Executable Architectures for Complex Software Systems

Sebastian Apel
Technische Hochschule Ingolstadt
Ingolstadt, Germany
Email: sebastian.apel@thi.de

Thomas M. Prinz
Course Evaluation Service
Friedrich Schiller University Jena
Jena, Germany
Email: thomas.prinz@uni-jena.de

***Abstract*—The design and implementation of complex software systems can be achieved by modern software architecture styles and well-chosen tool stacks. The resulting systems have their benefits in technical cleanliness, reproducibility, and automation (e. g., of processes). However, there is a gap between the design of the system architecture and its implementation. Well-advised architectures get lost in tool configurations and implementations of simple service-to-service communications that both do not belong to the scope of the architecture. How could this gap be closed without losing the advantages of tool stacks? This paper introduces the idea of focusing not on toolstacks, but on data models, data streams, algorithms, and business logic. Instead of designing architectures for documentation and overview, the architecture itself represents the executable system software. Our main idea is to describe the data model used in architectures and provide a language to describe the data's transformations.**

Keywords–Software Architectures; Distributed Systems; Development Tools; Model Transformation

I. Introduction

Architectures describe abstract components of complex software systems and how they communicate. They follow fundamental software engineering principles for independent and isolated development: high cohesion and low coupling [1].

Modern development approaches allow implementing an architecture closer to the components' descriptions to generate rudimentary applications (stubs), e. g., by using the *Google Web Toolkit* (GWT) [2][3], *swagger.io* [4], or *jHipster* [5][6]. Another recent trend for architectures is *microservices* [7]. This trend forces to create service-oriented components in isolation that are independent and resilient. The resulting service components represent functionality whose combination results in the complete and complex software system [8].

The usage of such modern development approaches requires different, individual tool stacks [9]. These tool stacks include middlewares, construction tools, container formats, process automation, and services deployment. This allows services and processes to be deployed in different runtime environments. The overall result should be a service-based modern software architecture.

Microservice systems claim to have advantages in technical cleanliness, reproducibility, reliability, and automation processes [9]. In reality, however, there is a gap between the design of the system architecture and its implementation, which usually leads to discrepancies between them. Architectures get lost in tool configurations and implementations of simple service-to-service communications. As a result, all application developers, from programmers to software architects, have to operate at multiple abstraction levels to implement the architecture, with much time not spent in application logic. A real-world project by Apel et al. [10] has shown that the implementation overhead ratio between functional and additional code is sometimes less than or equal to 1 : 3. In other words, for 100 lines of functional code, 300 lines of organizational code are needed. *It would be a gain in time, cost, robustness, and correctness if the developer can focus only on the application logic.* But how could this gap be closed without losing the benefits of different tool stacks?

This paper describes an idea to close this gap in Section II. Section III discusses this idea shortly compared to existing ones. A short conclusion ends the paper in Section IV.

II. Idea

Our idea includes four main aspects: (1) a meta-language that allows programming in different programming languages, (2) compilation into existing tool stacks, (3) automation of the appropriate tool stack selection, and (4) a suitable development platform.

The language (1) that allows implementation in different programming languages can be interpreted as a meta-programming or domain-specific language. However, it does not have to be a new one. One can assume an extension of Java or another popular programming language, such as is done in *ArchJava* [11].

The goal of the language is to reduce the effort of managing and configuring services and using different programming languages for them. 85% of software engineers within one study use multiple languages to solve problems during software development [9]. Instead of developing each service on its own, the language provides a common execution environment and abstracts from their communication and deployment. One advantage is that common data models can be implemented centrally and used in all components. It also avoids tedious mapping of parameters. The disadvantage is that it seems somewhat centralized, where the advantage of an independent service implementation can increase its generalization and minimize its coupling. When designing such a language, this fact must be carefully considered.

Since there is a trend towards data streams and data science, the language should enable data orientation. In addition to defining data structures and functional programming, it should also allow processes that connect different data

 ISBN: 978-1-61208-844-0

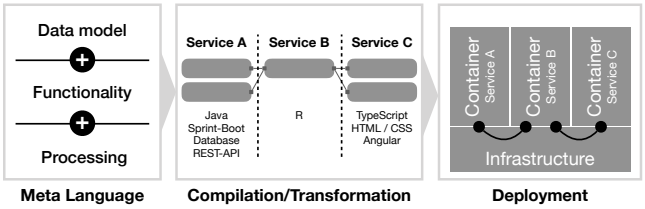

Figure 1. Linkage between meta language, compilation / transformation, and deployment.

streams.

Architectures described in the meta-programming language should be executable in an ad-hoc fashion as shown in Fig. 1. We propose to focus on both interpretation and compilation (2). Language interpretation has the advantage of fast error detection, debugging, and bottlenecks identification. Compilation should increase performance, especially if it distributes the various services across different (virtual) execution environments.

Traditional compilation translates a software system into one set of (virtual) system instructions. However, our idea is to compile the language into instructions in those programming languages that best fit the functionality's realization — in case the developer does not want to choose this and describes the functionality abstractly in the meta-language. In other words, since programming languages belong to different tool stacks, the compiler must translate the language into an individual set of tool stacks. Data structures, functions, and processing chains described at an abstract level would then have to be translated into multiple programming languages.

The compiled components and tool stacks must communicate with each other. A surrounding execution environment should enable this communication. Furthermore, compilation remains within the problem complexity of the architecture description. As in the case of the *Unified Modeling Language* (UML) [12], our goal is not to find a language that covers every use case by default. The language should provide bounds, and every part of it should be executable. It is not the idea to cover all existing development approaches. On the contrary, the focus is on questioning some daily development practices and searching for alternative approaches.

One difficulty with our idea is identifying those parts/ functionality of the language that will be compiled into the same components. Another difficulty is choosing an appropriate tool stack for these components (3). An automatic decision must interpret essential information provided in the language – such as functions and component-specific data. Another difficulty is the inclusion of existing dependencies (other systems, libraries, and services) and how their integration works within and between services.

Our idea follows well-established computer science principles in the problem description, compilation, and execution. As with ordinary programming languages, a development tool (4) should support development with the meta-language and with all phases of software development (planning, analysis, design, implementation, and maintenance). Since one of our primary goals is to reduce technical details, the tool should focus on problem-solving, i. e., describing and programming the software, rather than on the particular technical configurations.

This development environment is our final goal. It allows focusing only on architecture and business logic. Different components should be described in separate projects and supplemented by dependencies. However, since every project knows and uses the same meta-language, the development environment can provide support across programming language and tool boundaries. The application should be immediately interpretable in the environment. When projects are deployed, the application is compiled and made executable in the form of services with service-specific tool stacks. For example, the results of the compilation could be containers (such as Docker [13]) that are published. The compilation realizes the communication described in the architecture, and the service publication assures availability.

III. Short Discussion

Of course, our idea is not completely new and there are several, other approaches in the literature. For this reason, we compare our idea in the following with two approaches that have the same research direction but a different focus. The first of them is *ArchJava* by Aldrich et al. [11]. ArchJava is a *Java* extension that provides three new language constructs: *Components*, *Ports*, and *Connectors*. Components describe architectural components with their ports, i. e., what communication endpoints are needed and provided. Some components can be connected via connectors, which then results in a concrete software instance consisting of multiple components. ArchJava is very promising, but Aldrich et al. state as limitations that it is language-bound to Java and only runs on a single Java Virtual Machine (JVM). However, our goal is to be free of these limitations. In addition, ArchJava operates on a high architectural level and method calls are only allowed within components. Although this is understandable for an architectural view, in some cases developers would benefit from a low-level method call where (technical) architecture details are hidden from the developer. In particular, tool stack decisions are sometimes unnecessary or disruptive when high performance is not a concern. In summary, however, ArchJava offers good and clear concepts and its focus on implementation and language constructs should be strongly considered when implementing our idea.

A second approach to architecture-level design and implementation is Archface by Ubayashi et al. [14]. Archface is very high level in architecture decisions and is oriented towards UML. Like ArchJava, it provides the language constructs *component* and *connector* and a new one *architecture*. Components and connectors are special, abstract interfaces. Component interfaces describe the communication endpoints of the component, while connectors describe their interaction. Architectures finally define concrete implementations of the components and connectors and, therefore, define a concrete software instance. Archface is also promising, but seems to have the same limitations as ArchJava. Although

 ISBN: 978-1-61208-844-0

it can be implemented for different programming languages — like ArchJava —, it is unclear how different components communicate across languages. However, in summary, the ideas of Archface are valuable during implementation outlined by our idea.

IV. Conclusion

Our idea is a new executable meta-language that should support the complete application development lifecycle. This language covers data structures, functionality, and descriptions of processes. It enables that each part of the described application is compiled into a best fitting tool stack. The selection of service boundaries and tool stacks is done automatically. In addition, it automatically implements how the components (services) of the architecture communicate. The result is a correctly configured distributed application based on existing technologies. The developer benefits by always acting on the level of the architecture description to realize the application. To support the developer in using the language, one goal is to provide a development platform. This platform covers the development along the complete software development process. Starting with planning and analysis, the support is possible up to implementation and maintenance.

References

[1] L. Bass, P. Clements, and R. Kazman, Software Architecture in Practice, Third, Ed. Addison-Wesley, 2013.

[2] A. Tacy, R. Hanson, J. Essington, and A. Tokke, GWT in Action, 2nd ed. Greenwich, CT, USA: Manning Publications Co., Feb. 2013.

[3] Google, "[GWT]," [retrieved: March, 2021]. [Online]. Available: http://gwtproject.org/

[4] Swagger, "The best apis are built with swagger tools," [retrieved: March, 2021]. [Online]. Available: https://www.swagger.io/

[5] M. Raible, The JHipster mini-book, 5th ed. USA: C4Media, 2018.

[6] JHipster, "JHipster - Generate your Spring Boot + Angular/React applications!" [retrieved: March, 2021]. [Online]. Available: https://jhipster.tech/

[7] M. Viggiato, R. Terra, H. Rocha, M. T. Valente, and E. Figueiredo, "Microservices in practice: A survey study," arXiv preprint arXiv:1808.04836, 2018.

[8] F. De Paoli, "Challanges in services research: A software architecture perspective," in Advances in Service-Oriented and Cloud Computing, A. Lazovik and S. Schulte, Eds. Cham: Springer International Publishing, 2018, pp. 219–227.

[9] H. Zhang, S. Li, Z. Jia, C. Zhong, and C. Zhang, "Microservice architecture in reality: An industrial inquiry," in IEEE International Conference on Software Architecture, ICSA 2019, Hamburg, Germany, March 25-29, 2019. IEEE, 2019, pp. 51–60. [Online]. Available: https://doi.org/10.1109/ICSA.2019.00014

[10] S. Apel, F. Hertrampf, and S. Späthe, "Towards a Metrics-Based Software Quality Rating for a Microservice Architecture - Case Study for a Measurement and Processing Infrastructure," in Innovations for Community Services - 19th International Conference, I4CS 2019, Wolfsburg, Germany, June 24-26, 2019, Proceedings, ser. Communications in Computer and Information Science, K. Lüke, G. Eichler, C. Erfurth, and G. Fahrnberger, Eds., vol. 1041. Springer, 2019, pp. 205–220. [Online]. Available: https://doi.org/10.1007/978-3-030-22482-0_15

[11] J. Aldrich, C. Chambers, and D. Notkin, "Archjava: connecting software architecture to implementation," in Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA, W. Tracz, M. Young, and J. Magee, Eds. ACM, 2002, pp. 187–197. [Online]. Available: https://doi.org/10.1145/581339.581365

[12] Object Management Group, OMG Unified Modeling Language — Version 2.5.1, Object Management Group Std., Dec. 2017, [retrieved: March, 2021]. [Online]. Available: https://www.omg.org/spec/UML/2.5.1

[13] Docker Inc., "Empowering App Development for Developers — Docker," [retrieved: March, 2021]. [Online]. Available: https://www.docker.com/

[14] N. Ubayashi, J. Nomura, and T. Tamai, "Archface: a contract place where architectural design and code meet together," in Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010, J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, Eds. ACM, 2010, pp. 75–84. [Online]. Available: https://doi.org/10.1145/1806799.1806815

 ISBN: 978-1-61208-844-0