Language Architecture: An Architecture Language for Model-Driven Engineering

Niels Brouwers, Marc Hamilton, Ivan Kurtev and Yaping Luo
Altran Netherlands B.V., Eindhoven, The Netherlands

Keywords: Architecture, Model-Driven Engineering, Domain-Specific Languages.

Abstract: The increasing number of languages used to engineer complex systems causes challenges to the development and maintenance processes of these languages. In this paper, we reflect on our experience in developing real life complex cyber-physical systems by using MDE techniques and DSLs. Firstly, we discuss a number of industrial challenges in the modeling software engineering domain. To address these challenges, we propose the concept of language architecture as an organizational principle for designing, reusing and maintaining DSLs and their infrastructure. Based on this, a metamodel for a DSL is designed and a tool support (LanArchi) is developed. Finally the possible future directions are given.

1 INTRODUCTION

Today's software systems are often implemented with multiple languages. Familiar examples are web-based systems that use XML and HTML for content, CSS for layout, and data manipulation languages for storing and querying data. While general-purpose languages are still the main implementation choice, a recent statement (Nierstrasz, 2016) clearly indicates that software engineers need specialized languages for requirements specification and problem domain modeling among others. The advances in the Model-Driven Engineering (MDE) make the development of Domain-Specific Languages (DSLs) easier. As the number of languages used to engineer complex systems increases, it becomes challenging to develop and maintain a set of inter-related languages and their dependence on existing hardware and software development infrastructure.

In this paper we reflect on our experience in developing real life complex cyber-physical systems by using MDE techniques and DSLs. After the development of dozens of DSLs and continuous requests for new ones, it is clear that the process of DSL development and maintenance needs a disciplined approach supported by tools at a level beyond the usual language ingredients. Languages are not 'isolated islands' anymore, they answer business and technical needs at a higher abstraction level, they are key players in the automation of the engineering process.

In our practice, an engineer (usually an expert in mechatronics or electronics but rarely in software development) specifies a solution in a DSL that reflects a certain problem domain. It is known that such a DSL requires a significant amount of additional functionality to allow the engineers to get confident in the quality of their solutions and to produce an executable solution. This functionality comes in the form of simulators, test generators, translators to formal verification tools, and finally code generators.

Some of the tasks to obtain these functions are currently well supported. Language workbenches focus on defining language abstract and concrete syntax, generation of editors and possibly specification of generators. Languages for specifying transformation chains, such as MTC Flow (Alvarez and Casallas, 2013), aim at automating the generation process. However, we clearly identify the need of considering language development at a higher abstraction level and in a broader scope. Software architects need to know how code generated from domain models fits in the global software architecture. System architects are interested in how company problem domain is covered by a set of related DSLs. Project managers make staff allocation and training decisions based on the DSLs that will be used as a development instrument. Finally, evolution of languages may impact the existing software infrastructure. In order to address these needs an explicit representation of languages and the relations to their environment are needed.

We propose the concept of *Language Architecture* as an organizational principle for designing, reusing and maintaining DSLs and their infrastructure. Lan-

Brouwers N., Hamilton M., Kurtev I. and Luo Y.
Language Architecture: An Architecture Language for Model-Driven Engineering.
DOI: 10.5220/0006206001470156
In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017)*, pages 147-156
ISBN: 978-989-758-210-3

guage architecture is a kind of megamodel (Bézivin et al., 2005). Megamodels are models whose elements denote other models. The initial idea of megamodeling is to represent the relations among models, metamodels, and transformations between models. The proposed concept of language architecture, as the name suggests, is closely related to the idea of *Linguistic Architecture* presented in (Favre et al., 2012). The goal of the *Linguistic Architecture* is to make explicit the linguistic structure of software systems: the involved languages, software artefacts expressed in them and their relations. The purpose of a language architecture is to support the design of new languages and their evolution at a more global level: to make explicit the role of the languages in the engineering process and the required tools. A language architecture is therefore an additional view with language-centric focus that is needed along with the well-known views used in system architecture specification. We do not limit the applicability of this concept only to the software engineering process but we apply it for any engineering process in which DSLs are relevant.

In order to support the practical application of language architectures, we developed a DSL based on a metamodel for expressing such architectures. This approach allows tool supported creation and analysis of language architectures.

This paper first describes the industrial challenges we have experienced and that motivate our proposal (Section 2). The main concepts in Language Architecture are explained and further structured in a metamodel (Section 3 and Section 4). The metamodel serves as a base for an initial tool support for the approach (Section 5). The paper concludes with positioning our work in the existing body of knowledge and pointing to future research directions (Section 6 and Section 7).

2 INDUSTRIAL CHALLENGES

Within a large industrial context multiple DSLs may be developed, each capturing a particular subset of the company's domain of interest, a single concern of the system being developed or a distinct activity in the systems engineering process. Each of these languages provide a viewpoint, likely at different levels of abstraction, to serve different stakeholders' needs.

Consequently, with the increased application of model-based engineering approaches and corresponding automation of the engineering process, there is a rapid growth of languages being developed in large industrial software companies. This leads to a number of challenges.

Lack of Language Centric View in the Engineering Process. DSLs and the associated generators are the key instruments for automating engineering processes. In an automated process the focus on artifacts that used to be manually created is shifted more towards the tools that perform the automation and artifact generation. DSLs and their infrastructures enter the area of interest of new stakeholders. Once a critical mass of languages is established, the evolution and maintenance becomes challenging tasks. Unfortunately, awareness of the role DSLs fulfill in the engineering processes is often not present or neglected. Furthermore, there is little support for establishing a language-centric view over architectures and processes. These factors ultimately hinder the stakeholders to access the vital information they need. We discuss refinements of this challenge in the next paragraphs.

Capturing a Domain in Multiple Languages. As indicated above, multiple DSLs play a significant role in the engineering process or to capture the domain of interest, each of them targeting a different set of stakeholders. How to capture a large and complex domain in smaller domains of interest by a set of stakeholders, while ensuring the domain is completely captured? How to ensure each of the stakeholders capture their domain of interest consistently with related domains? To address these questions, an overview on the level across individual languages is helpful.

Impact Analysis Upon Language Evolution. Like all software artifacts, languages evolve over time. The evolution of languages is a complex subject in the field of model-driven engineering and is known as the model/meta-model co-evolution problem (Mengerink et al., 2016). With the current state-of-the-art of modeling tools, the evolution of languages, especially when taking into account the impact on dependent components such as generators, editors and existing models, is a costly and error-prone process. As a result, it must be carefully planned, executed, and validated. This requires explicit representation of dependencies to the evolving language thus allowing proper impact analysis. Unfortunately, in many cases the inter-language dependencies are not explicitly described, which makes the impact analysis very hard.

Identification of Modeling Software Platforms. Automation of non-trivial steps in the engineering process requires the development of advanced multi-step transformations. A typical code generator to transform a problem-oriented DSL into a working software consists of several intermediate model levels in which the last model level is transformed into executable programs. When developing multiple transformations within a given industrial context, reuse of intermediate model levels becomes feasible to achieve cost reduction, higher quality and standardization. Such intermediate model levels with corresponding code generators become a standardized software modeling platform that will play a significant role to automate software construction. Identifying such opportunities for reuse is difficult when transformation architectures are considered internal and not publicly matched to the engineering process.

Tool Integration in the Engineering Process. In the past, software was engineered with only a very limited set of languages and tools. All aspects of a software system, ranging from data management to control behavior, from concurrency aspects to user interface design, were often implemented using a single programming language. Software engineering increasingly becomes a discipline where tools provided by different vendors need to be integrated into the engineering process, each addressing only a subset of the required aspects. Selecting the wrong tool results in sub-optimal designs and implementations. Increasing the amount of tools an engineer needs to interact with increases accidental complexity. Finally, once integrated in the engineering process, tools can hardly be replaced with a competitor tool without having to make large development costs. These three drawbacks of integrating different tools need to be addressed by an architectural level view on the engineering process. What are the requirements imposed on tools to be integrated and what are the goals they need to fulfill? How to integrate them in a flexible manner to prevent vendor lock-in? How to prevent accidental complexity? These are some of the standing questions related to this challenge.

3 MAIN CONCEPTS IN LANGUAGE ARCHITECTURE

With the language architecture, we address the design of automated engineering processes. We perceive an engineering process as a collaboration of design processes with the goal of constructing a system. Design processes aim at specifying or changing (for example, a refactoring of a software system) aspects of engineering units. Engineering units are systems or parts of systems. Generally, a design process uses a combination of decomposition and refinement activities. Refinement adds details to the specification of an engineering unit while retaining the original intention of this (more abstract) unit.

In this paper, the term ‘process contribution’ denotes the architectural concepts needed to describe such engineering processes. More concretely, these can be languages, artifacts, human or automated transformations, (sub)systems being constructed, tool environments, and engineers. In the light of the language-centric view on the engineering process, we are interested in the language aspects of any involved engineering process contribution.

Languages and their relations are the primary concepts in the language architecture. Any process contribution has to expose its relevant language aspects as an interface. For example, a model checking tool should explicitly state the formal language it supports, that is, its interface states that the tool ‘consumes’ specifications expressed in this language.

Figure 1 shows some of the main concepts in language architectures depicted as a simple mind map. In the following, we elaborate on these concepts.

Language: Definition and Application. In an engineering process humans often use natural languages to communicate with each other. Such languages are very complex and intuitive and are acceptable communication means at an abstract level. However, to automate parts of the engineering process, we need to restrict the used languages to formally defined processable languages. In addition to that, the construction relations between languages (such as composition (Clark et al., 2015) and extension (Voelter, 2014)) and aspects of realization technology also need to be expressed.

In order to deal with any language at an abstract level, we pose no assumption on how languages are defined. Furthermore, a *Language* can potentially be a group of languages. For instance, UML can be perceived as an integrated collection of languages. Engineers can express an interest in a subset, such as use case and class diagrams.

Our approach to representing languages largely follows the one taken by (Clark et al., 2015), though there are some significant differences due to our global interest in specifying languages:

Concrete syntax is not considered to be a defining part of the language. Concrete syntaxes can be attached as an optional aspect.

Abstract syntax needs a flexible interpretation: for our architecture we need a specification of the most significant language concepts. Optionally, more detailed language descriptions can be attached (OMG standards, metamodels, etc.).

Static semantics will be delegated to the referenced language descriptions.

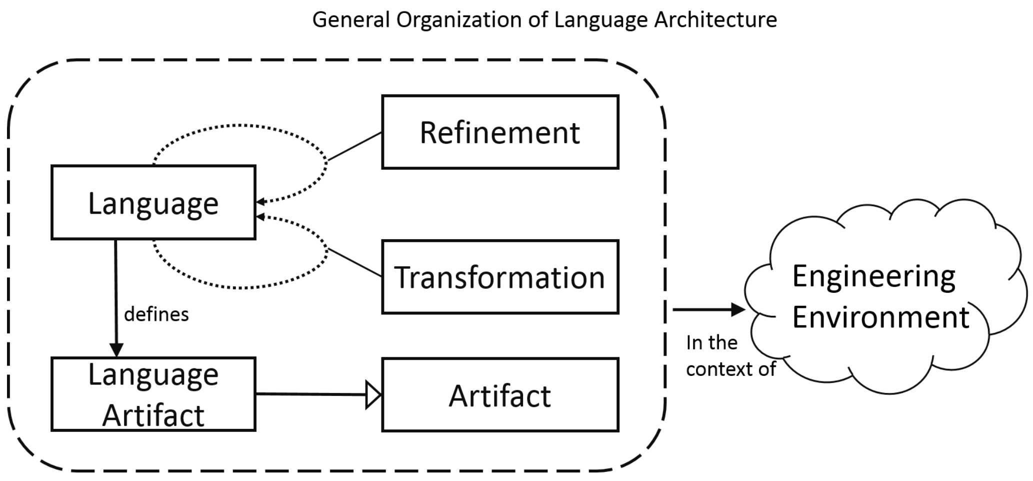

Figure 1: Main concepts in language architecture.

Meaning is dynamic semantics of languages which is enriched by a context. More details of this are explained in the following paragraphs.

When working with languages, the context where they are applied becomes relevant to the interpretation of these languages. In fact, many languages have a very limited priori semantics, which is enriched by the context. A sentence like "the component is optimized" could apply to software, but equally well to economics, electronics and other domains. In a specific context, this sentence may even be sufficient to derive optimized behaviors. Likewise, although a Uppaal specification (Larsen et al., 1997) has a formal semantics and can even be used to generate code, the result becomes useful and gets a meaning only in a given context. Even if some formal properties can be proven, without a relation to context, we don't know whether the specification refers to a washing machine or a missile launcher.

The application of a language in a context thus specializes the language by implicitly adding a mapping of language concepts to engineering concepts in this context. This is a subtle refinement of the language that is usually never explicitly given in a formal way. To capture the role of the context we introduce the concept of *Language Application* along with *Language Definition*. Language application refers to a language definition but is always present for a given context.

A language can have properties that can help in classifying its application. We select two key language properties: role and abstraction level. Example language abstraction levels are conceptual, logical, and physical. Languages play different roles in an engineering process. Example roles are:

'front-end' : languages optimized for the engineers' goals of developing systems;

intermediate : languages used throughout (automated) refinement to represent aspects of the development. Some intermediate languages are never used directly by engineers and may have no concrete syntax;

decoration : languages that can be used to drive the realization process, for example, to customize code generation;

formal languages : support formal analysis;

Refinement of Language Architecture Specifications. An engineering process can be specified at different levels of abstraction. Abstract specifications are expected to be refined by adding details to process contributions while respecting their original intention. We need to be able to express for every process contribution the goal(s) of its presence. As a first attempt we choose to define the goals in natural language.

Language Architecture must be capable of capturing the *Refinement* process of the involved languages. Imagine a process description in which we state that a specification is given in a state machine language without further restrictions. A refinement of the process can state that UML state machines are used. Another refinement dimension concerns language implementation technologies. At some abstract level, a language can be defined by the concepts available in a domain (a domain ontology). A technological refinement is to give a concrete language implementation using a grammar or a metamodel.

Artifacts. There are engineering contributions that are not languages but just *Artifacts* used in the engineering process. A typical example is a specification document. It may be a combination of models in different languages and natural text. When the artifact is directly defined by a language, we call it *Language Artifact*. Other more complex artifacts (e.g. specification document, contributing libraries, reference data, projects) can be defined as compositions of artifacts, some of them possibly being language artifacts. Following our goal to explicate every language aspect of engineering contributions, it should be possible to specify which languages are used in a composite artifact. It should be noted that in many cases there is no need to make the existence of artifacts explicit. For example, in a transformation chain where outputs relate to inputs it is not necessary to give concrete input artifacts.

Transformations. In design processes we observe decomposition and refinement. Both can be seen as a *Transformation*. The input and output of transformations are specified by languages. A transformation can be composed of other transformations arranged in a flow via intermediate languages. Again, we need to reason transparently about a transformation being potentially a group of transformations.

When we detail the properties of a transformation, a language mapping concept can be used. The goal here is not to exhaustively specify a transformation, but to use an elementary way of describing what the transformation should do, as an addition to the goals we can define on any engineering process contribution.

Concepts in the Engineering Environment. To relate the previously discussed process contributions to actual deliverables of the engineering process or to any other supporting tool we need to consider the context or the environment of the process.

The main outcome of a process is a system or a system component. As a motivating example for treating systems as process contributions, consider data obtained as a result of system calibration. The data can be fed back in the engineering process and therefore it is important to denote the system as a source.

Another important kind of engineering contributions consists of development tools. A model checker that consumes formal specifications in a given language is a tool that supports the engineering process and is a part of the process environment. Again, the interest is in the language aspects of the tools. We globally categorize process contributions into language processing contributions and non-processing contributions. They all expose languages as the relevant interfaces, but the first category can in addition expose language interfaces with a direction. A *Transformation* is a kind of language processing contribution. An *Artifact* is a kind of non-processing contribution.

Language processing contributions can be part of the result of the engineering process itself. A system that uses configuration language to customize its operations is an example of the result of a process that is a language processing contribution itself.

There are also language processing contributions that support the engineering process, such as build environments, test environments, etc. A kind of non-processing contribution other than *Artifact* is a storage environment like an archive.

General Organization of Language Architecture. Until now we identified specific process contributions and relations that we consider to be elements of language architectures. We also apply several well-known organizational principles when describing language architectures. All process contributions require a grouping or encapsulation concept with the option to selectively expose languages as interfaces. We propose a construct inspired by components in UML and in some architectural description languages. A component may encapsulate a potentially complex internal structure and if needed, can expose languages by using a port-like construct. This *Language Port* can be used to specify a language as an interface to the process contribution. A directed variant of a port can be used on processing components.

To connect elements, we need relations that have references to the languages at their ends. There can be, for example, a *Flow* relation from a language artifact to a transformation via a language that is specified as an input.

The issue of language compatibility arises when using flows that point to languages at their ends. Syntactic compatibility may be derived from construction relations, for instance, in case of a generalization, specialized languages will be compatible with their general language. Syntactic compatibility is, however, not always sufficient. Semantic compatibility has to be considered for the application context of the involved languages. The concept of model typing (Steel and Jézéquel, 2007) is also useful when resolving compatibility questions. Non-compatible connections imply a need for a transformation.

4 APPLYING LANGUAGE ARCHITECTURE IN PRACTICE

In this section we discuss aspects of applying the concept of language architecture in practice. First, the most important use cases are described based on our experience (Section 4.1). They serve as drivers for developing the tool support for language architectures. Currently available tool is an editor for specifying language architecture models. It is based on a metamodel explained in Section 4.2.

4.1 Use Cases of Language Architecture

We identify a number of uses cases for language architecture on the basis of the industrial challenges in Section 2.

Define Engineering Process. The architect can define the engineering process in terms of input/output language(s), transformations, and engineering components. Language architecture enables architects to refine or abstract parts of the architecture. Multiple refinements of an abstract entity can co-exist. A refinement tree can be derived for all components in the architecture. Refinement links support traceability of language architecture elements, which in turn is the enabling instrument for performing impact analysis and assessments.

Analyze Language Architecture. As the language architectures are captured in a processable form, they can be analyzed to find incompleteness, incompatibilities, redundancy, potential re-use, etc. Moreover impact analysis on the architectures can be carried out in case of the evolution of languages.

Validate language architecture The validity of language relations and refinements in a language architecture can be checked. If needed, transformations could be added or language choices may be reconsidered.

Extend Language Architecture. Depending on technology constraints and organizational choices, refinements can be created up to the level of physical language or transformation definition units (such as ecore, qvto, xml). In this case, the language architecture concepts can be re-used or extended, for example in the reference architecture definition (see Section 4.3). The rules/considerations/guidelines can be defined and associated with engineering process automation aspects.

4.2 Metamodel Design of Language Architecture

Based on the concepts defined in the language architecture description and the use cases, a metamodel of language architecture is created.

Figure **??**fig:lanarchiMM shows a snippet of the metamodel, where a number of main concepts are described. There are three types of components in a language architecture: *Artifact*, *ProcessingComponent*, and *LanguageDefinition*. As mentioned before, *Transformation*, *System*, and *Enviroment* are all types of processing components. A *Language* can be a *LanguageDefinition* or *LanguageApplication*. Moreover, languages can be exposed via ports: *LanguagePort* and *DirectedPort*. Finally, relations between languages (*LanguageRelation*) can be expressed having language references (ports, language definitions, language applications etc.) as relation ends.

4.3 Reference Architecture

Within a certain environment (e.g. a company), a number of choices are made concerning languages to be used, technological aspects and processes that are followed. These choices are often formulated as rules that guide the engineering process. Examples of rules relevant to language aspects are the definitions of abstraction levels, roles and choices of languages. Additionally, rules need to be defined concerning technologies to apply, guidelines for tools, interfaces between disciplines, connections to existing processes and others.

We call a set of such rules a *Reference Architecture*. The reference architecture provides the framework of choices and guidelines to refine language architectures in a specific context. From a practical perspective, we envisage that a reference architecture can be realized as a library of engineering contributions that are reused and refined in language architectures.

Figure 3 shows the relation between language architecture and reference architecture. Multiple language acrhitectures can exist in the framework of a single reference architecture. They can be extended with practical details such as the used build environments, deployment strategies and other. In this paper, we only focus on language architecture.

5 TOOL SUPPORT

In this section, the current implementation of an editor for language architectures (LanArchi Editor) is described with a small example.

5.1 An Example of Language Architecture for Model-based Testing

The complexity of languages and relations can grow rapidly even in a simple constellation, for example, a Model-based Testing (MBT) product, which allows generation of a test harness and a partial test suite based on the specification of a system. The goal of this example is to transform an ASD (G.H. Broadfoot,

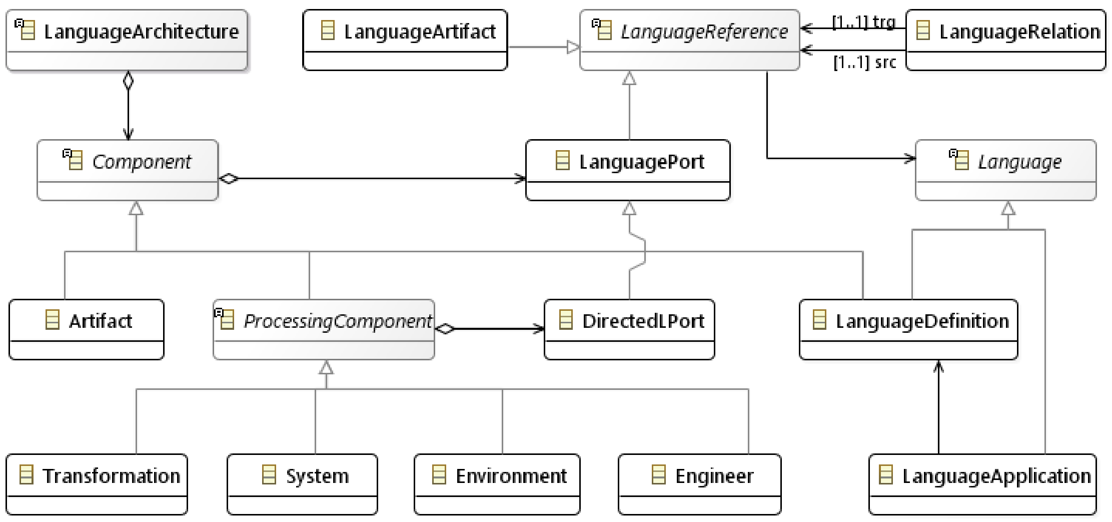

Figure 2: An extract of the metamodel of language architecture: only a number of main concepts are shown here.

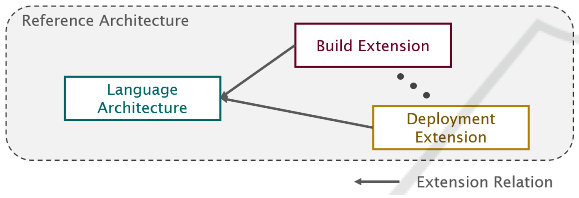

Figure 3: Language Architecture and Reference Architecture.

2005) model to a SpecExplorer (Veanes et al., 2008) test harness. ASD provides a language for specifying component interfaces and designs, and a tool that checks the models for presence of deadlocks and livelocks. SpecExplorer is a tool for model-based testing based on .Net.

In order to design a small engineering process that fulfills the example task, we take the ASD “language” with some additional test-related specifications as an input and the SpecExplorer “language” as an output from which the harness can be generated. In the previous statements the term language is in quotes because, as we will see shortly, ASD and SpecExplorer are technologies that expose more than one language. Figure 4 shows the main elements in this simple language architecture.

To be more specific (Figure 5), we can further refine the elements in this language architecture. Both input and output “languages” are sets of languages. We make a distinction between the languages that provide the concepts used by engineers to build models and the languages for serializing these models (referred to as Concepts and Storage respectively). To fill the gap between ASD models and SpecExplorer tests, a transformation needs to be built. ASD models are state machines that specify how an event is handled for every state. We call this style state-centric. SpecExplorer takes events as a leading concept specifying reactions for every event. We refer to this style as transition-centric. Therefore, in our example the required transformation obtains a transition-centric specification from a state-centric one. This task is generic and can be used for languages that adhere to the mentioned specification styles. We decided to built a generic reusable transformation from technology neutral state machine-like language (named System SM) to transition-centric specifications for specifying test suites. In this way we can deal with other pairs of input and output languages apart from ASD and SpecExplorer. Instead of ASD, we can use UML state machines or Excel state tables as inputs. Likewise, we can vary the test harness by transforming to conceptually similar but technologically different tool-sets. For instance, instead of SpecExplorer, Eclipse MBT can be used as output. Moreover, the additional test specifications are translated in parallel to SpecExplorer ‘cord’ scripts.

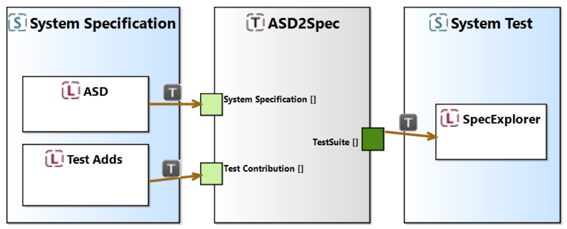

Figure 4: The abstract overview of our example: in this diagram the high-level architecture is shown. There is a transformation definition called *ASD2Spec*. This definition has two input ports (*System Specification* and *Test Contribution*) and one output port (*TestSuite*).

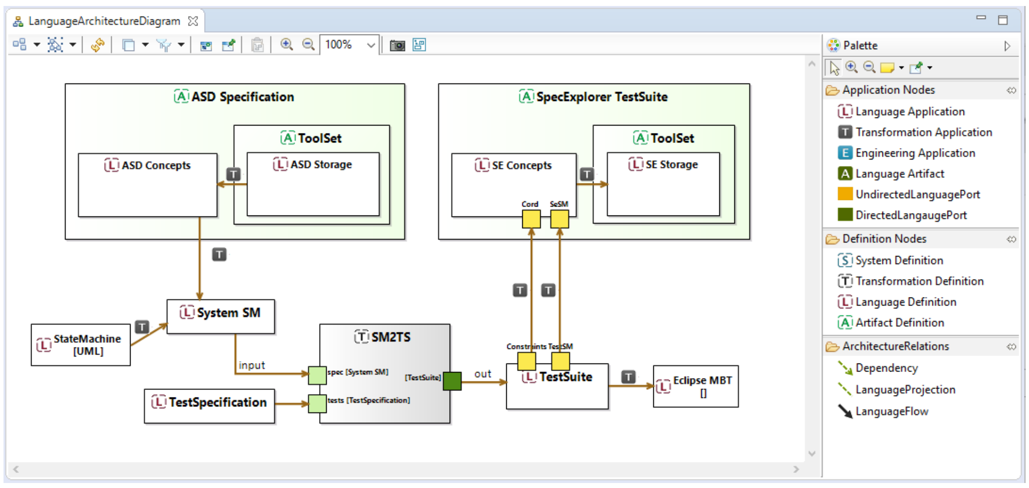

Figure 5: A screen-shot of the LanArchi editor: the left area is the drawing area where users can build their diagrams, the right area is the palette area where a number of notations are given.

5.2 The Language Architecture Models of the Example

With LanArchi tool, the architecture of our example can be constructed in different abstraction levels. Figure 4 shows an abstract architecture of the example. It provides a high-level view for users to understand the overall architecture.

To show the detailed information of the architecture model, a refined version can be created. Figure 5 shows a screen-shot of the model in the LanArchi editor. There are two definitions of artifacts: ASD Specification and SpecExplorer TestSuite. Each one has a tool set and concepts of the corresponding language. By using several chained transformations (shown as arrows or as boxes for a reified view), ASD Storage can eventually be transformed to SpecExplorer Storage. The figure also shows the intermediate languages (System SM and Test Suite) and the possibility to use different inputs (UML StateMachine) and outputs (Eclipse MBT).

6 FUTURE DIRECTION

The presented work on language architectures is ongoing and can be extended in several directions concerning the underlying concepts and the toolset.

Language Architecture Assessments. When an engineering process is expressed in a language architecture, it can be assessed if it satisfies certain criteria. Examples of criteria on engineering contributions are:

- Maturity level to identify the degree of automation or weak spots in the engineering process. Stakeholders might use this information for planning further process automation or to estimate project risks.
- Inventory of knowledge and expertise levels to plan personnel trainings.
- KPIs such as generated line of code (LOC) and number of clients as a means to evaluate business cases.

Language Architecture Patterns. In order to support language reuse and variability modeling, language architecture patterns can be developed to address quality attributes such as interoperability (prevent vendor lock-in), development costs, extensibility, scalability, maintainability.

7 RELATED WORK

As already mentioned, the concept of *Language Architecture* is a form of megamodel, i.e. a model where some model elements refer to other models. There exist several applications of megamodeling for different purposes. (Iovino et al., 2012) use megamodels to depict dependencies among models that need to be considered during model co-evolution. (Diskin et al., 2013) propose a graph-oriented view on megamodels and analyze the meaning of edges and vertices. The goal is to make explicit the meaning of relations among the models instead of treating them just as a graph edge. Linguistic architecture approach (Favre et al., 2012) aims at making the relations between software artefacts and their languages explicit. This approach also puts forward the idea that a linguistic architecture is a specific view that should be considered along with other views used in architecture descriptions.

Our approach shares commonalities with some of these works: language architectures should support in dealing with coupled evolution of models and indeed, we perceive them as views over existing software architecture, engineering process descriptions, and even enterprise architecture. Our driving concern for defining language architectures is to make an explicit representation of the role of languages in automating engineering processes. One consequence is that languages should be adequately described to serve the need of different stakeholders.

The need of considering a significant amount of inter-related languages, the support for language reuse, and presenting a language to different stakeholders are among the challenges identified in a process called Globalization of DSLs (Cheng et al., 2015) (Clark et al., 2015). (Clark et al., 2015) define globalization of DSLs as "purposeful construction, adaptation, coordination and integration of explicitly defined languages, to be amenable to mechanical and cognitive processing, with the goal of improving quality and reducing the cost of system development". In our practice we face these challenges in the scope of large companies and therefore our proposal contributes to solving some of them. Furthermore, several concepts found in the conceptual models in (Clark et al., 2015) are also present in our metamodel.

8 CONCLUSION

The growing application of DSLs in non-trivial industrial settings has a significant impact on engineering processes. Languages are the key enablers for engineering process automation. Management needs to understand the role of DSLs in relation to business goals and to take them into account when resources are allocated and people need to be trained. This calls for a language-centric view on system and process architecture descriptions. In this paper we proposed the concept of *Language Architecture* and a metamodel for a DSL in order to enable building such language centric views.

With the support of language architectures, language designers are aware of the main concerns that need to be addressed during the design process: the role of the language, the relations to its environment, the purpose of the language tool support. Promoting reuse of language components and fragments of transformation chains is also a very important aspect of our work. Furthermore, language architectures can assist in the strategic planning of process automation based on maturity assessment of process contributions.

Next steps for further development include professionalization of the tool and experimenting how it can support the reported industrial challenges. From theoretical perspective we plan to align with the work on DSL globalization in order to achieve better conceptual foundation for language architectures.

ACKNOWLEDGMENTS

We thank Mark van den Brand for his valuable comments on the paper.

REFERENCES

Alvarez, C. and Casallas, R. (2013). MTC Flow: A Tool to Design, Develop and Deploy Model Transformation Chains. In *Proceedings of the Workshop on ACadeMics Tooling with Eclipse*, ACME '13, pages 7:1–7:9, New York, NY, USA. ACM.

Bézivin, J., Jouault, F., Rosenthal, P., and Valduriez, P. (2005). Modeling in the large and modeling in the small. In *Model Driven Architecture*, pages 33–46. Springer.

Cheng, B. H. C., Degueule, T., Atkinson, C., Clarke, S., Frank, U., Mosterman, P. J., and Sztipanovits, J. (2015). *Motivating Use Cases for the Globalization of DSLs*, pages 21–42. Springer International Publishing, Cham.

Clark, T., Van Den Brand, M., Combemale, B., and Rumpe, B. (2015). Conceptual Model of the Globalization for Domain-Specific Languages. In Combemale, B., Cheng, B. H., France, R. B., Jézéquel, J.-M., and Rumpe, B., editors, *Globalizing Domain-Specific Languages*, volume 9400 of *Lecture Notes in Computer Science*, pages 7–20. Springer International Publishing.

Diskin, Z., Kokaly, S., and Maibaum, T. (2013). Mapping-Aware Megamodeling: Design Patterns and Laws. In *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*, pages 322–343.

Favre, J.-M., Lämmel, R., and Varanovich, A. (2012). Modeling the linguistic architecture of software products. In *International Conference on Model Driven Engineering Languages and Systems*, pages 151–167. Springer.

Iovino, L., Pierantonio, A., and Malavolta, I. (2012). On the impact significance of metamodel evolution in MDE. *Journal of Object Technology*, 11(3):3: 1–33.

Larsen, K. G., Pettersson, P., and Yi, W. (1997). UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152.

Mengerink, J., Schiffelers, R., Serebrenik, A., and van den Brand, M. (2016). DSL/Model Co-Evolution in Industrial EMF-Based MDSE Ecosystems. In *Workshop on Model Evolution at MoDELS 2016*.

Nierstrasz, O. (2016). The Death of Object-Oriented Programming. In *International Conference on Fundamental Approaches to Software Engineering*, pages 3–10. Springer.

G.H. Broadfoot (2005). ASD case notes: Costs and benefits of applying formal methods to industrial control software. In Fitzgerald, J., Hayes, I., and Tarlecki, A., editors, *FM 2005: Formal Methods*, LNCS, vol. 3582, pages 548–551. Springer, Heidelberg.

Steel, J. and Jézéquel, J.-M. (2007). On model typing. *Software & Systems Modeling*, 6(4):401–413.

Veanes, M., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., and Nachmanson, L. (2008). Model-based testing of object-oriented reactive systems with Spec Explorer. In *Formal methods and testing*, pages 39–76. Springer.

Voelter, M. (2014). *Generic tools, specific languages*. PhD thesis, TU Delft, Delft University of Technology.