Facilitating Development and Provisioning of Service Topologies through Domain-Specific Languages

Ta'id Holmes
Products & Innovation, Deutsche Telekom AG
Darmstadt, Germany
t.holmes@telekom.de

***Abstract*—In a model-driven engineering (MDE) context, the coordination of different roles such as enterprise architects and developers can be supported when dependencies between roles and artifacts are stated. Similarly, provisioning and deployment of service topologies can be facilitated. For specifying dependencies, an editor permits to define roles, artifacts, services, and service topologies in descriptive domain-specific languages (DSLs). Supporting coordination and automation, utilities are generated that synchronize workspaces, produce notifications, prepare the provisioning of service topologies, and perform their deployment. For showcasing the DSL editors and the coordination and automation tools a case study from a machine-to-machine context is taken. Addressing change impact and provisioning issues by minimizing turnaround cycles, the demonstration reveals possibilities of how to support MDE processes in the context of service topologies and shall foster a discussion on the potentials with regard to enterprise applications in general.**

***Keywords*-automation, coordination, development, DSL, MDE, provisioning, service topology**

I. Introduction

The engineering of enterprise applications involves various stakeholders such as architects (e.g., business and enterprise architects), developers (e.g., service and web engineers), and testers. Model-driven engineering (MDE) permits to incorporate different roles in the engineering process, each generally working with a distinct set of artifacts. For example, a textual or graphical domain-specific language (DSL) can offer an access for the specification of aspects of an enterprise application. This is because the DSL's corresponding metamodel inherently has a defined level of abstraction, making it easy for a respective role to relate to. Avoiding media discontinuity through model transformations, model-based approaches help to master complexity and proved to be efficient and viable.

Because model transformations are automated in MDE, regeneration is cheap and can take place continuously. Thus, MDE approaches permit iterative development to take place. In case of a service topology, i.e., a set of interdependent services specified and developed by a multitude of roles, the challenge of minimizing turnaround cycles re-emerges. This is because development needs to be coordinated and deployment of software as part of the service topology is complex. While supporting the various roles and the simultaneous development of an enterprise application in form of a service topology, provisioning thus needs to be automated as well.

In fact, the coordination of the various roles during development, their deliverables, and the application deployment is not covered by the MDE methodology and needs to be addressed. For example, a model change undertaken by an enterprise architect may have an impact for certain developers working with applied transformations. That is, the change needs to be propagated and communicated. This is costly in terms of time and creates an overhead to the engineering project. Ideally, all roles participating in the engineering process would be timely notified and provided with models or appropriate transformations. In this regard, it must also be considered that different developers may work with different technologies. That is, a code generator needs to be selected according to the respective target technology.

To exemplify, consider a service engineer who utilizes XML technologies, while another engineer working on a different part of the service topology accesses a database using object-relational mapping (ORM) software. Finally, a web developer processes data in JavaScript Object Notation (JSON). While all of the roles may work with particular views, an information model would comprise all of the concepts in required detail. As software code (e.g., an XML schema and an ORM layer) is generated from the model and provided for the roles, such a model is entangled with the systems it reflects. Within the scope of the model, this permits governance over the service topology and enforces conformance to the enterprise architecture.

For facilitating the coordinated development and automated deployment of service topologies, descriptive DSLs are presented (see Section III) for declaring artifacts, roles, services, and their dependencies. Without imposing rigid processes, the description of an engineering project as expressed in the DSLs (referred to as DSL programs) serves as a basis for generating coordination and automation tools such as for synchronizing workspaces, producing notifications, and provisioning of services. The demonstrator thus utilizes a DSL editor and showcases generated coordination and automation tools using a case study from a machine-to-machine context.

The remainder of this paper is structured as follows: Section II presents a motivating example and mentions related problems. The approach is presented in Section III. Next, Section IV revisits the case study by illustrating the applicability of the approach followed by a discussion in Section V. Finally, Section VI concludes.

II. Motivating Scenario

For illustrating a motivating example, let us consider a machine-to-machine (M2M) context in which a proof of concept (PoC) is to be developed as part of a service topology. In this scenario, it suffices to develop and demonstrate the PoC using a rudimentary setup. M2M devices submit data from sensors to an M2M gateway which relays data to a backend. There, a PoC implementation correlates the sensor data with user data and data from other sources and stores results in a database. Finally, a dashboard provides a reporting interface. Various roles are involved in realizing the scenario. In particular, an enterprise architect produces an information model describing the sensor data. A service developer, implementing the PoC, processes the data using XML technologies and uses an ORM layer for persisting results in a database. Another service developer implements representational state transfer (REST) services that offer results in JSON to web clients. Finally, a web developer accesses these services with a cross platform web application, realizing a dashboard.

Part of the development is model-driven. Notably, the information model constitutes the basis for various conforming instances used by different roles. As a consequence, multiple stakeholders exist for the information model. While the model is technology agnostic, code generators transform the information model to specific technologies. In this example, the model is transformed to an XML schema from which object-oriented Python code is generated. Using SQLAlchemy as an ORM layer, additional code for the mapping is generated from the information model.

The following challenges can be observed for this scenario: If the model changes, depending code needs to be regenerated and respective roles need to be informed. Consecutively, depending services need to be redeployed. The resulting coordination needs, transformation steps, and deployment tasks protract turnaround cycles. Besides the involved cost, this is problematic when striving for continuous and agile development. The described change impact may even impede the proper adoption of MDE in practice and lead to situations in which model-driven transformation and generation steps are not performed iteratively. That is, generated software evolves without synchronizing the models diverging enterprise architecture and development. Thus, undermining MDE approaches and their governance, initially constituting models no longer truthfuly reflect the systems.

III. Approach

For addressing the problems mentioned, i.e., to minimize turnaround cycles, the coordination during development and the provisioning needs to be facilitated. This can be achieved and automated if dependencies within engineering processes are known. For formalizing these, a model-based approach using DSLs is proposed.

Roles, artifacts, services, and service topologies are described in DSLs (see Sections III-A and III-C). Conforming DSL programs are parsed and their models are transformed to coordination and provisioning tools (see Sections III-B and III-D). The former category comprises tools for synchronizing workspaces and for producing notifications for informing roles. For the provisioning of the service topology, a script is generated that automates the packaging, uploading, and deployment of the services.

For management purposes and for realizing these use cases in a service-oriented fashion, an MDE server (cf. [1], [2]) is deployed. It comprises a repository (cf. [3], [4], [5], [6]) for managing MDE artifacts (i.e., metamodels and models) and services for realizing MDE activities (i.e., model transformations). First, a DSL program of a project for a service topology is registered at the MDE server. Invoking a generated service, the workspaces are then initialized in a version control system (VCS). In addition, tools are deployed that interact with the MDE server and notify roles depending on artifacts that are contained in the respective changeset. Finally, for the provisioning of the service topology the workspaces are processed for preparing the provisioning and services are packaged, uploaded, and deployed.

A. *Describing Roles, Artifacts, and Services*

A grammar for an external DSL has been defined using the Eclipse Xtext (Xtext) framework. A generated DSL editor – offering syntax highlighting, code completion, and validation – permits to describe the various roles, artifacts, and services in a textual manner. An `artifact` represents one or more files. A `file` as well as an artifact may specify a (meta-)artifact, e.g., for indicating a type or format. A `role` states what artifacts it `consumes` and `produces`. A `service` may depend on (`dependsOn`) some of these artifacts. In addition it `consumes` and/or `produces` runtime data. Roles and services may specify a transformation format or serialization for referenced artifacts. This leverages the fact that model transformations exist for models (or DSL programs) conforming to metamodels (or DSLs). Relating to the runtime it can be specified that a certain serialization is used (e.g., XML or JSON).

B. *Workspace Synchronization*

By stating which artifacts are produced or consumed by roles and services, the descriptive DSL program implicitly defines dependencies that require synchronization. These dependencies can directly be translated into publish / subscribe patterns. That is, publishers are roles producing artifacts and subscribers are roles that consume artifacts. Presuming all roles work with a VCS for managing development artifacts, i.e., for committing or checking out artifacts, respective publish / subscribe patterns can directly be integrated into the VCS. At this level *post-commit hooks* permit to realize programmatic notifications. Thus, a shell script is generated and deployed as a hook in a service topologies' repository that analyzes a changeset for subscribed artifacts. For all transformation formats that are specified by subscribers, required model transformations are applied by invoking the respective services from the MDE server. Resulting artifacts are provided to the roles by placing them into their workspaces. Finally, subscribers are notified by mail if enabled.

C. Service Topologies – A High-Level Description

The services defined in the previous DSL (see Section III-A) constitute elements of the targeted service topologies. For describing a service topology at a high level of abstraction, another DSL has been defined. Using Xtext's `import` statement, it makes use of language referencing (cf. [7, p. 119]) for realizing separation of concerns and for making definitions reusable across service topologies. In this DSL, a `hostingUnit` comprises `services`. After code generation, it is mapped to a server instance or a cluster, using an infrastructure as a service (IaaS) provider. The services which are grouped in `hostingUnits` may reference previously defined services (e.g., software as a service (SaaS) as part of an enterprise application) or relate to general services (e.g., platform as a service (PaaS)). Services may depend on other services (`implies`) forming a service stack. For example, an SaaS may build on a PaaS. At an IaaS level, `ports` can be specified for services they listen to. For not exposing them publicly, they can be quantified as `internal`.

While these aspects are covered by the DSL and respective provisioning steps are realized in code generation, the DSLs do not contain sufficient detail for automating the overall provisioning. To fill the gap between the high-level description and the low-level, technical configuration management, the approach integrates with Puppet as explained next.

D. Service Provisioning

Similarly to the dependencies between `roles` as leveraged for the synchronization of workspaces, also the dependencies between `hostingUnits`, `services`, and their dependent `artifacts` can be used for easing the provisioning of the service topology. For this, an artifact a service `dependsOn` is retrieved from the workspace of the role that `produces` the artifact and is then placed in a Puppet module. A `hostingUnit` is transformed to a Puppet node definition and includes Puppet modules for services. For this, the transitive closure of the services contained in a `hostingUnit` is calculated. For services that are not quantified as `internal`, enabling security rules are generated for respective ports.

By executing a generated script that utilizes IaaS clients the IaaS of the service topology is provisioned. A management server, acting as the Puppet master, distributes and deploys the configuration and software to clients. For this, cloud-init directives for configuring the Puppet client are passed as user-data when launching server instances.

When changes occur in the workspace, a script can be triggered that pushes incremental changes to the IaaS. This minimizes turnaround cycles and facilitates iterative development and continuous delivery.

IV. Revisiting the Case Study

A DSL program describing the roles, artifacts, services, and their dependencies of the scenario from Section II was created. Once registered at a MDE server, workspaces for the roles were initialized in Git as a VCS. After an enterprise architect created an information model describing sensor data, resulting artifacts were distributed. In the DSL program it was specified that the first service engineer required the information model in form of an XML schema (**`consumes`** `InformationModel` *`as XSD`*) while another expected an ORM definition based on the same information model. Thus, XML schemas and an ORM definition were placed in the workspaces of the respective developers which were notified by mail.

Once the missing development artifacts had been implemented and were available in the VCS, a deployment process could be triggered. That is, the entire service topology from IaaS to SaaS was provisioned through generated IaaS clients and available Puppet modules. The latter were partially generated, i.e., their contained artifacts were synchronized with the workspaces. The service topology comprising the PoC was provisioned within a matter of minutes. For this an IaaS provider based on OpenStack was utilized.

When the information model changed, the workspaces were updated using required transformations. A continuous integration server could inform developers of incompatibilities and the need to make subsequent updates. Finally, iterative development was facilitated by a generated utility that redeploys the service topology by considering the differential changes.

V. Discussion & Related Work

The approach presented is generic and neither limited to the case study nor its context. Relating to roles, artifacts, and services, it is applicable to any MDE project. A presumption, however, is the use of a common VCS among all roles and the existence of service-oriented model-transformations as realized by an MDE server.

The fact that the process of providing developers with required artifacts is automated supports the proper use of MDE in complex projects such as the development of enterprise applications or service topologies. From a developer point of view, there is a need to communicate with enterprise architects when a change relating to a model is desired. This procedure ensures governance over the system and through the generated synchronization tools strengthens collaboration between the different stakeholders.

The approach presumes that there is a rather clear idea what roles participate in the engineering process, what artifacts they produce and consume, and which artifacts are required for assembling the services. At the beginning of a project, this may not always be the case, however. While the act of formalizing these aspects can be beneficial for a project, the final set from an implemented service topology may only be known a posteriori. Thus, a model that describes a service topology, its services, artifacts, and roles involved in the engineering project needs to be updated regularly. This can be done manually by a project administrator or directly by developers.

One possibility to support and (partially) automate the maintenance of the model is to analyze the VCS as elaborated by Sarma et al. [8] (cf. also Cataldo and Herbsleb [9]). For this, the results can be taken as a reference for the artifacts that are produced by a role. That is, the fact that files exist in a *development area* (not containing provided artifacts) of a roles' workspace implicitly denotes that they are produced by the respective role. Following this idea, the explicit model from a DSL program can be made implicit when backed by appropriate synchronization tools. In this regard and for supporting an inferred model related research on coordination needs as found in Blincoe [10] has to be incorporated. In contrast to the literature, one distinction of the presented work is the focus on MDE with its singularity regarding model transformations.

Regarding provisioning, the OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) standard (cf. [11]) addresses the description of service topologies for deployment and migration scenarios. Without further tool support it requires experts to realize the desired automation. In contrast, the high-level DSL (see Section III-C) abstracts from many details and is tailored toward end-users. While it currently integrates with Puppet after model transformation it may be used to generate TOSCA making TOSCA and related technologies accessible.

VI. Conclusion

Whereas MDE embraces multiple roles, the methodology does not cover coordination issues, e.g., resulting from a change impact. Based on DSLs, a model-based approach has been presented that exploits specified dependencies between roles, artifacts, services, and service topologies. Utilities are generated and employed for automating synchronization of workspaces and provisioning of entire service topologies. Thus, an MDE process for enterprise applications can be facilitated by describing respective roles, artifacts, and services.

Acknowledgments

The author would like to thank Robert Schwegler for fruitful discussions, reviewers for providing valuable comments, and Mike Machado for proofreading.

References

[1] X. Blanc, M.-P. Gervais, and P. Sriplakich, "Model bus: Towards the interoperability of modelling tools," in *MDAFA*, ser. Lecture Notes in Computer Science, U. Aßmann, M. Aksit, and A. Rensink, Eds., vol. 3599. Springer, 2004, pp. 17–32.

[2] T. Holmes, U. Zdun, and S. Dustdar, "MORSE: A Model-Aware Service Environment," in *Proceedings of the 4th IEEE Asia-Pacific Services Computing Conference (APSCC)*, M. Kirchberg, P. C. K. Hung, B. Carminati, C.-H. Chi, R. Kanagasabai, E. D. Valle, K.-C. Lan, and L.-J. Chen, Eds. IEEE, Dec. 2009, pp. 470–477.

[3] K. Altmanninger, G. Kappel, A. Kusel, W. Retschitzegger, W. Schwinger, M. Seidl, and M. Wimmer, "AMOR – towards adaptable model versioning," in *1st International Workshop on Model Co-Evolution and Consistency Management, in conjunction with MODELS '08*, 2008.

[4] E. Stepper et al., "Connected Data Objects (CDO) model repository," The Eclipse Foundation, 2005, [accessed in June 2014]. [Online]. Available: http://eclipse.org/cdo

[5] M. Kögel and J. Helming, "EMFStore: a model repository for EMF models," in *ICSE (2)*, J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, Eds. ACM, 2010, pp. 307–308.

[6] L. G. P. Murta, H. L. R. Oliveira, C. R. Dantas, L. G. Lopes, and C. M. L. Werner, "Odyssey-SCM: An integrated software configuration management infrastructure for UML models," *Sci. Comput. Program.*, vol. 65, no. 3, pp. 249–274, 2007.

[7] M. Völter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, and G. Wachsmuth, *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[8] A. Sarma, Z. Noroozi, and A. van der Hoek, "Palantír: Raising awareness among configuration management workspaces," in *ICSE*, L. A. Clarke, L. Dillon, and W. F. Tichy, Eds. IEEE Computer Society, 2003, pp. 444–454.

[9] M. Cataldo and J. D. Herbsleb, "Coordination breakdowns and their impact on development productivity and software failures," *IEEE Trans. Software Eng.*, vol. 39, no. 3, pp. 343–360, 2013.

[10] K. Blincoe, "Timely detection of coordination requirements to support collaboration among software developers," in *ICSE*, M. Glinz, G. C. Murphy, and M. Pezzè, Eds. IEEE, 2012, pp. 1601–1603.

[11] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann, "TOSCA: Portable Automated Deployment and Management of Cloud Applications," in *Advanced Web Services*, A. Bouguettaya, Q. Z. Sheng, and F. Daniel, Eds. Springer, 2014, pp. 527–549.