CHAPTER

12 ARCHITECTING CLOUD SERVICES FOR THE DIGITAL ME IN A PRIVACY-AWARE ENVIRONMENT

Robert Eikermann, Markus Look, Alexander Roth, Berhard Rumpe, Andreas Wortmann
Software Engineering, RWTH Aachen University, Aachen, Germany

12.1 INTRODUCTION

Interconnectivity and ubiquitous computing are key components in the forthcoming age of digitalization. To support this, and to flexibly keep up with newly arising requirements, software is continuously moving from comprehensive desktop applications to smaller services that interconnect with other services and exploit user data to provide added-value experiences. This shift raises concerns for service users and service developers. For service developers, such a change in design and architecture requires engineering of scalable systems that can be developed and maintained independently. This is reflected in the microservices paradigm currently used within many software projects. In microservice architectures, the individual services contribute small parts of domain functionality. The backbone of such microservice architectures is built up from independently functioning base services providing different platform functionality. Such architectures supply millions of users with services. To scale, cloud computing techniques are used providing the necessary scalability and elasticity. The resulting architectures can compose different services to added-value services based on large sets of user data. For service users, employing added-value services requires provision of personal data to many different, yet partially interconnected services. Each of these may store copies of the data and share this with the services they use. This requires the user to abandon control of their data as well as consistently changing their data throughout various services.

In this chapter we discuss a system-of-systems architecture that centers services around the user as opposed to provider-driven services. For creating such architecture, we support the developer with a model-driven and generative methodology supporting reuse of existing services, automated conversion between different data models, and integration of ecosystems facilitating service composition, user data access control, and user data management. To this end, we motivate the need for flexibly composable cloud services by example (Section 12.2), before we identify challenges for such architectures (Section 12.3). Afterwards, we introduce preliminaries (Section 12.4), present our conceptual building blocks for a system-of-systems architecture (Section 12.5), and explain how code generation can facilitate development of composed added-value services (Section 12.6). The subsequent sections discuss related work (Section 12.7) and our approach (Section 12.8). Finally, we conclude this contribution (Section 12.9).

[ELR+17] R. Eikermann, M. Look, A. Roth, B. Rumpe, A. Wortmann:
Architecting Cloud Services for the Digital me in a Privacy-Aware Environment.
In: Software Architecture for Big Data and the Cloud, chapter 12, pages 207-226. Elsevier, June 2017.
www.se-rwth.de/publications/

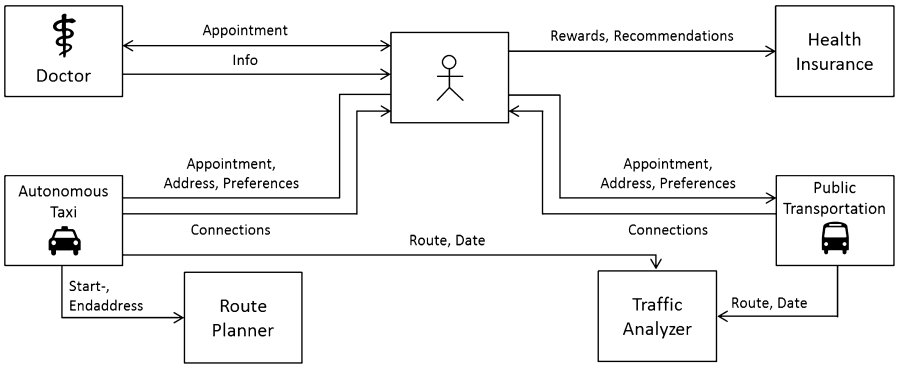

FIGURE 12.1

Overview of available services in our scenario directly or indirectly available through apps on the person's smartphone

12.2 EXAMPLE

This section gives an example scenario to demonstrate the supported forms of use cases. The overall scenario is a person who wants to simplify everyday life processes in order to improve work-life balance. To achieve this, the person already uses several specialized apps on her smartphone. Fig. 12.1 shows systems participating in this scenario. All actions are initiated by the person using her smartphone. In general the person is in good health but likes to participate in several preventive checkups that are recommended by her health insurance. As these preventive checkups often have irregular or long time intervals, her health insurance has a monolithic app where recommendations for checkups are conveniently published. This app uses the Health Insurance Service denoted in Fig. 12.1. She regularly uses this app to inform herself about upcoming preventive checkups. If an examination by her doctor is indicated by the health insurance app, she negotiates an appointment with the doctor. The doctor also has a standalone app to make appointments online. The corresponding service is the Doctor Service in Fig. 12.1. Additionally to the preventive checkup service, the health insurance company also offers a bonus program to reward health supporting behavior of its customers. Our person lives in a larger city and therefore she needs to plan carefully how to get to the doctor's office just in time for her appointment. She can choose from different available mobility offerings performed by the local public transportation company, the new car sharing company that implements a taxi service with autonomous cars or just use her own car. Depending on time and personal preferences she favors one service over the others. As denoted in Fig. 12.1 each, the autonomous taxi and the public transportation company, offer a dedicated service, which is available through a standalone app, to let customers check availability and book trips. Unlike buses, autonomous taxies don't need to stick to a predefined route, and the autonomous taxi internally uses a routing planer for the optimal driveway. The Route Planner has an integrated map annotated with additional information like special streets only available for certain types of vehicles. To overcome high exhaust fumes' concentrations in big cities, bus lanes may also become available for autonomous taxies, or streets may be closed for vehicles exceeding a certain size.

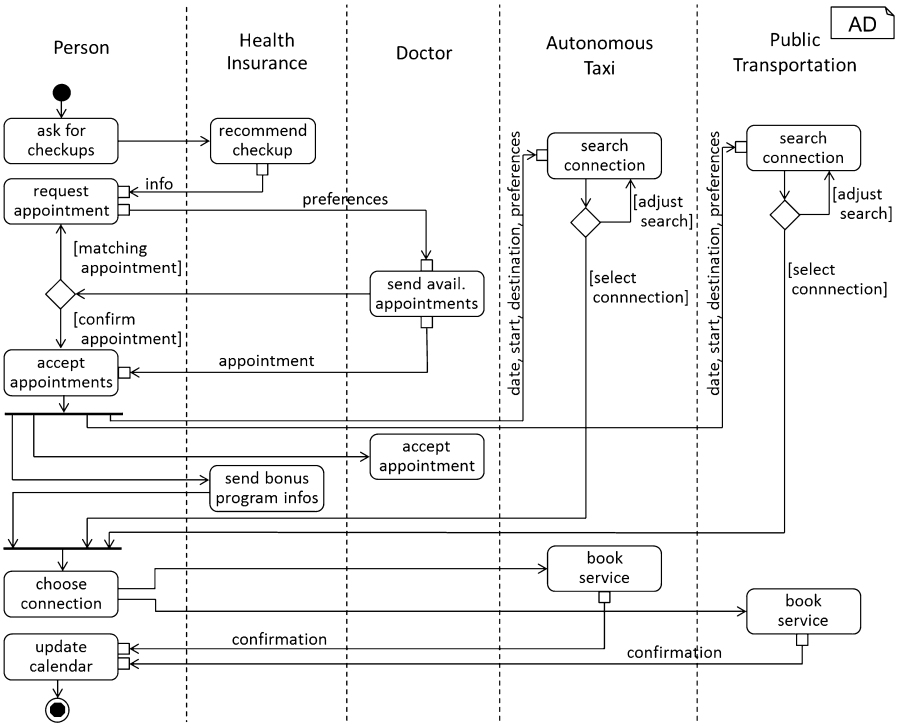

FIGURE 12.2

Activity diagram showing all steps our person has to manually to use different services

Including this, route planning becomes much more dynamic due to high flexibility of available streets. The Traffic Analyzer internally uses several data sources to calculate utilization of all streets.

This data is then extended to a utilization forecast. As a result, the Traffic Analyzer Service is able to provide detailed information about expected time adjustments for routes. In combination with the Route Planner, the Traffic Analyzer Service can be used for a better route selection and estimate the required time for a route. These two services are used in the background by the mobility apps, and our person doesn't come directly in touch with them.

Several apps available on the smartphone, bringing vital services to the users, have to be used separately. Common data must be manually shared between apps, and personal preferences are only collected specifically for each app's use cases. The person from the previously described scenario uses apps where available. In everyday life a combination of different apps and services is often desired. The composition is done manually using the smartphone. Different apps are used in specific order, and data, like personal data or results from previous service calls, is transferred from one app to another. Necessary steps in terms of data transformation and service composition are shown in Fig. 12.2 and described in the following paragraph. Each swimlane in the activity diagram in Fig. 12.2 corresponds to the respective service from Fig. 12.1.

Upcoming preventive checkup recommendations are received through the health insurance app. The person must read the information and decide if this checkup is applicable to her. If this is the case, the app connected to her doctor is used to start an appointment negotiation. In order to be able to use this app, she had to give her personal contact information to the app provider. Even worse she must put a list of her doctors at the app's disposal. This allows her to contact the doctors via the app.

To start an appointment negotiation, she must manually transfer the information from the health insurance app, check general availability in her calendar, and send the appointment request. Attached to that, she has to transfer her personal (timing) preferences into the appointment request because the app only supports appointment preferences in terms of weekdays or availability in the morning, midday or afternoon. The doctor receives this request for an appointment and can check which open slots he can offer. Our person checks if a slot fits well into her time schedule and confirms that slot if applicable. Otherwise she answers with slightly adjusted preferences maybe aiming at appointments a bit more in the future. This is repeated until a suitable slot is found and confirmed. If a suitable slot for an appointment is found, the doctor sends the appointment details to her. The appointment is then displayed in the app and the time information has to be transferred into a calendar item in her calendar app. In doing so, three subsequent steps are initialized. First, the doctor receives a confirmation which appointment was ultimately chosen, which may trigger necessary preparations for the doctor's staff. Second, the health insurance app is used to gather detailed information about available vouchers in the bonus program. Third, our person has to look how to reach the doctor's office. Several providers, like local public transportation or an autonomous taxi service, are present, and each one has a dedicated app available to inform about service details. As our person wants to compare service offerings, she has to use each app but this must be done separately. The apps need information like date and time, home- and doctor's address, and personal preferences. They are very similar for each app but have to be manually entered again. Each app presents a list of mobility offers matching her request. By manually switching between the apps, she can choose the offer most suited for her. As the health insurance app already indicated that the upcoming examination is covered by the bonus program, she does not only need to compare different offers of the mobility services, or consider which offers are eligible for the voucher. As the last option she could also user her own car. Finally, she has to book the chosen offer which includes that she has to enter her personal payment information and submit it to the service provider behind the app. Again, her personal data is spread into another system, and no integrated mechanisms allow her to keep track which data she entered where.

As shown and described in Fig. 12.2, apps on smartphones are already supporting everyday life. However, many apps lack interconnections, so that identical information must be manually transferred from app to app.

12.3 CHALLENGES

Our notion of services follows the W3C, for which "[a] service is an abstract resource that represents a capability of performing tasks that form a coherent functionality from the point of view of providers entities and requesters entities" (https://www.w3.org/TR/ws-gloss/). This entails that a service fulfills a single (coherent) functionality, it is used by requesters, and it may use different providers. Requesters and providers may be persons or other services (i.e., services are meant to be composable). Addition-

ally, services yield a description detailing the interfaces and the semantics of the service. The interfaces define the possible interaction in terms of messages or data types consumed and produced by the service whereas the semantics of the service describes the functionality of the service and the effect of using the service. A single service is a standalone system that relies on other services to fulfill its functionality. This allows for detached development of such a service.

12.3.1 SERVICE COMPOSITION

Research and engineering in services have produced a multitude of different, hardly composable services, which require different representations of the same data types or yield incompatible interfaces [11,24]. Nonetheless, cloud-computing has brought forth a plethora of services that are in use and contribute to large cloud-based architectures. Millions of users use the added-value services of such architectures by providing their personal data to the services. The development of such added-value services, for instance, composing various mobility providers to a comprehensive mobility service is usually done for specific purposes and requires tremendous handcrafting. This is error-prone and ultimately hinders services reuse. Development, integration, and usage of reusable, privacy-aware services yield many challenges to *service developers*, *providers*, and *users*. The challenges for services providers are mainly socio-economical and hence not considered in this article.

12.3.2 TECHNOLOGY ABSTRACTION

Engineering complex services usually reuses existing technologies to avoid reinvention solutions for cross-cutting concerns, such as data distribution, persistency management, or load balancing [22,59, 40,18]. Service developers can choose various technologies to reuse from, but must comprehend and apply these concerns properly. For instance, selection of a proper data store based on the properties of collected data requires expertise in database technologies. Thus, a service developer engineering a service for a specific domain must also become an expert in cross-cutting issues. Similarly, when and how to replicate services to reduce resource consumption when the services are not demanded and to scale-up when demand is high requires proper expertise.

12.3.3 SERVICE AND DATA INTEGRATION

Service developers have the burden of freedom to select from various mechanisms to describe the interfaces of services [21]. Whichever notion is selected may hinder future composition and reuse with other services. Bridging the gap to connect services employing different interface notions requires careful handcrafting. Adapting between, for instance, time-synchronous and asynchronous services requires expertise in both interface kinds. Similarly, the adaptation of data types requested and provided by different services requires further handcrafting. For example, connecting a service requiring a person's place of residence to a service providing the person's address requires correct interpretation of both sets of fields and proper (de-)composition to enable usage of the latter service. This becomes even more complicated if various representations for the same information are available (for instance, different date formats).

12.3.4 TRUSTED USE OF PERSONAL DATA

Added-value services usually exploit (large amounts) user data to provide better user experience [53, 14,32]. However, this usually requires participating users to hand over control of their data (such as medical data or mobility preferences) to the services and subsequently controlling how the data is processed is impossible, revoking it is hardly transparent, and changing it requires to perform updates with each services that was granted a copy of the data (for instance, at user registration). For instance, if our person in the example scenario needs to change her mobility preferences, she currently would require updating this data for all mobility providers she used before. Restoring the users' control over their data and its usage is crucial for trusted added-value services of the future. Providing this data and access management in a centralized, accessible fashion reduces the redundancy of multiple data updates with each service. While the latter may be subject of the services' ecosystem, the individual services must be developed in a fashion to rely on centralized, ecosystem-provided user data only. This also ensures that services can rely on the most up-to-date data easily.

Overall challenges are that service users must be enabled to dynamically compose, specifically tailored to their needs, different monolithic services. To foster this, they have to be able to control data flow, access to data, and consistent change of their data. This includes looking up who accesses which data for which reasons, adding and revoking permissions for specific data and services, and propagate updates to all related services.

Thus, service developers of cloud-based user-centric services operating on big data have to be enabled to:

- Develop services for reuse in different contexts.
- Specify data transformation between services with different interfaces.
- Choose technology depending on the nature and properties of necessary data.
- Retrieve personal data transparently from a Digital Me without collecting superfluous identity information.

12.4 PRELIMINARIES

Large distributed systems, such as the Internet-of-Things, consist of heterogeneous, interacting, software modules that capture various domain expertise. Engineers developing services for such systems must cope with the complexity of integrating solutions formulated by respective domain experts in different general programming languages (GPLs). Integration of these solutions requires comprehension of multiple GPLs, their libraries and communication mechanisms, although these express similar concepts (such as the data types to exchange or the interfaces of services). This lack of conceptual abstraction ultimately increases engineering efforts [18].

Model-driven engineering (MDE) [56,24] is a software development paradigm that lifts abstract models to primary development artifacts. Such models can be domain-specific, focus on important concepts, and avoid the notational noise [58] of GPLs. To this effect, possible models for specific concerns are characterized by domain-specific languages (DSLs), which, for instance, capture concepts of data structures (UML class diagrams [41]), software architectures [35,36], or system behavior [1, 54]. In MDE, models are used for communication, documentation, analysis, and ultimately execution. For the latter, models are either interpreted at run-time [18,2] or translated into GPL artifacts using code

generators embodying software engineering expertise [10]. As such models further can be independent of GPL details, they are available for translation into different GPLs and enable us to reuse domain expertise with different programming languages, ecosystems, and services [46]. Multiple domains, including automotive [2,26], avionics [17], robotics [6,45] have employed MDE with various success [34,57].

A prominent kind of DSLs for the description of hierarchical software architectures are architecture description languages (ADLs) [35,34]. One of these languages is the MontiArc [29] component & connector (C&C) ADL, which provides concepts for atomic components and composed components, component interfaces of typed, directed ports, and connectors between these interfaces to architecture developers. Components are black boxes that compute behavior: Atomic components either encapsulate models of other DSLs to describe their behavior or are linked to GPL artifacts [47]. Composed components contain a hierarchy of subcomponents and their behavior emerges from the interaction of their subcomponents. These subcomponents can be atomic or composed again, allowing for architectures of arbitrary complexity with direct access to specific functionality where required. The components exchange messages via their typed ports, which rely on UML/P class diagrams [48] to describe the messages' structures. The encapsulation of components and the flexibility of class diagrams enable for a detached and agile [49] development of components and facilitate a post-deployment integration. With these, service developers can engineer services as components that provide and require services via their stable interfaces of typed ports reuse other services via hierarchical composition. Consequently, cloud-based distributed systems have adopted ADLs for description of services in form of software components as well [23,11,40]. These ADLs feature modeling concepts specific to cloud-based architectures, such as replicating components of message contexts for stateless systems. However, important challenges, such as data migration and privacy-control have yet to be addressed in such systems-of-systems.

Fig. 12.3 depicts a C&C architecture that describes a public transportation service at its top and the related data structures at its bottom. The service enables users to calculate transportation plans based on starting location and destination and considers the user's personal preferences to optimize route calculation. The added-value service Public-TransportationService is composed from three other services. Two of these are specific to the public transportation service, the third (TrafficAnalyzerService) is incorporated from another service provider. This form of tight coupling via integration hides the public transport service's dependency from the traffic prediction service encapsulation ensures it appears as a single, reusable service to potential users. The service also interacts with a banking service and a persistency service (both supplied by different service provides as well) via their public interfaces of typed ports. This loose coupling enables service developers to reuse other services without changing the configuration of their services. The PersistencyService employs the replicating subcomponent Persistence, which replicates itself whenever necessary.

12.5 SYSTEM-OF-SYSTEMS APPROACH

As stated before, modern cloud-computing architectures are made up of several services [33]. These services are composed from other fundamental services to added-value service, relying on personal data. The overall service landscape of such a cloud-computing is huge and quite heterogeneous. The necessary service compositions are manually implemented and specific to the desired service combi-

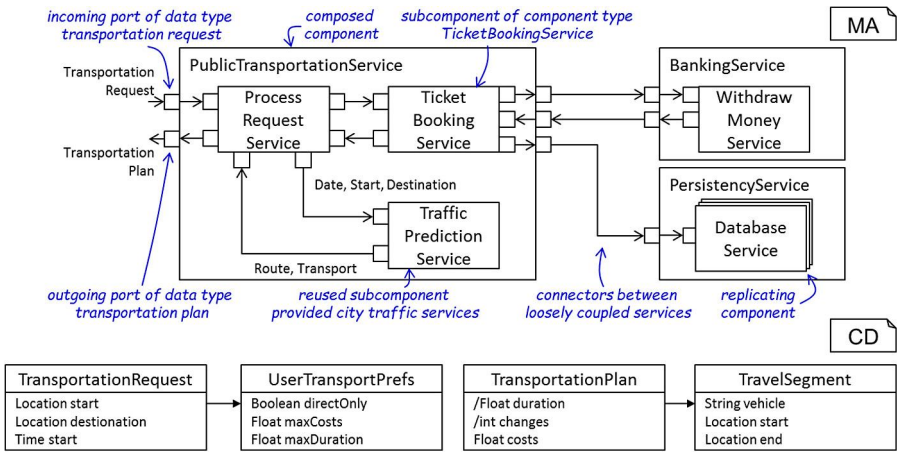

FIGURE 12.3

C&C software architecture for a public transportation service. It consists of composed and atomic components that interact via their interfaces of typed ports. The PublicTransportationService incorporates a services provided by another developer (the TrafficPredictionService) and is loosely coupled to services of other developers. An excerpt of the related UML/P class diagram data types is depicted at its bottom

nation. For alleviating the manual composition of services the different integration and communication challenges, as described have to be tackled.

For this we follow the typical distinction of Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Service-as-a-Service (SaaS) [59]. Within this contribution we do not consider IaaS and assume that there is always sufficient infrastructure available that can be used on demand in order to be elastic. For the distinction between PaaS and SaaS, we consider PaaS as the layer containing common services used for developing services of the SaaS layer. The SaaS Services are used by persons and typically have some sort of GUI whereas the PaaS services are used by those services in order to ease development [59]. Service provider want to offer high quality functionality, thus the services within our architecture have to be elastic regarding high workloads and responsive to provide such a quality. Furthermore, it should be developed with minimal costs. These requirements pose additional requirements to the service developer. The developer needs to increase reuse of developed services since it is not feasible to reimplement every bit of the desired functionality and the parts that have to be implemented must be easily implementable.

We consider persistency, privacy, service lookup and data transformation services as available platform services. These services can be replicated on demand but are, just like other services, developed in a standalone manner. We start out with the introduction of the Persistence service utilized by our architecture, followed by Data Conversion-, Privacy-, and Personal Data service.

12.5.1 PERSISTENCE SERVICE

The Persistence service, as shown in Fig. 12.4, offers the functionality to store data. Most applications and services need to be able to store data in order to be able to make it available again to other users or

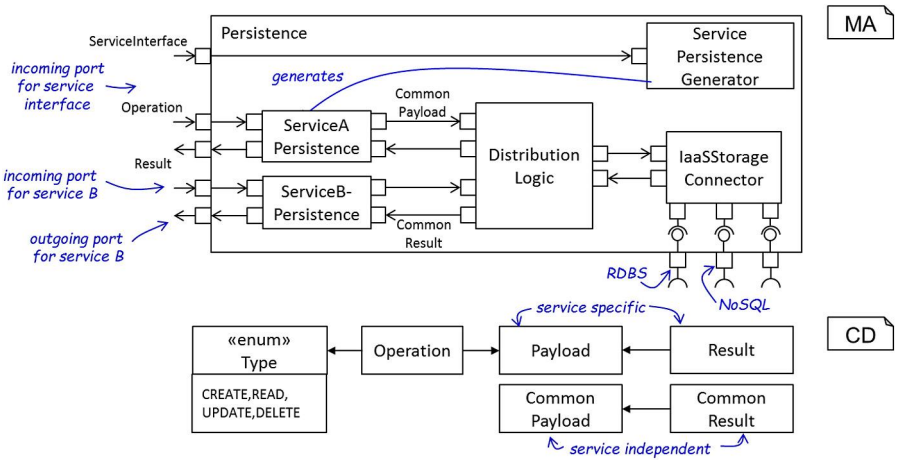

FIGURE 12.4

Persistence component realizes store and retrieve functionality for supported services. The SchedulerPartGenerator component generates a scheduler for each supported service in order to partition data to suitable database back-ends

services. In general object oriented or relational databases [31] and NoSQL databases, such as Apache Cassandra and MongoDB [50] can be distinguished. Each paradigm has its benefits and drawbacks. Relational databases, such as Oracle or PostgreSQL, are especially suited to store structured data. Most of those relational databases are able to fulfill the ACID principles [27] and can therefore ensure consistency. On the downside they may not keep up with processing a huge amount of data. Relational databases organize data in tables with different columns. Each entry of a table is represented as a row. Each data field of a data entry becomes an entry of a column in the corresponding row. Different types of data entries are stored in different tables and merged via join operations. In contrast, NoSQL databases are well suited to process large amounts of data by relaxing the CAP-theorem [8] in terms of not ensuring consistency. Thus, they are the de facto standard supporting digitalization where big data approaches become even more important. They basically provide key value storage functionality and do not support relational structures. On the downside, joins between data are costly.

Nevertheless, both approaches are not mutually exclusive. Moreover, a hybrid approach is necessary that selects the best approach for a certain type of data. This may also lead to splitting up a data model into a structured and a volume part.

The Persistence service consists of the Persistence component that contains several subcomponents, as shown in Fig. 12.4. As stated before we envision a description of a service interface in order to compose them. This description can be sent once to the persistence component where it is used as input for the ServicePersistenceGenerator. This generator generates service specific components and ports within the Persistence component. Thus, each service has an incoming and an outgoing port in order to store or retrieve data. Via the incoming port an Operation is sent. This Operation has a Type that specifies the CRUD type of the operation. Additionally, it has a service specific Payload containing the data. The service specific component ServiceAPersistence receives the Operation and transforms

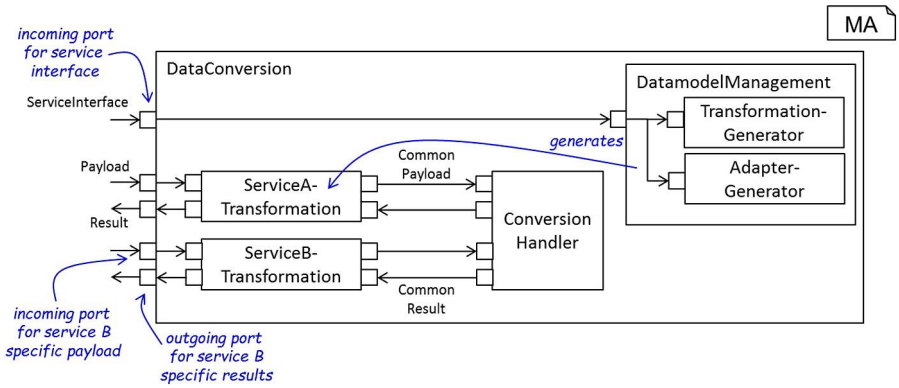

FIGURE 12.5

DataConversion component transform requested data specific to a service. DatamodelManagement component generates and instantiates specific transformation components on demand

the specific payload into a CommonPayload. This CommonPayload is passed on to the DistributionLogic component. The DistributionLogic component decides which database to use based on the incoming payload. The DistributionLogic communicates this decision to the IaaSStorageConnector that directly communicates with a database. In Fig. 12.4 three service ports are given as an example. The RDBS service port is able to store data in a relational database, and the NoSQL service port is able store data in a NoSQL database. For reading data, an Operation with Type read is sent to the Persistence component. The data is loaded into a service independent result from the database and transformed into a service specific result in the ServiceAComponent.

The Persistence service consists of service specific parts and parts necessary for fulfilling the functionality. The latter parts can be implemented once whereas the service specific parts are generated out of an abstract description of a service using the Persistence service. It should be noted that there is not one single Persistence service but several instantiations. For realizing the Digital Me, the Persistence service is necessary since it enables storing the data transparently from the location. Especially since the PersonalData service can instantiate its own Persistence service that stores data in a trusted location.

12.5.2 DATACONVERSION SERVICE

The DataConversion service, as shown in Fig. 12.5, transforms data from one data model to another. This is needed since data models typically do not match. For this the DataConversion service follows the same idea as the Persistence service that was shown in Fig. 12.4. It also uses the common interface description of a using service that can be uploaded to the DataConversion component. The description is delegated to the DatamodelManagement subcomponent that contains two different generator components: the TransformationGenerator and the AdapterGenerator. Both generator components produce parts of the service specific components contained in the DataConversion component. Such service components, i.e., ServiceATransformation and ServiceBTransformation, are located within

the DataConversion component. This transformation is able to transform the service specific data model into a generic metamodel. The metamodel itself consists of a Root that contains several Components. The components are designed with a composite pattern. Thus, it has a subclass Directory that can contain multiple components and a subclass Entry that may have a Value. Each Entry has a name, each Value a name and a value. This metamodel is based on the Internet-of-Things data model [30]. The metamodel allows storing heterogeneous kinds of data by using the datatype as the Entry name and each field as a Value. This transformation from and to a metamodel is part of the generated functionality of the service specific components. They make use of the transformation into the metamodel for transforming from and to the different service data models. This is transparent to other services. Other services are able to simply use the service once they have uploaded their interface description. Thus, they can exchange data between different services. This exchange is possible since we use a generative approach, which is presented in the next section. That generative approach connects to services via the DataConversion component by interrupting the data flow with a connection from the incoming port of one service to the outgoing port of the other service. The DataConversion service that is shown in Fig. 12.5 may use the Persistence service in order to store the data of different services in a metamodel data format. The concept of transforming into the metamodel as well as the generation of such transformations has been applied in the COOPERATE research project and has been presented in [21]. It should be noted that the DataConversion service itself is stateless and may therefore have multiple instances and even its own instance of a persistence service.

Within the cloud-architecture we use the DataConversion component to interconnect different services supporting the Digital Me in composing services as needed.

12.5.3 PRIVACY SERVICE

The PrivacyService component depicted in Fig. 12.6 enables connected components to exchange data without the need to expose confidential personal data, like contact addresses. The core functionality is based on the ServiceADataDelivery component. An external component, like ServiceB, connects to the ServiceADataDelivery component port and on the other side the concrete ServiceA is connected. ServiceA has registered at the TokenRegistry and a Token is created. ServiceB uses this Token to establish a connection to ServiceA. Using this indirection in communication between components, components like ServiceA do not have to expose their personal long term contact details to foreign components. This supports the Digital Me by allowing communication by distributing tokens. These tokens could be made publicly available in order to retrieve offers from other services. Unlike the permanent contact address tokens can be easily revoked and such prevent further communication with this token. Additionally, the tokens can have attached limitations, like an expiration date, or the token can only be used by a certain components. Limitations are defined in the TokenRequest and are then connected to the resulting Token. For each component, a DataDelivery component can be generated by the DataDeliveryGenerator component.

12.5.4 LOOKUP SERVICE

The LookUp service supports finding the PersonalData services. It uses the functionality of the Privacy service to make the LookUp service publicly available without direct exploitation of the PersonalData service. Other services have to process the personal data of multiple users. For this, they have to be

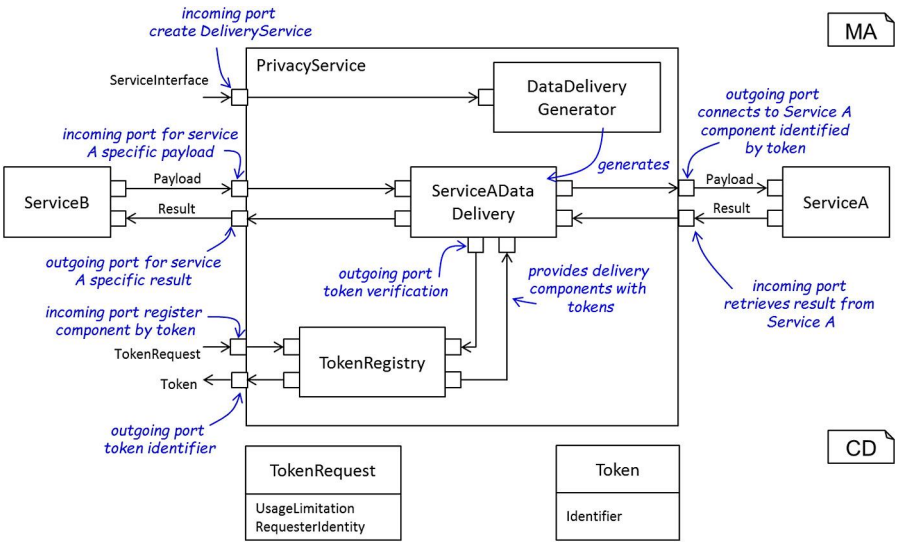

FIGURE 12.6

PrivacyService component ensures decoupled communication between components. Tokens hide away concrete identification and supports temporary access to components

able to find and address the PersonalData service of each single user. Therefore, the LookUp service gets an anonymous token under which a user is known to the service. The LookUp service is able to process this token and to return an instance of the PersonalData service of the respective user. The single services for personal data are all known to a trusted registry within the cloud-architecture described in this paper. The registry is a single endpoint known to all LookUp services. The LookUp service itself can be replicated multiple times within the architecture.

12.5.5 PERSONALDATA SERVICE

Fig. 12.7 shows the PersonalData service. This service uses the previously introduced DataConversion and Persistence services. The PersonalData service is the implementation of the Digital Me as digital representation of a user within the cloud-architecture. The service itself can be instantiated locally, e.g., on the user's phone or remote in a trusted location. Within this paper we assume that the personal data service and especially the Persistence service used by the PersonalData service run on the user's phone. It should be noted that the previously explained services are instantiated multiply. Thus, there is not one single persistence service but several, as stated before. Services querying personal data may use this service. Thus, they post a read Operation to the PersonalData component. This Operation is delegated to the data conversion where it is transformed into the data format of the personal data. The data is loaded from the respective databases transformed back and handed back to the service as a Result.

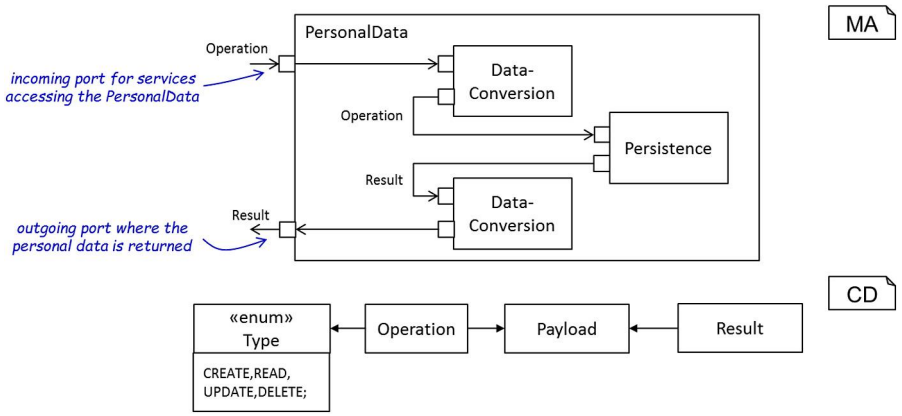

FIGURE 12.7

PersonalData component is composed of DataConversion and Persistence components, to store personal data and provide different data formats through conversions

Overall the fundamental PaaS services of the presented cloud-architecture provide a set of typically used functionality. A service that wants to use other services by composition can connect to those services via the DataConversion service which transforms the data adequately in order to communicate between the different services. If the services want to use personal data of a specific user, they can find the PersonalData service via the LookUp service. Within the PersonalData service the querying service is authenticated and authorized by the Security service. The data is loaded by the Persistence service. On its way back to the querying service the loaded data is transformed into the services data format and anonymized before it is returned. By this the service user can easily reuse the existing services and compose new services.

12.6 GENERATIVE APPROACH

The services, as described in Section 12.5, only provide basic functionality such as persistence and lookup, each of which is developed by a different service provider. In order to develop a new service for the scenario described in Section 12.2, such basic services have to be composed. However, there are two major issues that have to be tackled when developing a new service: *specification of exchanged data* and *service composition*.

Assuming the higher-level service to negotiate an appointment for a person with a doctor, the mobility service is to be developed, as shown in Fig. 12.8. The first step is to define the data that is exchanged by the involved parties. In a privacy-aware environment, users specify what information is exchanged. For example, Fig. 12.1 shows the usage scenario and what information is allowed to be exchanged. First, the doctor is allowed to use a person's address (TAddress) and negotiate appointments (Appointment). Second, a person is allowed to access the doctor's address (DAddress) but not his appointments. Third, the mobility service is allowed to use a person's address, the doctor's

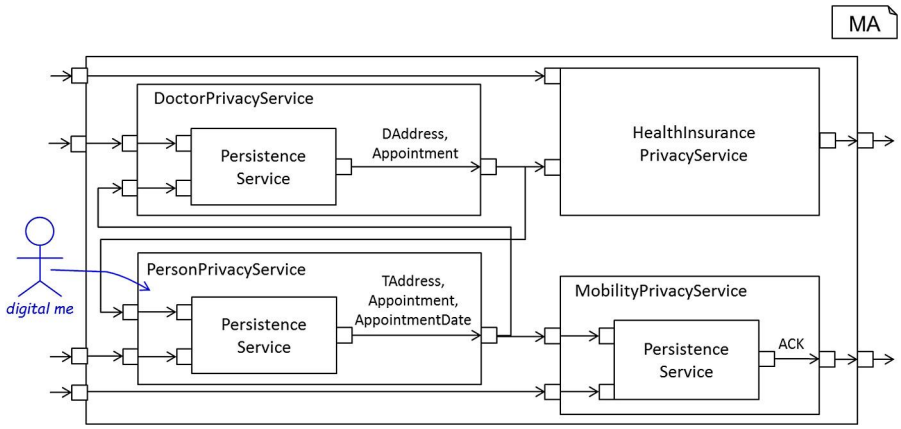

FIGURE 12.8

MontiArc model of the new higher-level service showing only the exchanged data

address, and the date of the appointment. Finally, the health insurance may access the appointment date (AppointmentDate). Such specification of the exchanged data is the foundation for the service developer to develop a new higher-level service.

The next step is the design of the architecture of the overall service by the service developer. One possible description of this new service is shown in Fig. 12.8. It is described using the MontiArc ADL. Such an abstract description can *manually* be realized by implementing the necessary functionality using the basic services. However, when manually implementing such a new service, the service developer has to write glue code for service composition and afterwards adapt services if required. Existing service composition approaches help in realizing such an implementation, such as [53,22], which give an overview of existing composition approaches.

Another way of developing a new higher-level service is a *generative approach*. The main idea is to develop such new services by systematically transforming abstract representations into executable source code. This is done by employing a *code generator*. The example shown in Fig. 12.9 shows a high-level view on the system neglecting privacy, type safety, and even communication concerns. For example, the DoctorPrivacyService requires the address in a different format, which is not considered in the high level. Clearly, such implementation details have to be implemented manually by service providers, when choosing not to use a generative approach.

In order to reduce the required implementation efforts, a generative approach can automatically synthesize a more detailed description of a service, which is based on the aggregation of basic services and respects the particular needs of each basic service. Given the abstract description in Fig. 12.8, a more detailed description as shown in Fig. 12.9 can be generated.

Using the abstract description of each basic service, which specifies how to use the service, type conversion, as well as communication conversion, can be established. For example, in Fig. 12.9 the DoctorPrivacyServiceWrapper can negotiate an appointment with the PersonPrivacyServiceWrapper only by using the LookUp service, which ensures that the digital representation of a Person is used.

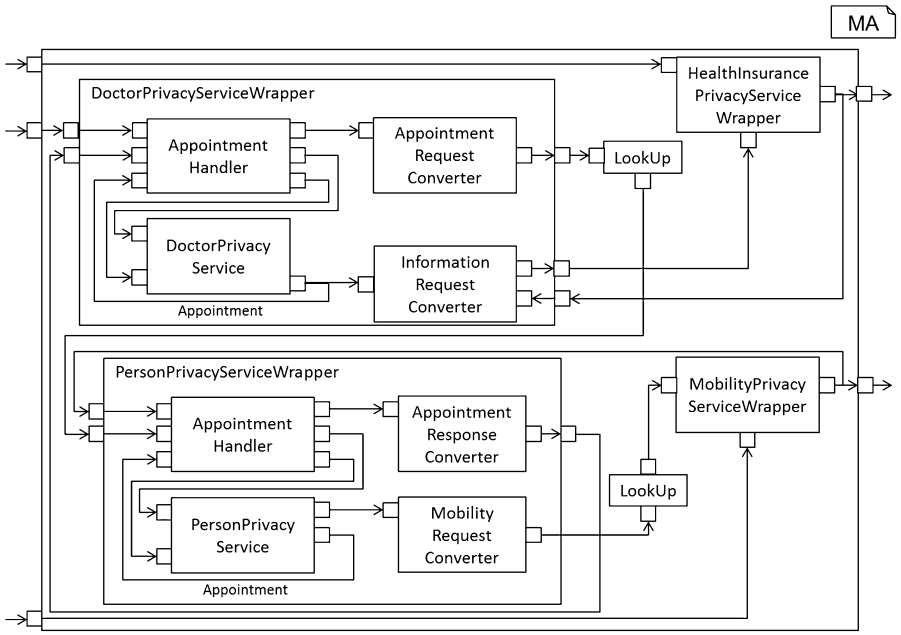

FIGURE 12.9

MontiArc model of the higher-level implementation details including data conversion and additional appointment requests

A generative approach provides essential advantages. First, for each PrivacyService shown in Fig. 12.9 a wrapper is generated, which ensures data conversion and provides additional components to implement business logic, e.g., AppointmentHandler in the DoctorPrivacyServiceWrapper. Hence, the service developer can focus on implementing business logic rather than composition of services. Conversion between different formats can automatically be ensured by adding additional components such as the AddressRequestConverter component, which concerts Appointment into the required format for the PersonPrivacyServiceWrapper component. Second, in a generative approach targeting a privacy-aware environment, additional LookUp components can automatically be added.

12.7 RELATED WORK

The related work of our approach domes from different research directions. In Section 12.3 we already described the challenges we faced, such as service composition, technology abstraction, service and data integration, as well as trusted use of personal data. In addition to these challenges, we make use of ADLs as well as generative approaches. For discussing related work, we keep up the identified challenges.

12.7.1 SERVICE COMPOSITION

Most approaches supporting service composition make use of plugin systems, such as OSGi [42], or other plugin architectures [5,43]. These plugin architectures consist of several plugins that are bound to other plugins at runtime. Therefore different services are able to use other services without knowing them beforehand. Typically, service lookup and discovery is done via a service registry. While there are many such architectures, OSGi is the most widespread plugin system.

12.7.2 TECHNOLOGY ABSTRACTION

The technology abstraction is achieved by making use of ADLs and generative approaches. The ADLs have already been discussed in detail in Section 12.4 where we provided preliminaries for the paper and introduced related languages and concepts. Furthermore, we employ generative approaches in order to generate technology specific code from our abstract descriptions, as it has already been proposed in [10,52,4]. However, because abstraction does not always suffice to describe the required functionality, additional approaches to integrate handwritten extensions have to be employed. A detailed overview of currently existing approaches is presented in [19,20]

12.7.3 SERVICE AND DATA INTEGRATION

There are several approaches to service and data integration in the literature. Some focus on finding a common language for modeling data, some focus on finding a common data model, while others focus on semantically integrating different data models in a highly automated fashion. A good overview is given in [28] and in [44]. As an example for approaches that aim at providing a common language for modeling data, the plethora of XML derivatives [3] and their corresponding query languages [7] has to be named. More approaches integrating data via XML are again given in [28]. Apart from that, there are approaches providing a common data model for specific purposes, resulting in different standards. Currently there are data models emerging for nearly all concerns, including Building Information Modeling [16] or sensor streams in the Internet-of-Things approaches [9], to name only a few. Additionally, many ontology approaches emerge, including ontology matching and merging approaches [38,39,13]. Other approaches focus on mapping the different data schemas in an automated fashion in order to automatically integrate the different data [37]. These approaches aim at integrating different data sources [21] or employing machine learning techniques in order to learn the mapping [12,15].

12.7.4 TRUSTED USE OF PERSONAL DATA

A similar approach to our proposed architecture is taken by the di.me project, funded by the European Union. The project focusses on an integrated personal information sphere [51]. Such a personal information sphere contains several digital faces, i.e., data the user selected for sharing with other services. The approach taken by the di.me project allows a fine-grained approach to protecting personal data. Nevertheless, the cloud-architecture in this paper focusses on the composition of services in order to support the user in a digitalized world. This of course includes protecting the information but goes beyond that. Apart from this, the di.me project focusses on a centralized data sharing platform which is a different concept from the approach undertaken within this paper. In [32] an information management assistance system is presented that allows users to find out who has which data and to monitor

the data flow. The aim of the cloud-architecture presented within this paper is not only to monitor the flow of data but also to be able to change the data consistently at a single point. In [55], an architecture for privacy-enhanced mobile communities is presented. This architecture contains several services that have to adhere to the architecture specifications. Thus the services are more strongly coupled whereas our architecture aims at the integration of heterogeneous services.

12.8 DISCUSSION

Our approach enables service users to model data access without explicit knowledge about the implementation details facilitating reuse of platform independent models in an agile development environment. However, it does not ensure that the services once granted access to a specific data do neither copy it, nor pass it on. Approaches to data tracing have been presented for homogeneous, enclosed systems, such as the Android Kernel [14]. Whether their practices are feasible for heterogeneous cloud-based services is subject to current research. Having the Digital Me as a single data store for user data introduces a possible bottleneck and honeypot to the ecosystem. However, research big data management and security to fix this is ongoing.

While we presented our approach following a scenario, the approach is not limited to it. The presented basic components are generic regarding the scenario but provide necessary functionality in any software ecosystems. The presented components may not be complete and may have to be extended due to changes in technology or research. Nevertheless, the presented methodology enables providing new components. Additionally, existing components might be extended or updated due to new requirements. This has to be done by the provider of such components. Nevertheless, the design and methodology of the overall approach remain and benefit greatly from the generative approach taken. These benefits are a reduced time-to-market and a significantly improvement in certifying the generated code by certifying the code generator. However, a generative approach can only automatically synthesize wrapper components if the underlying components do exist, such as the basic components presented in Section 12.3. For instance, in cases where no converter is present in the basic services, a service shell is synthesized, allowing service providers to implement the required functionality. A possible approach is to automatically generate OSGi Plugins, which realize the conversion allowing for easy adaptation.

Another disadvantage is that, whenever additional business logic is required such as negotiation of appointments, the business logic has to be implemented manually by the service provider. However, the granularity of the employed modeling languages [25] may not provide a sufficient abstraction such that additional approaches are required to add the wanted granularity. This can, e.g., be achieved by using handwritten extensions such as proposed in [19,20], where an overview of possible handwritten extension approaches is presented.

12.9 CONCLUSION

We have presented a vision of model-driven cloud architecture engineering that relies on reusable service components. It enables developers of cloud services and architecture to efficiently build upon

existing services. These services exist in the context of ecosystems providing important base services to support reuse, service composition, and user data management. For the latter, the notion of a Digital Me provides benefits for all participating roles: it facilitates data access control and data update for service users and it liberates service developers from providing data management features. Furthermore, it always yields the most up-to-data user data available. Using a generative approach, services for the digitized world can be developed on a more efficiently on a better suitable, namely higher, level abstraction. This ultimately enables each role, participating in service development, to contribute their domain expertise to dedicated challenges and facilitates service component reuse.

REFERENCES

[1] A. Angerer, R. Smirra, A. Hoffmann, A. Schierl, M. Vistein, W. Reif, A graphical language for real-time critical robot commands, in: Proceedings of the Third International Workshop on Domain-Specific Languages and Models for Robotic Systems, 2012.

[2] M. Broy, F. Huber, B. Schätz, AutoFocus—Ein Werkzeugprototyp zur Entwicklung eingebetteter Systeme, Inform. Forsch. Entwickl. 14 (3) (1999).

[3] S. Boag, D. Chamberlin, M.F. Fernández, D. Florescu, J. Robie, J. Siméon, M. Stefanescu, XQuery 1.0: An XML query language, 2002.

[4] M. Brambilla, J. Cabot, M. Wimmer, Model-Driven Software Engineering in Practice, 1st edition, Morgan & Claypool Publishers, 2012.

[5] D. Birsan, On plug-ins and extensible architectures, Queue 3 (2) (2005) 40–46.

[6] H. Bruyninckx, M. Klotzbücher, N. Hochgeschwender, G. Kraetzschmar, L. Gherardi, D. Brugali, The BRICS component model: a model-based development paradigm for complex robotics software systems, in: Proceedings of the 28th Annual ACM Symposium on Applied Computing, ACM, 2013.

[7] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, F. Yergeau, Extensible markup language (XML). World Wide Web Consortium Recommendation REC-xml-19980210, http://www.w3.org/TR/1998/REC-xml-19980210, 1998.

[8] E. Brewer, Pushing the CAP: strategies for consistency and availability, in: Computer, IEEE Computer Society Press, 2012.

[9] P. Barnaghi, W. Wang, L. Dong, C. Wang, A linked-data model for semantic sensor streams, in: Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing, IEEE, 2013, pp. 468–475.

[10] K. Czarnecki, U.W. Eisenecker, Generative Programming: Methods, Tools, and Applications, Addison-Wesley, ISBN 0201309777, 2000.

[11] E. Cavalcante, A.L. Medeiros, T. Batista, Describing cloud applications architectures, in: Software Architecture, Springer, Berlin, Heidelberg, 2013.

[12] A. Doan, P. Domingos, A.Y. Halevy, Reconciling schemas of disparate data sources: a machine-learning approach, SIGMOD Rec. 30 (2) (2001) 509–520.

[13] L. Ding, P. Kolari, Z. Ding, S. Avancha, Using ontologies in the semantic web: a survey, in: Ontologies, Springer, US, 2007, pp. 79–113.

[14] M. Dam, G. Le Guernic, A. Lundblad, TreeDroid: a tree automaton based approach to enforcing data processing policies, in: Proceedings of the 2012 ACM Conference on Computer and Communications Security, 2012.

[15] A. Doan, J. Madhavan, P. Domingos, A. Halevy, Learning to map between ontologies on the semantic web, in: Proceedings of the 11th International Conference on World Wide Web, ACM, 2002, pp. 662–673.

[16] C. Eastman, C.M. Eastman, P. Teicholz, R. Sacks, K. Liston, BIM Handbook: A Guide to Building Information Modeling for Owners, Managers, Designers, Engineers and Contractors, John Wiley & Sons, 2011.

[17] P.H. Feiler, D.P. Gluch, Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language, Addison-Wesley, 2012.

[18] R. France, B. Rumpe, Model-driven development of complex software: a research roadmap, in: Future of Software Engineering (FOSE '07), 2007.

[19] T. Greifenberg, K. Hölldobler, C. Kolassa, M. Look, P. Mir Seyed Nazari, K. Müller, A. Navarro Perez, D. Plotnikov, D. Reiß, A. Roth, B. Rumpe, M. Schindler, W. Wortmann, A comparison of mechanisms for integrating handwritten and generated code for object-oriented programming languages, in: Conference on Model-Driven Engineering and Software Development, SciTePress, 2015.
[20] T. Greifenberg, K. Hölldobler, C. Kolassa, M. Look, P. Mir Seyed Nazari, K. Müller, A. Navarro Perez, D. Plotnikov, D. Reiß, A. Roth, B. Rumpe, M. Schindler, W. Wortmann, Integration of handwritten and generated object-oriented code, in: Model-Driven Engineering and Software Development Conference, in: CCIS, vol. 580, Springer, 2015.
[21] T. Greifenberg, M. Look, B. Rumpe, K.A. Ellis, Integrating heterogeneous building and periphery data models at the district level: the NIM approach, in: Proceedings of the 10th European Conference on Product and Process Modelling, ECPPM 2014 – eWork and eBusiness in Architecture, Engineering and Construction, Vienna, Austria, CRC Press/Balkema, Netherlands, 2014.
[22] M. Garrigam, C. Mateos, A. Flores, A. Cechich, A. Zunino, RESTful service composition at a glance: a survey, J. Netw. Comput. Appl. 60 (2016).
[23] M.J. Hadley, Web Application Description Language (WADL), Sun Microsystems, Inc., 2006.
[24] C. Herrmann, H. Krahn, B. Rumpe, M. Schindler, S. Völkel, Scaling-up model-based-development for large heterogeneous systems with compositional modeling, in: Proceedings of the 2009 International Conference on Software Engineering in Research and Practice, vol. 1, 2014.
[25] A. Haber, M. Look, P. Mir Seyed Nazari, A. Navarro Perez, B. Rumpe, S. Völkel, A. Wortmann, Integration of heterogeneous modeling languages via extensible and composable language components, in: Conference on Model-Driven Engineering and Software Development, SciTePress, 2015.
[26] F. Höwing, Effiziente Entwicklung von AUTOSAR-Komponenten mit domänenspezifischen Programmiersprachen, in: INFORMATIK 2007: Informatik trifft Logistik. Band 2. Beiträge der 37. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 24.–27. September 2007 in Bremen, Deutschland, 2007, pp. 551–556, http://subs.emis.de/LNI/Proceedings/Proceedings110/article1774.html.
[27] T. Härder, A. Reuter, Principles of transaction-oriented database recovery, in: ACM Computing Surveys, ACM, 1983.
[28] A. Halevy, A. Rajaraman, J. Ordille, Data integration: the teenage years, in: Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB Endowment, 2006, pp. 9–16.
[29] A. Haber, J.O. Ringert, B. Rumpe, MontiArc – architectural modeling of interactive distributed and cyber-physical systems, 2012.
[30] IoT.est, Internet of things environment for service creation and testing (IoT.est). Available at: http://ict-iotest.eu/iotest/, 2013.
[31] A. Kemper, A. Eickler, Datenbanksysteme: Eine Einführung, Oldenbourg Verlag, 2011.
[32] S. Labitzke, Who got all of my personal data? Enabling users to monitor the proliferation of shared personally identifiable information, in: Privacy and Identity Management for Life, Springer, Berlin, Heidelberg, 2011.
[33] M.W. Maier, Architecting principles for systems-of-systems, in: INCOSE International Symposium, 1996.
[34] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, A. Tang, What industry needs from architectural languages: a survey, IEEE Trans. Softw. Eng. 39 (6) (2013).
[35] N. Medvidovic, R.N. Taylor, A classification and comparison framework for software architecture description languages, IEEE Trans. Softw. Eng. 26 (1) (2000) 70–93.
[36] N. Medvidovic, E. Dashofy, R.N. Taylor, Moving architectural description from under the technology lamppost, Inf. Soft. Technol. 49 (1) (2007) 12–31, http://dx.doi.org/10.1016/j.infsof.2006.08.006.
[37] T. Milo, S. Zohar, Using schema matching to simplify heterogeneous data translation, VLDB J. 98 (1998) 24–27.
[38] N.F. Noy, M.A. Musen, SMART: automated support for ontology merging and alignment, in: Proc. of the 12th Workshop on Knowledge Acquisition, Modelling, and Management (KAW'99), Banf, Canada, 1999.
[39] N.F. Noy, M.A. Musen, The PROMPT suite: interactive tools for ontology merging and mapping, Int. J. Hum.-Comput. Stud. 59 (6) (2003) 983–1024.
[40] A. Navarro Pérez, B. Rumpe, Modeling cloud architectures as interactive systems, in: MDHPCL@ MoDELS, 2013.
[41] Object Management Group, OMG unified modeling language (OMG UML), Infrastructure Version 2.3 (10-05-03), 2010.
[42] OSGi Alliance, OSGi Service Platform, Release 3, IOS Press, Inc., 2003.
[43] J. Rathlev, Anwendungsentwicklung mit Plug-in-Architekturen: Erfahrungen aus der Praxis, in: Software Engineering, 2011, pp. 183–196.
[44] E. Rahm, P.A. Bernstein, A survey of approaches to automatic schema matching, VLDB J. 10 (4) (2001) 334–350.

[45] A. Ramaswamy, B. Monsuez, A. Tapus, Model-driven software development approaches in robotics research, in: Proceedings of the 6th International Workshop on Modeling in Software Engineering, ACM, 2014.
[46] J.O. Ringert, B. Rumpe, A. Wortmann, Multi-platform generative development of component & connector systems using model and code libraries, in: ModComp Workshop 2014—1st International Workshop on Model-Driven Engineering for Component-Based Systems, Valencia, Spain, in: CEUR Workshop Proceedings, vol. 1281, 2014.
[47] J.O. Ringert, B. Rumpe, A. Wortmann, Language and code generator composition for model-driven engineering of robotics component & connector systems, J. Softw. Eng. Robot. 6 (1) (2015) 33–35.
[48] B. Rumpe, Modellierung mit UML: Sprache, Konzepte und Methodik, 2nd edition, Springer, Berlin, 2011.
[49] B. Rumpe, Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring, 2nd edition, Springer, Berlin, 2012.
[50] E. Redmond, J. Wilson, Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement, Pragmatic Bookshelf, 2012.
[51] S. Scerri, R. Gimenez, F. Herman, M. Bourimi, S. Thiel, Digital.me – towards an integrated personal information sphere, in: Proceedings on the Federated Social Web Summit, 2011.
[52] T. Stahl, V. Markus, Model-Driven Software Development, John Wiley & Sons Ltd., 2006.
[53] S.E. Tbahriti, C. Ghedira, B. Medjahed, M. Mrissa, Privacy-enhanced web service composition, IEEE Trans. Serv. Comput. 7 (2) (2014).
[54] U. Thomas, G. Hirzinger, B. Rumpe, C. Schulze, A. Wortmann, A new skill based robot programming language using UML/P statecharts, in: ICRA IEEE International Conference on Robotics and Automation, IEEE, 2013, pp. 461–466.
[55] M. Tschersich, C. Kahl, S. Heim, S. Crane, K. Böttcher, I. Krontiris, K. Rannenberg, Towards privacy-enhanced mobile communities—architecture, concepts and user trials, J. Syst. Softw. 84 (11) (2011) 1947–1960.
[56] M. Völter, T. Stahl, J. Bettin, A. Haase, S. Helsen, K. Czarnecki, B. von Stockfleth, Model-Driven Software Development: Technology, Engineering, Management, Wiley, 2013.
[57] J. Whittle, J. Hutchinson, M. Rouncefield, The state of practice in model-driven engineering, IEEE Softw. 31 (3) (2014).
[58] D.S. Wile, Supporting the DSL spectrum, J. Comput. Inf. Technol. 9 (2001).
[59] Q. Zhang, L. Cheng, R. Boutaba, Cloud computing: state-of-the-art and research challenges, J. Internet Serv. Appl. 1 (1) (2010).