Model-Integrating Microservices: A Vision Paper

Mahdi Derakhshanmanesh
University of Koblenz-Landau
Institute for Software Engineering
manesh@uni-koblenz.de

Marvin Grieger
University of Paderborn
Department of Computer Science
marvin.grieger@uni-paderborn.de

Abstract: Model-integrating development is a novel approach that aims to provide a comprehensive conceptual framework for the engineering of flexible software systems. The atomic building blocks for architecting model-integrating software are model-integrating components which support the modular cooperation of flexible models and efficient code at runtime. Model-integrating components achieve flexibility by using models at runtime and operations on them like querying, transforming and interpreting. Microservices achieve flexibility by upgrading whole components at runtime. In this short paper, we sketch the vision of Model-Integrating Microservices (MIMs) that combine the strengths of model-integrating components with microservices to support continuous software engineering. With this early work, we intent to initiate a fruitful discussion about architectural design considerations in the community.

1 Background and Motivation

Model-Integrating Development (MID) [Der15] is a novel approach that aims to provide a comprehensive conceptual framework for the design and development of flexible software that can be adapted easily and – in parts – even autonomously [ST09]. Moreover, being able to make changes quickly and to evolve software easily also supports longevity [GRG^{+}15]. We observe that these capabilities are similarly useful in the emerging trend of Continuous Software Engineering (CSE).

The building blocks for MID are *Model-Integrating Components* (MoCos) which we have presented and implemented in previous research [DEIE14]. MoCos result from the symbiosis of *Component-Based Development* (CBD) [SGM02] and *Model-Driven Development* (MDD) [BCW12] concepts. A MoCo is a non-redundant, reusable and executable combination of logically related models and code in an integrated form where both parts are stored together in one component.

In traditional MDD, models (e.g., UML activity models, feature models, component models) are used at design time. In contrast, in MID models are used *directly* at runtime, too. As a result, the internals of MoCos can be easily *monitored* and *analyzed* using model queries, as well as systematically *modified* using repeatable model transformations. The same techniques can be used for evolving components using an editor during development and maintenance, and adapting them at runtime using an administration panel or an autonomic adaptation manager component.

By combining the strengths of modeling languages (e.g., abstraction, separation of concerns) and programming languages (e.g., performance) within components, the MoCo concept yields flexible and well-performing software.

Being able to evolve software easily and quickly is also a main goal of the current trend of *microservices* [New15]. Microservices support a fine-grained approach to the modular implementation of distributed, flexible, fault-tolerant and highly responsive software systems. In contrast to the MoCo approach, which concentrates on the technical means for adaptability of component internals at runtime, the microservice approach concentrates on upgradeability of individual, reasonable-sized software components as a whole.

To summarize, we observe that our ongoing work on MoCos has the potential to fulfill goals similar or complementary to microservices. In order to understand the opportunities and challenges, we sketch an initial vision of a novel *software architecture concept* that aims to combine the strengths of MoCos with the capabilities of microservices. This concept is introduced subsequently and shall serve as an initial baseline for discussion.

2 Architecture Concept

The architecture concept of *Model-Integrating Microservices* (MIMs) builds on top of the established MoCo concept. An MIM is a MoCo that is realized with microservice technology. Therefore, an MIM adds a couple of additional benefits to the MoCo concept.

The benefits of MoCos over other component concepts are: (i) *enhanced flexibility* because the software system and its individual components can be observed using model queries, can be modified by adapting models using an editor or model transformations and can be executed using model interpreters, (ii) *support of separation of concerns* because each model targets a concern, (iii) *understandability and maintainability* because models are assumed to be easier to understand and easier to handle than code, (iv) *self-documentation* because a well designed modeling language is assumed to be a documentation and (v) *no synchronization problem* because there is no redundancy between model and code within a MoCo unless it is introduced willfully, e.g., to realize reflection.

The expected added value of MIMs over the MoCo concept are: (i) *self-containment* because a MIM is deployed together with its full infrastructure and dependencies, (ii) *distribution* because MIMs communicate over a network and (iii) *decentralization w.r.t. data* because each MIM manages its own data; especially including its models.

An example of two MIMs is depicted schematically in Figure 1. A description from an *external point of view* and an *internal point of view* is given subsequently.

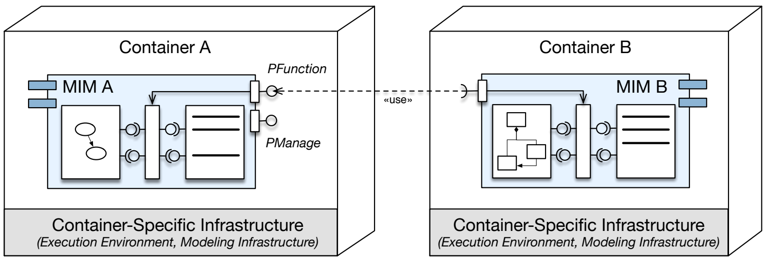
Container A
PFunction
MIM A
PManage
«use»
Container B
MIM B
Container-Specific Infrastructure
(Execution Environment, Modeling Infrastructure)
Container-Specific Infrastructure
(Execution Environment, Modeling Infrastructure)

Figure 1: High-Level Sketch of two Connected Model-Integrating Microservices

2.1 External View

Externally, each MIM runs in its own process and is deployed in a standardized *container*[1] that can be executed on any (virtual) server node. In the given example, there is one in `Container A` (`MIM A`) and one in `Container B` (`MIM B`).

Encapsulation supports full compatibility with other microservices as MIMs can be used in a black-box manner. Conceptually, and in line with the original MoCo concept, each MIM offers its *application functionality* via a group of interfaces at the `PFunction` port. When using these interfaces, consumers do not notice any difference to other microservices. Optionally, an MIM can offer invasive *management functionality* – such as transformations on its encapsulated models – via secured interfaces at the `PManage` port. A similar splitting between a functional interface and a control interface is discussed in related work [Kra09].

To realize MIMs, the *Container-Specific Infrastructure* comprises an *Execution Environment* and also a *Modeling Infrastructure*. In the tradition of microservices, each MIM comes with all required dependencies and with the infrastructure for handling models at runtime (e.g., model query and transformation engines, model interpreters).

2.2 Internal View

Internally, MIMs stress the microservice idea of *technology diversification* where each team can choose the tools and languages that suit their skills and goals best.

On the one hand, to leverage existing code and the ability of skilled people to develop software in a well-known programming language such as Java, each MIM can have a *code portion*. This is sketched by the right side of each MIM in Figure 1.

On the other hand, to leverage the powerful capabilities of existing modeling languages and technologies, each MIM can have a *model portion*. This is sketched by the left side of each MIM in Figure 1. Any kind of modeling language can be (re)used for this purpose. For instance, textual, visual and hybrid *Domain-Specific Modeling Languages* (DSMLs) can express application logic and data structures.

[1] *Docker containers* are a contemporary example (`https://www.docker.com/what-docker`).

Importantly, in contrast to MDD, no code is generated for these designed models. Models such as process descriptions and can be executed by model interpreters that traverse the model's abstract syntax graph representation at runtime. In this regard, DSMLs serve a similar purpose like interpreted scripting languages.

Models are integrated systematically with code inside of an MIM. This approach assumes that, from a technical point of view, code objects and model objects can both be handled equally, i.e., they can be referenced and their behavior can be invoked by calling methods of a facade. Objects from the code portion and the model portion can be connected in a hard-coded manner but there are flexible alternatives, too. For example, the *mediator pattern* [GHJV95] can be followed as illustrated in Figure 1.

3 Impact and Synergy Effects

Based on experience from our established and ongoing research on MoCos, we are convinced that using microservices to realize the MoCo concept in the form of MIMs will enable the engineering of even more flexible software systems. This is due to the fact that the MoCo concept and the microservice concept address complementary concerns to achieve dynamism and to support continuous software engineering.

MoCos focus primarily on the *internals of components*, i.e., on the use of domain-specific and general-purpose modeling languages and modeling capabilities such as querying, transforming and interpreting for the systematic analysis and adaptation of parts of components during design and at runtime. Microservices focus primarily on the *outside of components* and their containers, i.e., on component size and boundaries, aspects of distribution, updatability and rapid (re)deployment.

Next, we describe an excerpt of *opportunities* and *challenges* related to the MIM concept.

3.1 Opportunities

In terms of *opportunities*, the MIM concept brings a couple of advantages over traditional microservices and vice-versa.

Most notably, the MIM approach brings the whole *world of modeling* and especially *models at runtime* to the microservice world. This enables the use of various kinds of general-purpose and domain-specific modeling languages, thus supporting all advantages of modeling like abstraction and separation of concerns via different views. Central to MIMs is the added flexibility of using models at runtime, so changes to component internals can be made systematically without swapping the whole component.

To illustrate this benefit in the context of CSE, take for example the timeline depicted in Figure 2. In this example, the evolution of an MIM over time is shown. The merits of the MoCo concept allow to perform manual or automatic *micro-adaptations* by transforming the integrated models. Such changes are lightweight but limited in scope. In contrast, the

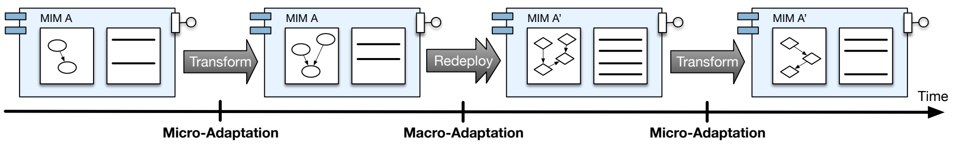

Figure 2: Example Timeline of an Evolving Model-Integrating Microservice

microservice concept enables to perform heavyweight *macro adaptations* by upgrading (i.e., modifying and redeploying) a component as a whole. In a nutshell, a MIM-based software system can be evolved on various levels of granularity so software engineers can choose the most appropriate modification technique per context.

Moreover, realizing the existing MoCo concept using microservices, related technology and best practices, the more general MoCo concept can be optimized for one specific kind of technological space and community. Thereby, the original MoCo concept also benefits from the MIM vision, because microservices and available container technologies support realizing dynamically evolving software from an *infrastructure perspective*. This point has been a weak point in MoCos, so far.

3.2 Challenges

In terms of *challenges*, the MIM concept brings a couple of open issues to be tackled.

For instance, the MIM concept strongly requires the rapid and smooth development of DSMLs. However, current meta-tools are not quite there, yet. Moreover, the *fast (re)deployment of the modeling infrastructure* together with MIMs needs to be supported.

Obviously, the introduction of modeling comes with a technological overhead. Despite offering this approach to handling complexity, microservices already come with their own technological and organizational overhead. Therefore, adding the extra costs for the design and use of DSMLs needs to be considered with care. Regarding fundamental conceptual issues, we discussed challenges for the required *modeling infrastructure* such as (i) modularization and integration of metamodels, (ii) links between distributed models, (iii) specification of model semantics as well as (iv) data and control flow between models and code in earlier work on MoCos [DEG15].

Further challenges are inherited from the nature of traditional microservices. For example, *eventual consistency* must be managed because data may exist redundantly (e.g., across models in different MIMs that are distributed across a network) and *fault tolerance* must be supported because MIMs can become unavailable (e.g., during model processing).

4 Concluding Remarks

We believe that the symbiosis of architectural concepts from the worlds of MoCos on the one hand, and microservices on the other hand, opens doors for interesting opportunities in continuous software engineering. With MIMs, software engineers can benefit from the flexibility of modeling languages across the full software lifecycle including runtime. By presenting this early work, we hope to initiate a discussion on architectural design considerations. Moreover, we plan to realize an MIM variant of the MoCo infrastructure [Der15] as a technical basis for carrying out additional feasibility studies.

Acknowledgements. This work is supported by the Deutsche Forschungsgemeinschaft (DFG) under grants EB 119/11-1 and EN 184/6-1. We thank Gregor Engels for pointing us at microservices and we thank Jürgen Ebert for his feedback on the text.

References

[BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool, 2012.

[DEG15] Mahdi Derakhshanmanesh, Jürgen Ebert, and Marvin Grieger. Challenges for Model-Integrating Components. In *Proceedings of the 2nd International Workshop on Model-Driven Engineering for Component-Based Software Systems co-located with 18th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2015), Ottawa, Kanada, September 28, 201*, 2015.

[DEIE14] Mahdi Derakhshanmanesh, Jürgen Ebert, Thomas Iguchi, and Gregor Engels. Model-Integrating Software Components. In Juergen Dingel and Wolfram Schulte, editors, *Model Driven Engineering Languages and Systems, 17th International Conference, MODELS 2014, Valencia, Spain, September 28 - October 3, 2014*, Valencia, Spain, 2014. Springer.

[Der15] Mahdi Derakhshanmanesh. *Model-Integrating Software Components - Engineering Flexible Software Systems*. Springer, 2015.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GRG$^+$15] Ursula Goltz, Ralf H. Reussner, Michael Goedicke, Wilhelm Hasselbring, Lukas Märtin, and Birgit Vogel-Heuser. Design for future: managed software evolution. *Computer Science - Research and Development*, 30(3-4):321–331, 2015.

[Kra09] Sacha Krakowiak. Component Control. In *Middleware Architecture with Patterns and Frameworks (Distributed under a Creative Commons license)*, chapter 7.4.5. 2009.

[New15] Sam Newmann. *Building Microservices*. O'Reilly and Associates, 2015.

[SGM02] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 2002.

[ST09] Mazeiar Salehie and Ladan Tahvildari. Self-Adaptive Software: Landscape and Research Challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, 2009.