A Model-based Software Development Kit for the SensorCloud Platform

Lars Hermerschmidt, Antonio Navarro Perez, and Bernhard Rumpe
{hermerschmidt, perez, rumpe}@se-rwth.de

Department of Software Engineering,
RWTH Aachen University,
Aachen, Germany

Abstract. The development of software for the cloud is complicated by a tight entanglement of business logic and complex non-functional requirements. In search of a solution, we argue for the application of model-based software engineering to a particular class of cloud software, namely *interactive cloud software systems*. In particular, we outline an architecture-driven, model-based method that facilitates an agile, top-down development approach. At its core it employs a software architecture model that is augmented with additional aspect-specific models. These models serve as concise, formal, first-class specification documents and, thus, as foundation for tool-supported analysis and synthesis, in particular, code generation. We hypothesize that many crucial cloud-specific non-functional requirements can be satisfactory addressed on the architecture level such that their realization in the system can be synthesized by tools with small impact on the business logic.

Note: This paper is based on the technical report *clArc – Architecture-Driven, Model-based Engineering of Cloud Software* which is currently in process of publication in the *Aachener Informatik Berichte* series.

1 Introduction

The number of software-driven embedded devices has been growing for decades. Such devices perceive and affect the real world, serve a multitude of diverse purposes, and are found in many public, industrial, and domestic places. They control vehicles, traffic infrastructure, factory machines, power plants, and energy grids. They measure electricity, temperatures, fill levels, global positioning, velocities, and operating conditions, or capture visual and audio information.

Enabled by the ongoing pervasiveness and sinking costs of internet connectivity, more and more embedded devices are no longer locally isolated but globally accessible and globally interconnected. Information and control shift from the individual device to the internet. Integrated, large-scale distributed systems emerge as a consequence. Their intimate relation to the real world and their network- and software-driven logic has led to the common label of *cyber-physical systems*. [1]

The software that drives those systems is of a *interactive/reactive* character: it reacts to impulses from its environment (e.g. sensor input or user interaction) with commands that control physical parts of the system (e.g. actors or monitoring systems). The software is also time-sensitive: its reactions have to happen within a given time window.

New use cases explore the potentials of cyber-physical systems, for instance, smart grids, smart traffic and smart homes. [2–4] The development of systems that implement these use cases, however, is complicated by their inherent complexity and subsequent development costs. Engineers of such systems must integrate system components and aspects on a technological level (e.g. devices, networking, protocols, infrastructure resources, software architecture), on an economic level (e.g. supported business models, integration with legacy systems and processes, customer and installation support), and from the perspective of legislators (e.g. privacy and customer autonomy). As a consequence of this tight integration of concerns, only big companies with big budgets can successfully implement such systems and bring them to product-ready matureness.

The *SensorCloud* project develops a large scale, cloud-based platform for services based around internet-connected sensor and actor devices. [5] The platform's technological purpose is to provide a generalized and easy-to-use infrastructure for sensor- and actor driven cloud services. Its economic purpose is to modularize and industrialize the development and provision of such services. Multiple different stakeholders can contribute different parts of the service stack, for instance, sensor/actor hardware, domain-specific data sets, domain-specific services, and client software.

From a software engineering perspective, the success of such a platform is largely influenced by the efficiency of development of software for the platform. In the context of the SensorCloud, third-parties develop *cloud services* that leverage sensors, sensor data, and actors in order to provide functionality for end customers. However, the development of cloud services is complicated by a tight entanglement of business logic and complex non-functional requirements.

In this paper, we describe the concepts and modeling languages at the core of a model-based SDK for developing cloud-based software in general and SensorCloud services in particular. This SDK is based on the *clArc toolkit*. It is particularly aligned to the domain of cloud-based cyber-physical systems and understands them more generally as *interactive cloud software systems*. At its core, this toolkit core employs a software architecture model as well as accompanying aspect-specific secondary models. These models serve as concise, formal, first-class specification documents and, thus, as foundation for tool-supported analysis and synthesis, in particular, code generation.

Chapter 2 describes the clArc toolkit underlying the SDK. Chapters 3 and 4 describe the architecture style and modeling language at the core of the toolkit. Chapters 5 and 6 describe additional modeling languages for deployment and testing. Chapter 7 briefly outlines the tooling for processing and executing these modeling languages.

2 Model-based Engineering of Cloud Software

clArc (for *cl*oud *arc*hitecture) is a model-based, architecture-centric toolkit for developing *interactive cloud software* systems. It is based on the language workbench MontiCore [6–8] and the architecture description language MontiArc [9, 10]. The toolkit

1. defines a *software architecture style* for distributed, concurrent, scalable and robust cloud software that *permanently interacts* with a heterogeneous environment of physical things, services and users,
2. provides textual, executable *domain-specific modeling languages* for specifying and implementing software systems according to that architecture style
3. provides a *framework* for implementing individual code generators that synthesize an executable runtime framework for cloud software based on models written with these DSLs and targeted towards individual cloud infrastructures and platforms.

These three components are part of a methodology that seeks to make the development of interactive cloud software systems more reliable and efficient. It does so by (a) providing an appropriate level of abstraction for the specification of the software system and (b) by providing means to map the software's specification to an executable implementation without methodological discontinuities between design and implementation activities.

2.1 Interactive Cloud Software Systems

The software architecture style [11] of clArc describes architectures as cloud-based [12], interactive [13–15] software systems. Such systems are characterized by permanent interaction (a) on a system level between the software and its environment and (b) on the software level between the internal parts from which the software is composed. Interacting parts (software components) and their interactions (messages passed between components) are both first-level elements of the architecture style. The emphasis on interaction is a result of essential requirements imposed on those systems:

- *Reactiveness*: the software responds to events and requests from its environment within a given time window.
- *Statefulness*: the software continuously and indefinitely tracks and updates a conversational state between itself and its environment.
- *Concurrency*: the software interacts with many interaction partners in parallel and must, hence, be inherently *concurrent*.
- *Distribution*: the software is composed from distributed parts that are deployed on runtime nodes, communicate asynchronously, may replicate dynamically, support failure awareness and can be recovered from failures.
- *Scalability*: the software can adapt to changing work loads by increasing or decreasing the number of its parts and by consuming or releasing necessary computing resources.

These requirements are usually tightly entangled with each other and with the software's actual business functions. As a result, they hinder the software engineer in focusing on the software's business functionality that he actually wants to analyze, realize, test, extend, or modify.

The central idea behind the clArc toolkit is to address these requirements on an appropriate level of abstraction given by a description of the system's *software architecture* according to the architecture style of interactive systems.

2.2   Executable Modeling Languages

Modeling [16–19] can be employed in the engineering of software systems to describe system aspects explicitly and formally. Domain-specific modeling languages are a tool to efficiently write concise models.

Appropriate high-level structure, distribution and interaction are the key factors when reasoning about requirements and overall system design. The explicit description of system aspects through models makes them visible, substantiated and documented in a coherent notation. Conversely, they remain implicit and invisible on the lower levels of system implementation.

However, software developers too often respond to models with skepticism. [20] In many projects, models are still created only at the beginning of the development process and thereafter abandoned. The software is subsequently developed detached from its models. Changes in requirements or design are only incorporated in the software's source code, but not in its models. As a consequence, the models play no essential part in the software's construction. Developers, thus, perceive models as causing additional work without essential contribution to the final product. Models are seen as documentation, at best.

The disconnect between models and product is grounded in process discontinuities caused by a lack of automated transitions between descriptions in models and the final product: models have to be manually transformed into an implementation and manually synchronized with a changing implementation.

The clArc toolkit incorporates the expressive power of models into the development process of cloud software and focuses on the elimination of process discontinuities. Models in clArc are automatically linked to a respective implementation and, thus, *executable*. A set of valid clArc models, therefore, can be compiled into an implementation through a *code generator*.

More precisely, the system aspects described by models are realized in the form of a *generated framework*. This framework provides interfaces for *handwritten code* that implements the systems business logic. In this way, the business logic is cleanly separated from system aspects described by models. Models and code are both first-class artifacts that are integrated via well-defined interfaces. Contrasting other model-based methods, the generated code is treated the same way compiled machine code is treated in traditional programming: it is neither modified manually nor inspected.

Code generators are highly specific to the technological infrastructure the cloud software targets. Consequently, the clArc toolkit does not provide concrete code generators, but (a) a framework to efficiently develop individual code
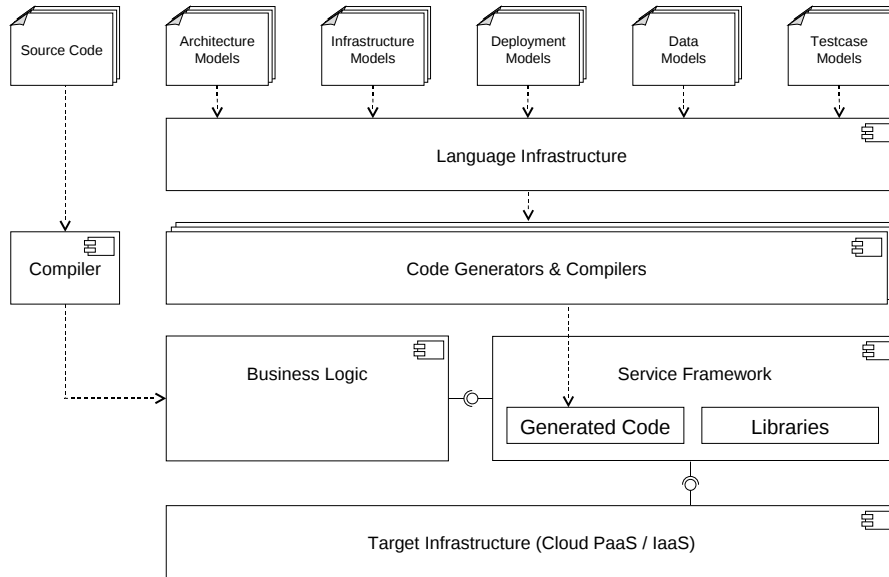
Fig. 1. Code Generation

generators for specific cloud infrastructures and (b) a library of pre-developed standard libraries to be reused in concrete code generators.

3 Modeling Languages

The clArc toolkit employs several textual modeling languages.

The *Cloud Architecture Description Language* (clADL) is the central modeling language of clArc. Its models define *logical software architectures* that implement an *architecture style* targeting the domain of distributed, concurrent, scalable, and robust cloud software. All other languages integrate with each other by referencing clADL models.

The *Target Description Language* (TDL) describes *physical infrastructure architectures* on which software is executed. The *Mapping Description Language* (MDL) relates clADL and TDL models to each other by defining deployments of software architectures onto infrastructure architectures. In combination, these models describe the overall *system architecture* that comprises the software architecture and infrastructure architecture.

In combination, the clADL, TDL and MDL can be used to configure a code generator to generate code according to a specific system architecture given by models of those languages. The generated part of the software architecture's implementation is then custom generated according to the the deployment given by TDL and MDL models.

The *Architecture Scenario Description Language* defines exemplary interaction patterns in a particular software architecture. Models of this language can be used to specify test cases. The *Test Suite Definition Language* configures test setups of such scenarios. Both languages are used for model-based testing of software architecture implementations.

3.1 Architecture Style

The architecture style of clArc uses *components* as the elemental building block of software architectures. Components in clArc are modules with well-defined import and export interfaces. They are executed in their own discrete thread of control and *interact* with each other through *asynchronous message passing* (with FIFO buffers at the receiver's end) over statically defined *message channels*. They encapsulate their state and do not share it with their environment by any other means except explicit message passing.
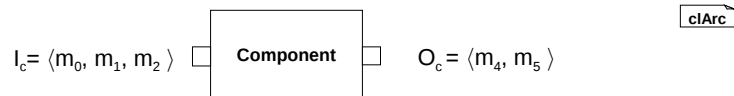
$$I_c = \langle m_0, m_1, m_2 \rangle \quad \boxed{\textbf{Component}} \quad O_c = \langle m_4, m_5 \rangle$$

clArc

Fig. 2. A component receiving incoming messages from the channel I_C and sending outgoing messages to the channel I_C

Components are arranged in a decomposition hierarchy of components. In this hierarchy, leaf nodes represent the atomic building blocks of business logic while inner nodes compose, manage and supervise their immediate child components. Failures are propageted bottom-up through the hierarchy until they are handled or escalate at the root component.
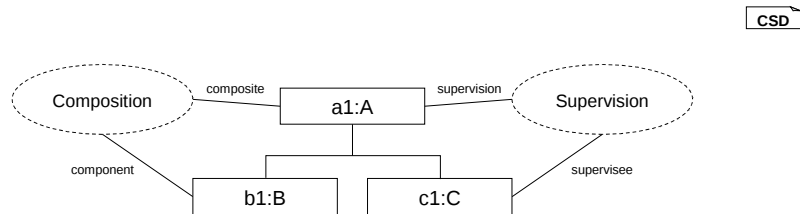
CSD

Composition — composite — a1:A — supervision — Supervision

component — b1:B   c1:C — supervisee

Fig. 3. A hierarchy of component runtime instances

This architecture style has several beneficial implications.

– The execution semantics of the software do not depend on the actual physical distribution of components. Interactions are always and completely described

by discrete, passed messages and always asynchronous in nature. Hence, components can be regarded as distributed by default.

– Component executions are self-contained. Every component has its own thread of control. Blocking operations or component failures do not influence other components directly. Failure recovery can be implemented within the local scope of the failed component.
– Components encapsulate a state and are, hence, in combination capable of representing the overall state of a continuously running system.
– Components do not need to have direct knowledge about the receivers their sent messages. They only need to know the message channels they are connected to. Channels, in turn, are implemented by a communication middleware that is independent from individual components. This middleware can deliver messages dynamically to different runtime instances of replicating receivers using different communication patterns (e.g. message queues) and technologies (e.g. different protocols and message formats). In this way, the middleware can implement several load balancing and scaling strategies and integrate heterogeneous infrastructures.
– Component implementations are technology-agnostic as their execution and interaction semantics are defined homogeneously and independent from concrete target technologies. Such components are easier to port between different infrastructures and easier to test and simulate.

This architecture style bears much resemblance to actor-based systems [21] but differs in some aspects. First, components may have not just one but many buffers for incoming messages. Second, these buffers are strictly typed. Third, communication channels are statically defined by the software architecture model and cannot be altered at runtime. Fourth, components do not control the instantiation of other components. Instead, the software architecture is statically defined by a given architecture model and automatically instantiated by the generated framework according to that model. Only the number of replicating runtime instances is dynamic.

4 Architecture Modeling

At the center of our language family is the cloud architecture description language (clADL). This language follows the *components and connectors* paradigm. It describes the structure of a software system in terms of system parts (components) and their mutual relationships (interfaces and connectors). [22] The clADL is derived from MontiArc, an architecture description language for distributed, interactive systems. [9] As MontiArc, the clADL is based on the semantics defined by the FOCUS method. [14] More precisely, our ADL is an extension of MontiArc that adds cloud software specific syntax and semantics.

The clADL describes cloud software architectures in terms of interacting *components*. A component is a distinct system part that implements a certain

function. Components communicate with other components by exchanging *messages* as atomic units of information. Messages are exchanged via explicit *connectors* between components. Thereby, components and connections describe a network that represents the system architecture. In this network, components act autonomously and exchange messages asynchronously without an subordinately imposed control flow. This semantics applies to a *logical* point of view and does not enforce a particular *technical* realization.

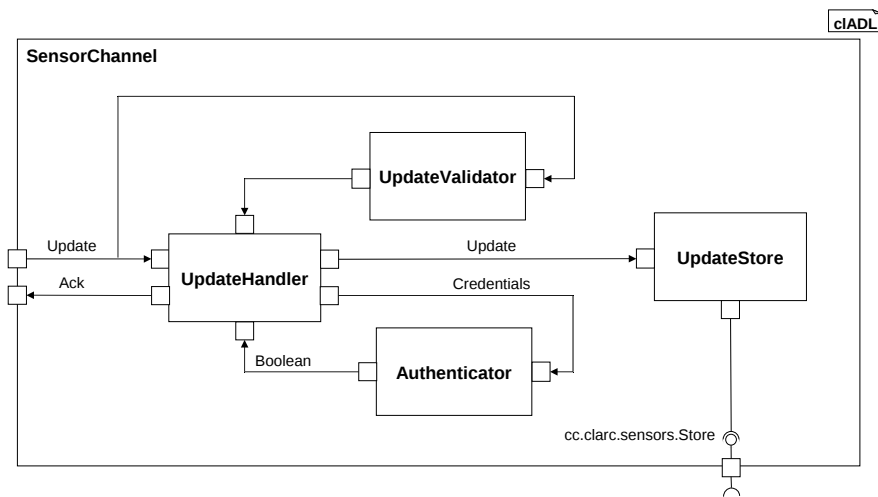Figure 4 shows the architecture of a simple cloud service in a graphical representation.

Fig. 4. The software architecture of a simple cloud service.

This service is modeled as a decomposed component named `SensorChannel`. It receives streams of sensor data represented by messages of type `Update` and acknowledges them by responding with `Ack` messages. The component is internally decomposed into four subcomponents. The `UpdateHandler` receives all incoming `Update` messages from its `SensorChannel` parent component, interacts with the `Authenticator` and the `UpdateValidator` to analyze the update, sends valid updates to the `UpdateStore` and finally sends an acknowledgement to the `SensorChannel`'s respective outgoing port. The `Authenticator` checks whether the received Update has valid credentials. The `UpdateValidator` checks the received data for its validity. The `UpdateStore` uses a *service port* to write the update to a database provided by the service's underlying platform.

A component is syntactically defined by its name and its *interface*. Its interface is defined as a set of *ports*. Ports are the end points of *connections* between components and can either (as incoming ports) send or (as outgoing ports) receive messages. A component may be *decomposed* into further subcomponents

and is, moreover, denoted as their parent component. Hence, components can again be understood as systems on their own. In fact, the system as a whole can be described as one single root composed component. Accordingly, we call components that are not decomposed into further subcomponents atomic components.

The semantics of a component defined by its externally observable (black box) *behavior* that is given by the relation between the sequence of received messages and the sequence of sent messages. The behavior of atomic components is given by a behavioral specification. It may be described in various ways, for instance, in declarative logic or functional/imperative programming languages. That specification includes the notion of an internal component *state* that changes depending on the sequence of incoming messages and implies the sequence of outgoing messages. The behavioral specification of composed components is inductively given by the aggregated behavioral specifications of its subcomponents and the topology of their connections.

Messages are syntactically defined by a name and a *type* that determines the kind of that specifies the kind of information they carry. Accordingly, ports reference a message type that determines the kind of messages they can communicate. Message types come with different internal syntactical structures that denote the basic structure of their carried information. They may be primitive types (e.g. a number or a string) or complex types that are composed of primitive types in a certain syntactic structure. Message types can be defined through external languages, for instance, Java and UML/P class diagrams. [23]

Connections syntactically connect exactly one outgoing port with one incoming port. Thus, connections haven an implicit direction in which messages can be communicated.

4.1 Replication

Subcomponents can be modeled as *replicating components*. Usually, the declaration of a component within the context of a parent component semantically implies the *fixed* existence of a *single* runtime instance of that component in the context of its parent component. In contrast, the declaration of a replicating component implies a *variable* number of *multiple* instances. This notion allows us to describe quantitative system scalability in terms of dynamic replication of system parts. That means, the number of instances of replicating components may increase or decrease dependent on specific circumstances. Thereby, the system dynamically adapts to increasing or decreasing load demands.

By implication, replicating components may be dynamically created and destroyed. To represent this, every replicating component maintains a *lifecycle* state that corresponds to a lifecycle model. Basically, a components lifecycle determines if the component is idle and therefore a candidate for destruction or if it is busy and therefore protected from destruction.

4.2 Contexts

The semantics of channels where the receiver is a replicating component prototype are not in itself fully specified. The model does not define the mechanism that selects the replicating component's concrete runtime instance as the receiver of a given message. The semantics of channels only define the constraint that individual messages are only received by one receiver.

However, in many cases the concrete receiver of a message matters. A common real-world example are web systems that handle multiple user sessions. Runtime component states might be associated to such user sessions. Hence, interactions between such components should happen between those instances that have a state associated to that user session.

Contexts are a mechanism to resolve the ambiguities of receiver selection in such scenarios. A context is a type for *context tokens* and is declared in the scope of a component type. Context tokens are markers that can be assigned to components and messages.

Context tokens are assigned to messages that are sent through *context gates* of the respective context. Context gates can be defined on connectors and ports. Context gates can *open a context* by assigning a new token of that context to the message passing the gate. Furthermore, they can *close a context* by removing all tokens of that context from the message passing the gate. In addition, every outgoing port of a composite component implicitly closes all contexts defined in that component on messages passing this port.

The tokens assigned to messages that pass context gates serve as an identifier similar to session IDs in web systems. When messages with context tokens are received by a component, this component is also implicitly associated with these tokens.

4.3 Service Interfaces

Components may declare *service ports*. In contrast to other ports, service ports are not endpoints of message channels but represent *service interfaces* that provide *operations* that can be called on other software or by other software. Service ports can be required by a component (e.g. to call operations on the runtime infrastructure the component is being executed) or provided by a component, hence allowing other software running in the same runtime to call operations on the component.

5 Infrastructure and Deployment Modeling

Models of the clADL describe logical software architectures. They omit the details of its technical realization and physical distribution, that is, its complete *system architecture*. For a complete description of the actual system, additional information is necessary, in particular, information about the technical infrastructure it will run on (e.g. a Java EE Application Server), its technical configuration (e.g. the URL it will be bound to) and the concrete mapping of our

software architecture onto this infrastructure. Alltogether, this information constitutes the *deployment* of an architecture implementation.

A deployment is described by *infrastructure models* and *mapping models*. Infrastructure models describe the technical infrastructure on which an archtecture implementation will run. For instance, an infrastructure can consist of application servers, and databases. Mapping models relate components in the software architecture to elements of the infrastructure architecture. For instance, a mapping model could specify that selected ports are accessible via a RESTful interface.

5.1   Target Description Language

The *Target Description Language* describes *physical infrastructure architectures* onto which software can be deployed for execution.

Targets are the essential building blocks of infrastructure architectures. Every target represents a discrete part of an infrastructure. Targets are called targets because components of the software architecture can be targeted at them for deployment.

Targets are defined by *target types*. Target types are defined in individual target models which in combination form the overall infrastructure architecture model. Target types can extend other target types. Thereby, they inherit all the structural and semantic properties of the other target type and enter a topological "is a" relationship with it.

The language defines a fixed set of target kinds.

- *Locations* represent physical locations (e.g. certain data centers or regions with particular legislations).
- *Resources* represent physical resources of information technology like computation and storage (e.g. hardware servers, virtual servers, storage systems).
- *Runtimes* represent containers for the execution of software (e.g. virtual machines, application servers, database management systems).
- *Artifacts* represent software documents or archives containing software documents (e.g. Java Archives, Web Archives, compiled binaries).
- *Modules* represent grouped software modules with shared properties (e.g. software parts executed in the same security sandbox).
- *Endpoints* represent resources for communication (e.g. web services, message queues).

Targets can contain subtargets which are target prototypes of a particular target type. Possible containments are constraint, for instance, a location can contain resources but a resource cannot contain locations. Subtargets can be declared as replicating subtargets and thus allow for multiple runtime instances of that subtarget.

Figure 5 shows an example of a hierarchy of nested target prototypes. In this example, a resource `Server` contains two other resources `VM_A` and `VM_B`

(i.e. Virtual Machine). VM_A contains a runtime `ApplicationServer` which contains an artifact `WebArchive` which, again, contains two modules `Logic` and `UserManagement`. VM_B contains a runtime `MySQLServer` which contains an endpoint `MySQLAccess`. The `Server` is contained in an implicit location of the general target type `Location`.
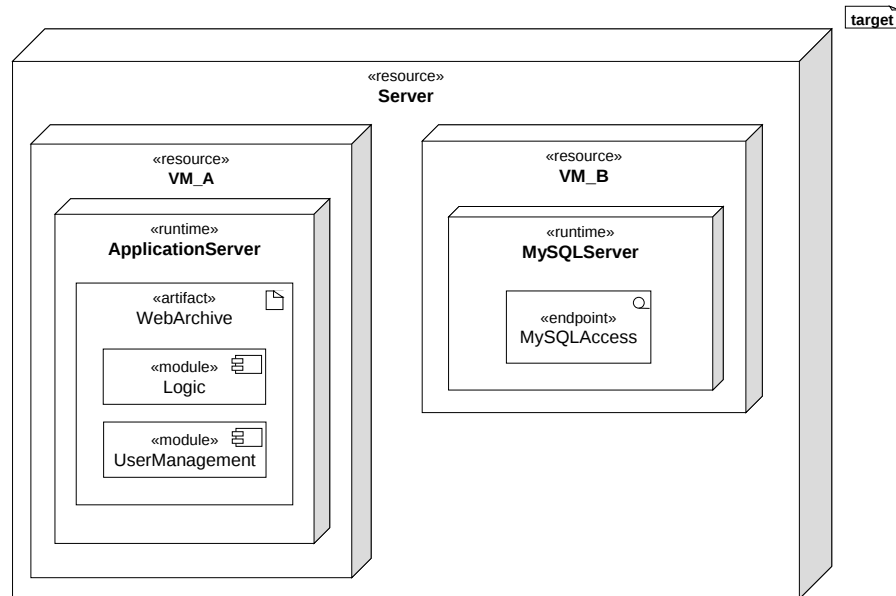

Fig. 5. An example for a hierarchy of targets

Target types may declare *target properties*. Target properties are variables that are assigned to string values in target prototypes. They can, for instance, describe TCP port numbers of services or identify the operating system of a server. Properties can be assigned to values in subtarget declarations.

5.2 Mapping Description Language

The *Mapping Description Language* defines *deployments* of a software architecture onto an infrastructure architecture.

Mappings are collections of concrete mappings between components and targets. Hence, a mapping relates a model of a logical software architecture defined by clADL models to a model of a physical infrastructure model defined by TDL models. Mapping declarations map a referenced component as well as all otherwise unmapped subcomponents of the hierarchy it defines to the referenced target.
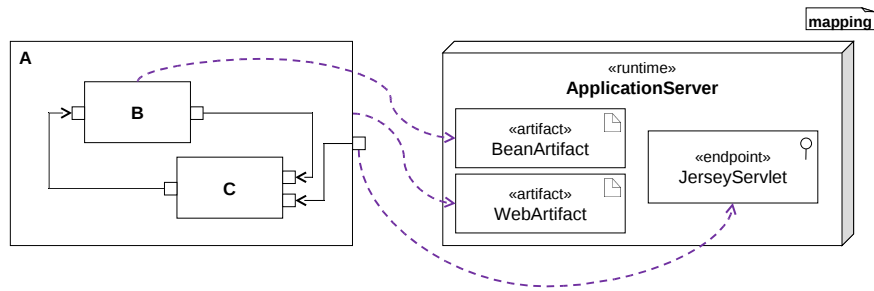
Fig. 6. A mapping model with two mappings

Figure 6 shows an example of a software architecture with a component `A` being decomposed into subcomponents `B` and `C` put next to an infrastructure architecture consisting of a runtime `ApplicationServer`, two artifacts `BeanArtifact` and `WebArtifact` and an endpoint `JerseyServlet`.

6 Model-based Testing

The *Architecture Scenario Description Language* describes exemplary interaction scenarios in concrete software architectures. The *Architecture Test Suite Definition Language* defines test setups of software architectures and corresponding scenarios.

A scenario describes a *valid*, chronologically arranged, partially ordered set of interactions in a software architecture. Interactions are described as messages passed between component prototypes. Scenario descriptions bear resemblance to *Message Sequence Charts* [24]. Thus, scenarios are partial *protocol definitions* [25]

Figure 7 shows an interaction between the subcomponents of the `SensorChannel` component introduced in figure 4. This representation is similar to UML sequence diagrams. Components are depicted with ports and timelines while interactions are depicted as arrows between these timelines.

Scenarios can serve as specifications for model-based tests. From a scenario model, the expected behavior of each participating component can be derived. This behavior can be used (a) as a reference against which components under test can be evaluated and (b) as a basis for mocking components with whom components under test interact.

7 Language Execution

The model-based SDK for the SensorCloud consists of several tools that combine source code of traditional programming languages and models of the clArc language family.
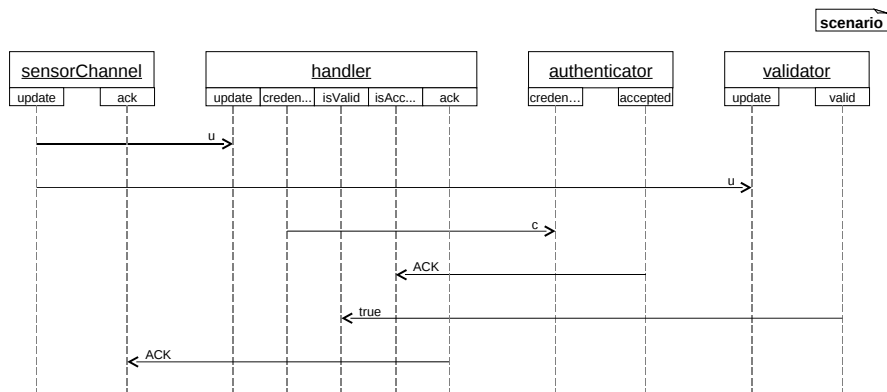
Fig. 7. A Scenario of Interactions inside SensorChannel

At the core is a *modular code generator* that is composed of many generator modules, each being responsible for different aspects of the generated code (e.g. protocols, resource management). This code generator generates a deployment-specific service framework based on clADL, TDL and MDL models. This framework is customized with handwritten code and integrates with the SensorCloud's platform APIs. The result is a complete implementation of a SensorCloud service. The use of models lifts the abstraction of this implementation and makes it agnostic of the actual SensorCloud's API. Platform-specific code is strictly left to the code generator. In this way, the platform can evolve without affecting the implementation of its services.

In addition, a test-specific code generator can generate a functionally equivalent variant of a service's implementation that can be executed locally. In this way, the service can be functionally tested without the need for a testing infrastructure. Concrete test cases can be generated from scenario models.

8 Conclusion

In this paper we described a model-based SDK based on the clArc toolkit for developing cloud services for the SensorCloud platform. The SDK is in ongoing development and will enter its evaluation phase in the third and final year of the SensorCloud's research and development schedule.

References

1. Lee, E.A.: Cyber-Physical Systems - Are Computing Foundations Adequate? October (2006)
2. Iwai, A., Aoyama, M.: Automotive Cloud Service Systems Based on Service-Oriented Architecture and Its Evaluation. In: 2011 IEEE 4th International Conference on Cloud Computing, IEEE (July 2011) 638–645

3. Harper, R., ed.: The Connected Home: The Future of Domestic Life. Springer London, London (2011)
4. Haller, P., Odersky, M.: Scala Actors: Unifying thread-based and event-based programming. Theoretical Computer Science **410**(2-3) (February 2009) 202–220
5. : SensorCloud
6. Krahn, H.: MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering. Dissertation, RWTH Aachen University (2010)
7. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: A Framework for Compositional Development of Domain Specific Languages. International Journal on Software Tools for Technology Transfer **12**(5) (March 2010) 353–372
8. Department of Software Engineering at RWTH Aachen University: MontiCore
9. Haber, A., Ringert, J.O., Rumpe, B.: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical report, RWTH Aachen University, Aachen (2012)
10. Haber, A., Ringert, J.O., Rumpe, B.: Towards Architectural Programming of Embedded Systems. In: Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VI. (2010)
11. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software Architecture: Foundations, Theory, and Practice. (January 2009)
12. Armbrust, M., Stoica, I., Zaharia, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A.: A View of Cloud Computing. Communications of the ACM **53**(4) (April 2010) 50
13. Selic, B., Gullekson, G., Ward, P.T.: Real-Time Object-Oriented Modeling. Wiley professional computing. Wiley (1994)
14. Broy, M., Stø len, K.: Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement. Springer Verlag Heidelberg (2001)
15. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems - Specification. Springer (1992)
16. Rumpe, B.: Modellierung mit UML: Sprache, Konzepte und Methodik. Xpert.press. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
17. Rumpe, B.: Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring. Xpert.press. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
18. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. 2nd edn. Addison-Wesley Professional (May 2005)
19. Friedenthal, S., Moore, A., Steiner, R.: A Practical Guide to SysML: Systems Modeling Language. (July 2008)
20. Engels, G., Whittle, J.: Ten years of software and systems modeling. Software & Systems Modeling **11**(4) (September 2012) 467–470
21. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. Dissertation, Massachusetts Institute of Technology (December 1986)
22. Medvidovic, N., Taylor, R.N.R.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering **26**(1) (2000) 70–93
23. Schindler, M.: Eine Werkzeuginfrastruktur zur Agilen Entwicklung mit der UML/P. Dissertation, RWTH Aachen University (2011)
24. Krüger, I.: Distributed System Design with Message Sequence Charts. PhD thesis, Technische Universität München (2000)
25. Selic, B.: Protocols and Ports: Reusable Inter-Object Behavior Patterns. In: ISORC. (1999) 332–339