notations are not the solution. A tool must seamlessly support both, and more: tables or symbolic notations such as math or chemical formulas are required as well. Ideally the notations should be combinable, as in embedding textual notations in diagrams or tables (with full IDE support).
- **Language Composition:** Languages must be composable. It must be possible to embed an expression language in a state machine DSL, to use different languages to describe different viewpoints of a system, to "overlay" trace links over models expressed with any language. It must also be possible to extend languages to "grow" them towards a domain gradually.[2]
- **External Tools:** Ideally, all modeling needs might be addressed with languages native to the tool. However, there will likely always be a need to integrate existing modeling tools

[1] These requirements are very much driven by my experience with MPS; It is such an empowering tool that I cannot imagine going back to before MPS.
[2] Interestingly, the upcoming SysML 2 language has this idea built deep into its architecture. Now we need tools that realizes the potential behind this design!

(such as the ubiquitous Simulink or Doors); so a meaningful way of integrating existing tools is required. In addition, initially, existing IDEs like MPS or modeling tools such as MagicDraw might serve as the editor frontend for models.

TRANSFORMATIONS ARE THE KEY

Mathematically, things like scoping, type checking or the creation of a view can be seen as functions that take the current model as an input and compute types, scopes or the structure of a view. When we move this idea into the world of models (which are essentially typed graphs), these functions become transformations of these graphs. These transformations

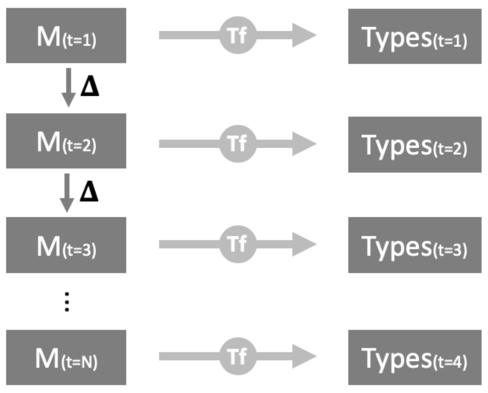
- create a derived model that embodies a particular aspect of the tool,
- in many cases we then run predefined algorithmic analyses on this model
- finally, we associate some kind of feedback (messages, data) with the original model element seen by the user

Let's see how this maps to the various "services" required to be run by a modeling tool:

- **Type checking:** we transform the AST into a model that represents the types associated with the AST nodes, we then run a type compatibility checker and produce messages that represent typing errors
- **Scoping:** we compute a "scope bucket" for each AST node, which is the set of target nodes visible from the current node's location in the AST. If the AST contains a reference that is not in this bucket, we annotate an error message.
- **Interpretation:** the transformation produces (and evolves) a model of the runtime state. Semantics definitions, which are effectively interpreters, are often literally defined as transformations.
- **Workflow and Recommendations:** Here, the basis is to capture the overall state of the model in a set of context-specific metrics; these are continuously collected and aggregated by analysis and transformations of the model at hand.
- **Views:** a derived view is obviously a transformation of a particular AST.
- **Reports:** these are obviously transformations to a "document-like" derived AST.

Often, several of these transformations might be pipelined. For example, we might first run a transformation of the AST that simplifies its structure (desugaring a functional program, flattening instantiation hierarchies, transforming a DSL to a state machine) before we then perform an "analyzing transformation" (interpreting the program, checking the flattened hierarchy for inconsistencies, model checking the state machine).

To keep the user experience interactive (as per the requirement above) the transformations (and ideally, the algorithmic analyses) must execute very quickly. The only way to make this work for non-trivial model sizes is to make them incremental; this means that the transformation processes the delta between the old and new model state to update the output instead of rerunning the complete transformation on the changed input.



The user edits their model M. Every edit operation (of a language-specific granularity) produces a change **Δ** that produces a new version of the model. The evolution triggers the transformations that produce derived models incrementally, such as the type system as shown above. Analyses run on the derived model, and messages are created. These are then associated to the nodes in the input model M.

The transformations will be unidirectional; bidirectional transformations have proven to be cumbersome to write and maintain because of the need to have a reversible semantic mapping; often this is impossible to find. Instead, the transformations maintain an internal trace that allows errors on the result of the transformation to be "piped back up" to a node in the source. In the relatively rare case that the transformation result should be editable a separate back-transformation has to be defined by the user.

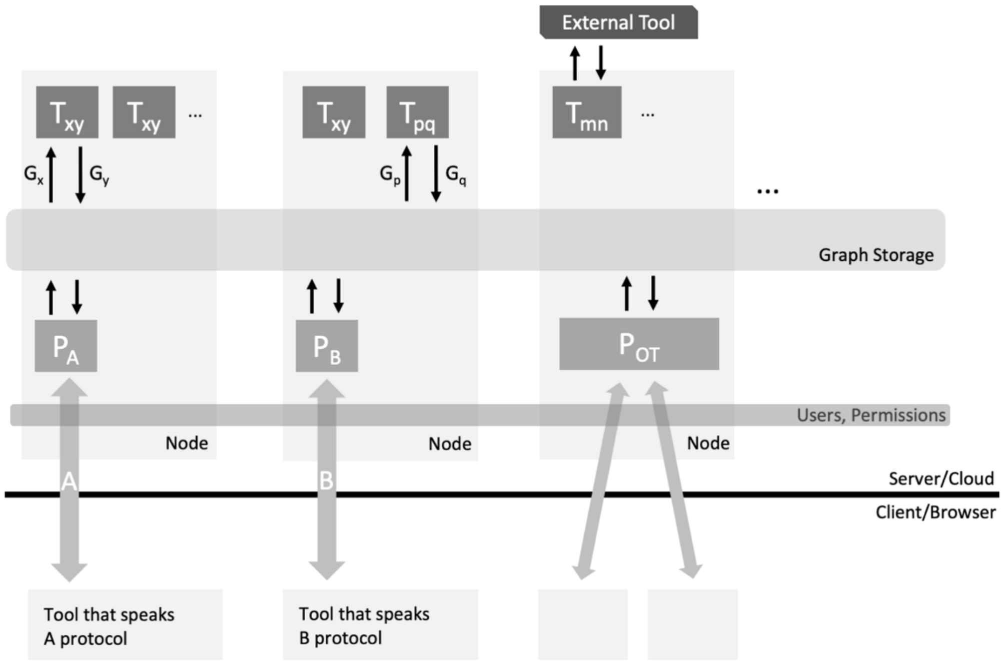
A note on code generation: I do not discuss the generation of textual artifacts in the context of this paper. The reason is that in contrast to all the other aspects discussed above code generation a) usually does not have to be interactive, but can done on demand, most likely by a CI server and b) there are lots of mature technologies available. More specifically, I would suggest to build a model exporter to Eclipse EMF and then use any of the existing EMF-based templating engines to generate the textual output. In the rare case where code generation must be incremental, it can be rethought as a regular, incremental transformation to the AST of the target language, plus simple unparsing.

THE CENTERPIECE: THE ACTIVE REPOSITORY

If we combine the focus on transformations with the needs for scalability and collaboration, we come to the conclusion that the core building block of a modern modeling platform is what I call an Active Repository. It stores arbitrary models as graphs (more specifically: trees with cross-references). A particular transformation "looks for" particular types of models in the storage; when one of them changes (through a delta initiated by a user edit or because another transformation updated it) it is triggered and performs its task of producing its output.

The reason why I am treating storage and transformation together in this paper is that, very likely, a particular storage structure is necessary to facilitate incrementality without too much overhead in terms of performance and storage. I would assume that immutable/persistent/functional data structures are used inside the repository. Every change conceptually creates a new micro-version of the whole tree. This way, global undo/redo is just about "moving back" in the list of versions. Technically, of course, the whole tree should not be physically copied; various database technologies and in-memory data structures are available that handle such efficient storage; ultimately these technical details are out of scope for this document.

Architecturally, the active repository can be seen as a version of the Blackboard pattern[3]:



The set of clients work on a shared repository. Clients are responsible for rendering models, allowing the user to interact with them, and render analysis results—plus caching parts of them to improve performance. A user's edit produces a change to the model, and either the delta or the "change command" is sent to the repository where it triggers the above-mentioned transformation cascade. In the illustration above, the boxes marked with T are transformations that read graphs of the a particular type (for example *x*) and transforms it to another kind of graph (such as *y*).

Each client subscribes to the regions of the model they are interested in; updates to those regions are sent back to the client using different protocols (such as P_A or P_B). The client does not just receive updates to the model they update, but also to the derived models that represent scope buckets, types or error messages; they use this information to drive the UI on the client.

To support the required collaboration scenarios, the repository is conceptually centralized. Whether is physically runs on one (reliable) machine, or is distributed over many machines in a cloud-setting or is distributed/replicated among the clients is still to be defined; my current hypothesis is that it runs as a set of cloud services.

A client might also be automated: a continuously running synchronizer from another system, or a "sensor" in the real world; this potentially leads to a high rate of change, bringing our system close to those from the realm of complex event processing.

[3] Buschmann et al., Pattern-Oriented Software Architecture, Vol. 1: A System of Patterns, 1996

The protocols between clients and the repository are similar in spirit to the Language Server Protocol. However, the protocols here would work on the abstract syntax and transport deltas. Different kinds of editors are able to produce different granularities of deltas. In graphical and projectional editors, every user action is directly available as a delta. In parser-based systems, we might want to treat (the equivalent of) files as one unit or rely on incremental parser and/or tree diffing techniques. The finer-grained the deltas the client sends, the more interactive the user experience. Client edits go through a conflict resolution layer likely based on Operational Transforms[4] to make sure several users can edit concurrently.

Note that the server does not store whole artifacts/files. Instead, the repository stores each node separately, it's effectively a graph database. Otherwise the "active" part of the Active Repository, the incremental transformations, are hard to build. Also, this architecture is not what is known as a Model Bus; the focus is not on exchanging models. Instead, the focus is on "doing the actual work" in the form of transformations in the repository.

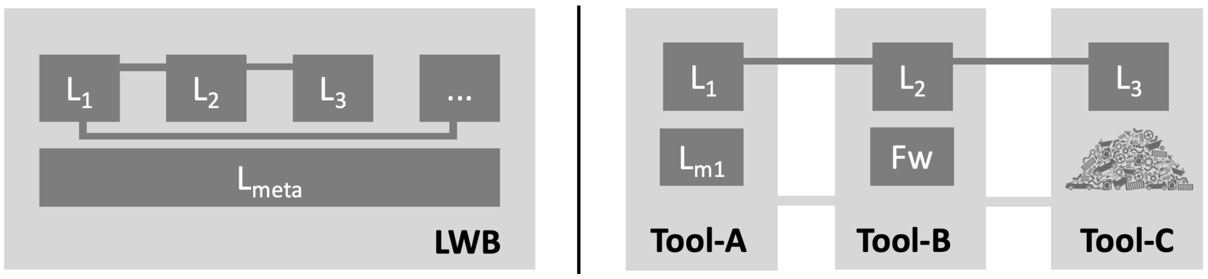
AN OPEN PLATFORM

Focusing initially on the Active Repository (instead of on the clients, notations and ways to define languages) has the advantage that different kinds of clients can be connected, including existing modeling tools: especially in the short term I expect existing tools, from Xtext and MPS over Simulink and Doors to be storing some of their data in the repository.

The set of services that works on the models is of course extensible. While services such as scopes or type checking should be part of the platform "out of the box", there will be a frameworks and APIs for developing new services; additional services can be contributed by the community or developed specifically for particular use cases. This approach facilitates the formation of a community around a common core.

INTEGRATABILITY

Tools like MPS are wonderful as long as you can do "everything" inside that single tool: to integrate languages, you only have to care about the semantic integration, there are no tool integration hassles. The integration between languages is seamless, semantically rich, and users get a consistent UX. On the other hand you have to fully buy into the tool and potentially re-develop some of the languages already available in other tools. And you lock yourself into the language workbench like MPS.



The other alternative is to integrate a suite of existing tools. In addition to semantic integration, you also have to "mechanically" integrate the tools. This is often a lot of work; in fact, often it

[4] https://en.wikipedia.org/wiki/Operational_transformation

is so much work to deal with the technicalities of the tools that the actual semantic integration is hardly attempted. The integration is shallower, there is not a consistent UX for the users. On the other hand, existing languages can be reused, you don't tie yourself to one tool, and this approach is of course much easier to do in distributed environments where everybody has their own most preferred (set of) tools.

Pragmatically speaking, you probably want the core set of languages used by a particular team in one tool (LWB-style integration) but also allow the more loose kind of integration for tools that are used less frequently. From an architectural perspective, the Active Repository proposed in this paper can accommodate both patterns:

A language and frontend tool that is custom made to work with the repository will fully rely on its services. It is a "dumb" editor, that renders models, accepts user interactions and displays analysis results, sending small deltas to the repository.

Existing tools can be integrated at two levels. The most basic one just sends completely edited models as one big "delta" to the repository for storage and to run some transformations. This does not exploit the active repository, but might be necessary pragmatically. A slightly richer integration will still use the tool for editing and the analyses that are built into the tool, but it might *also* display the results of other services the repository provides. An example is a state machine modeling tool: it might do all the "cheap" model processing locally (scopes, types), but then send the model to the repository where a model checker runs an expensive analysis; the tool then shows the results to the user.

MULTITENANCY

An important question is whether a single installation of the system should be able to run on a shared repository in the cloud, hosting models from different organizations—basically a public "modeling as a service" approach. This has lots of security implication in terms of isolating one party's data from all the others. The underlying concept is multi-tenancy support. I think we don't need this; many users will want to install their server-side components in a private cloud anyway. And even if the repository will be deployed to an actual Amazon or Google cloud, the usage scenarios probably allow for separate instances for different using organizations.

POTENTIAL COLLABORATORS

IncQuery Labs works on incremental queries and transformations and more specifically, on a scalable cloud-based version of their technology.

JetBrains works on a "Web-MPS" that focuses on client-side rendering and user interaction; such a repository would be synergetic.

itemis has lots of experience with these kinds of languages and their use and could help guide the evolution of the tool as well as the implementation. More specifically, itemis had a roughly similar project before, called Convecton. It was cancelled for internal reasons[5]. The concept of incremental transformations is implemented by Shadow Models:

http://voelter.de/data/pub/SLE2019.pdf

[5] I am happy to discuss the reasons in one-on-one conversations.

NASA JPL is already working on a repository that facilitates data exchange for existing COTS tools. Extending the idea into an Active Repository in the above sense would be straightforward.

I have had interest expressed in such a tool and project by various large companies including Siemens, Bosch, Alibaba, the Dutch Tax Agency and a few others.

RESEARCH VS. ENGINEERING

This is primarily an engineering project: for the core functionalities, no *basic* research is necessary. Instead, it is about adapting existing cloud technologies for model processing, or, when coming from the other perspective, scaling established language engineering approaches to a cloud-scale system. At the intersection of the two there is of course an opportunity for *applied* research: for example, how to efficiently replicate parts of the overall model to the client so it can be rendered and interacted with? How do you use existing incremental transformation techniques in a distributed database?

There are also many opportunities for research in the services. For example, how do you incrementalize model checking so that it can benefit of the incremental maintenance of its input model? However, these techniques can evolve over time and are not critical to the Active Repository itself.

NEXT STEPS

At this point I am sending this paper around to interested parties, collecting feedback and improving the paper along the way. When this phase is done, I will set up an informal structure to work on the requirements and make them one level more concrete. Ultimately, I will try to find a way of getting this developed. I see two alternatives of how this could be done:

– One option is to find a commercial company who will develop the technology commercially, backed by the concrete interest and (financial) support of other organizations who are interested in using the technology.
– The other option is that we all join forces (engineers, money, existing software components) and develop the infrastructure as open source software, perhaps in the context of an established open source organization.

In either case the architecture of the Active Repository allows for a rich ecosystem of commercial and open source service implementations to develop around it.

ABOUT THE AUTHOR

Markus Voelter is a recognized expert on modeling, DSLs and language engineering. Over the last 10 years he has mainly worked with Jetbrains MPS, arguably the most powerful tool for everything language-related; he has built or consulted on DSLs in a wide variety of technical and business domains (incl. most of the examples mentioned above) and was instrumental in the growth of the MPS team at itemis. The ideas in this paper are based on this experience with building DSLs and also with a (since cancelled) project at itemis of building a similar platform as a closed product. Find out more at http://voelter.de/publications/