A Pattern-based Foundation for Language-Driven Software Engineering

– Relevant Publications –

Tim Reichert

January 2011

List of Relevant Publications

The following publications of the author resulted from the PhD research and are, therefore, included in this appendix.

Tim Reichert, Edmund Klaus, Wolfgang Schoch, Ansgar Meroth, and Dominikus Herzberg. A Language for Advanced Protocol Analysis in Automotive Networks. In *Proceedings of the 30th International Conference on Software Enginieering*, ICSE '08, pages 593-602, New York, NY, USA, 2008. ACM.

Tim Reichert and Dominikus Herzberg. A Domain Specific Language for Uncovering Abstract Protocols and Testing Message Scenarios. In *Proceedings of Software Engineering 2008 (Workshops)*, pages 427-430, 2008.

Tim Reichert and Dominikus Herzberg. Teaching Language-Driven Software Engineering. In *International Conference of Education, Research and Innovation (ICERI)*, Madrid, Spain, November 2009.

Dominikus Herzberg and Tim Reichert. Concatenative Programming - An Overlooked Paradigm in Functional Programming. In *ICSOFT 2009 - Proceedings of the 4th International Conference on Software and Data Technologies, Volume 1, Sofia, Bulgaria, July 26-29, 2009*, pages 257-263, INSTICC Press, 2009.

Dominikus Herzberg and Tim Reichert. Software Engineering for Telecommunication Systems. In *Benjamin W. Wah et al. (Eds): Encyclopedia of Computer Science and Engineering*. Wiley, 2009.

Dominikus Herzberg, Tim Reichert, and Nick Rossiter. Towards Modeling Language Interoperability – Getting Meta-Level Architectures Right. *Forschungsbericht der Hochschule Heilbronn 2008/2009*, 2008.

A Language for Advanced Protocol Analysis in Automotive Networks

Tim Reichert
School of Computing, Engineering & Inf. Sciences
Northumbria University
Newcastle upon Tyne, NE2 1XE, United Kingdom
tim.reichert@unn.ac.uk

Edmund Klaus
Department of Software Engineering
Heilbronn University
74081 Heilbronn, Germany
eklaus@stud.hs-heilbronn.de

Wolfgang Schoch
Department of Software Engineering
Heilbronn University
74081 Heilbronn, Germany
wschoch@stud.hs-heilbronn.de

Ansgar Meroth
Automotive Competence Center
Heilbronn University
74081 Heilbronn, Germany
meroth@hs-heilbronn.de

Dominikus Herzberg
Department of Software Engineering
Heilbronn University
74081 Heilbronn, Germany
herzberg@hs-heilbronn.de

ABSTRACT

The increased use and interconnection of electronic components in automobiles has made communication behavior in automotive networks drastically more complex. Both communication designs at application level and complex communication scenarios are often under-specified or out of scope of existing analysis techniques. We extend traditional protocol analyzers in order to capture communication at the level of abstraction that reflects application design and show that the same technique can be used to specify, monitor and test complex scenarios. We present CFR (Channel Filter Rule) models, a novel approach for the specification of analyzers and a domain-specific language that implements this approach. From CFR models, we can fully generate powerful analyzers that extract design intentions, abstract protocol layers and even complex scenarios from low level communication data. We show that three basic concepts (channels, filters and rules) are sufficient to build such powerful analyzers and identify possible areas of application.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

verification, languages

ICSE'08, May 10–18, 2008, Leipzig, Germany.

Keywords

protocol analysis, protocol specifications, automotive systems engineering

1. INTRODUCTION

The amount of electronics and software in automobiles has been increasing rapidly over the last two decades. Modern vehicles contain a growing amount of Electronic Control Units (ECUs) that are in charge of different subsystems, ranging from motor control to entertainment [20]. Bus systems connect these distributed ECUs into communication networks and thus allow previously autonomous subsystems to exchange information in order to provide more advanced functionality. Coping with the system complexity that results from increasingly sophisticated and more and more interconnected subsystems poses one of the great challenges for the automotive industry today. Problems caused by faulty electronics and/or software are quickly becoming the number one reason for car defects. Electronics and software related product recalls cost car manufacturers heavily in money and reputation. In addition to that, crucial subsystems such as breaks, steering and airbags require utmost reliability from software and electronics [6].

Figure 1 shows a typical communication context in a modern car. Subsystems in distinct domains such as drive train or multimedia are connected by specialized bus technologies, e.g. CAN (Controller Area Network) [1] or MOST (Media Oriented Systems Transport) [4], to form subnetworks. Communication between components in different subnetworks is established through gateways that bridge technological differences. To provide an example of advanced functionality that requires such communication contexts, consider the implementation of an adaptive break light. Becoming a standard in today's cars, an adaptive break light warns following drivers, for example through rapid blinking, when an emergency brake maneuver is executed. To correctly implement such functionality, data from different subsystems in different subnetworks has to be exchanged and compared.

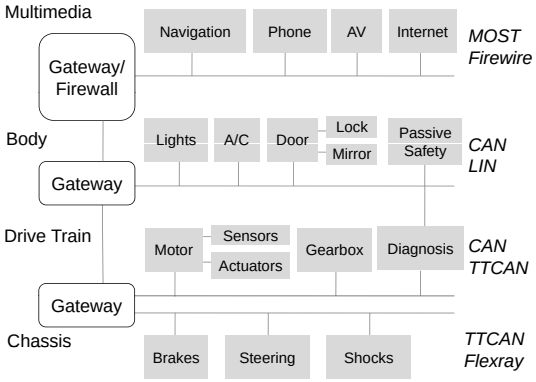

Figure 1: Typical communication context in a modern automotive network

This is done by sending messages over the communication bus. Among the relevant data might be the speed of the car, possibly provided by the instrument cluster, and the reaction of the anti-lock brakes, provided by the break system. In addition to that, the light system has to be instructed to blink rapidly.

A well known technique for specifying communication behavior between distributed components that originated in the telecommunications domain is definition through protocols and protocol stacks [5]. Protocols define the format and order of messages that can be exchanged between components. ECUs then have to implement these protocols in order to successfully communicate with other ECUs. *Protocol analyzers* can greatly help both testing and understanding systems that communicate based on protocols. They are components that either observe the network traffic or stimulate complete applications in a more or less static manner. A protocol analyzer scans messages on the network, decodes the messages, i.e. it changes the binary coded messages in accordance with the protocol standard into a reader-friendly textual or graphical representation, and offers more or less comfortable monitoring functions.

Protocol analyzers are effective tools when communication behavior is explicitly specified or even standardized through protocols. However, in today's complex systems, not all communication behavior is specified in protocols. While protocols usually exist for lower levels of communication, application specific behavior that involves high-level communication patterns is often not formally specified. Instead, communication intentions are often implicit, undocumented or even evolved during implementation – with the design intention hidden in the source code. A simple example of a communication pattern that is usually not specified by a protocol is the implementation of notifications on top of a request/reply protocol. We presented this example in Section 2. In addition to the aforementioned problems, complex communication scenarios such as the one that occurs in the context of the adaptive break light, where a sudden event triggers a chain of messages between different components in different subnetworks, are often under-specified or not specified at all. This poses a severe problem for system comprehension and testing approaches in general and protocol analysis in particular, as communication not specified in a protocol is shown at the wrong level of abstraction during analysis.

In our work, we tackle the problems associated with underspecification of communication behavior between ECUs with a novel approach for the specification of analyzers for protocols and scenarios. Three basic concepts (channels, filters and rules) are sufficient to build such analyzers. Using a Domain Specific Language (DSL) that implements our approach, we can specify communication behavior at different levels of abstraction. The result are analyzers that can be the basis for automated testing as well as system comprehension and re-engineering of underspecified systems. In addition to that, we propose our language as a means to formally specify complex scenarios that are often not adequately captured using existing formalisms. The novelty of our approach is thus its use of a single formalism to capture both communication design intentions and communication scenarios.

The remainder of this paper is organized as follows. Section 2 provides an overview of basic concepts including abstract protocols and complex scenarios; examples that we use throughout the paper are introduced there. In Section 3 we present our approach in detail and show how it can be applied to protocols and scenarios and how it can make a contribution at different stages of the software development process. Section 4 introduces a domain-specific language that implements our approach. We give an overview of concrete and abstract syntax, provide examples and discuss implementation issues. Related work is explored in Section 5. We conclude in Section 6 with a reflection on our results and lay out possible directions for future research.

2. THE CHALLENGE: PROTOCOLS AND SCENARIOS

As described in the previous section, the different ECUs and buses in modern cars constitute complex, distributed systems of communicating entities [23]. Such systems have been the subject of much research in the telecommunication and computer networking domain and many of the techniques in these domains have been applied to networks in cars as well, for example layering. Layering is a fundamental technique for designing distributed systems. It is a method to provide in a stepwise fashion higher-level service to users on the layer above, and to separate levels of services by precisely defined interfaces. This overall design principle is reflected by the use of protocol stacks where higher layer network services rely on lower layer services until a physical layer is reached [14].

2.1 Abstract Protocols

Protocol analyzers can monitor system communication on different layers of a protocol stack [11]. However, the parts of the communication design that are not specified as part of a protocol are invisible during analysis. Instead, only the effect that the implementation of a design has on a lower protocol layer can be monitored. This is severely limiting the benefits of protocol analysis, as communication on the application level is – in contrast to lower level communication – typically not captured by protocols. Our solution is to extend the scope of protocol analysis to capture higher levels of communication design. In the following, we provide

an example for the kind of communication behavior that is typically underspecified.

Consider a communication protocol that specifies the communication between a client and a server. The protocol is based on request and reply messages. A valid communication always consists of a request by the client and a reply by the server. This entails, that the server cannot contact clients directly without receiving a prior request. Now consider an application where the client needs information about a particular state change that happens on a server. The most intuitive solution for an implementation would be, that the server sends a notification message to the client as soon as the state change occurs. However, when client and server communicate using the request/reply protocol just described, this seems impossible.

One solution to this problem is to add a notification service to both the client and the server. The service encapsulates the request/reply-based communication and offers an interface that provides notification capabilities to the rest of the application. To achieve this, the service implements a polling strategy. That is, the service on the client periodically sends requests to the server. Upon receiving a request, the service on the server can reply and transmit either a negative acknowledgment or deliver the notification message. Thus, for applications using the notification service, communication is based on notifications, not on requests and replies.

The left sequence chart in Figure 2 shows how a possible communication involving two notifications might look like when traced with a protocol analyzer for the request/reply protocol. The communication consists of all request/reply messages between client and server. However, the notification service layer that was introduced in the software is invisible. That means, using the protocol analyzer, it is impossible to view the system behavior at a level of abstraction where a server sends notifications to a client, although that is the way application developers think about the communication behavior. Also, viewing communication at this higher level is what is typically needed for reasoning about application behavior. For instance, we might be interested in what notifications are sent from the server and how the client reacts – independent of how the notification mechanism is implemented. That is, we wish to monitor communication in terms of application level abstractions.

In order to make the communication at this level of abstraction accessible using a protocol analyzer, the analyzer has to understand the additional service level introduced by an application. Once uncovered, this new level of abstraction can then be used to analyze reactions of components to notifications without having to consider sequences of requests and replies. We can define the higher level of abstraction as a protocol and relate it to communication on a lower level. For our example, this means that we first define a very simple notification protocol, where valid communication consists only of a server sending a notification message to a client. We then relate this protocol to the request/reply protocol by describing how sequences of request and reply messages relate to notification messages: a request message from a client followed by a positive reply from the server to the same client should be interpreted as a notification message from server to client. The right sequence chart in Figure 2 shows how the lower level communication depicted on the left can be rendered using the newly defined notification protocol layer. The notification protocol captures the design intention of the application designer. We call such protocols *abstract protocol* that constitute abstract layers of service.

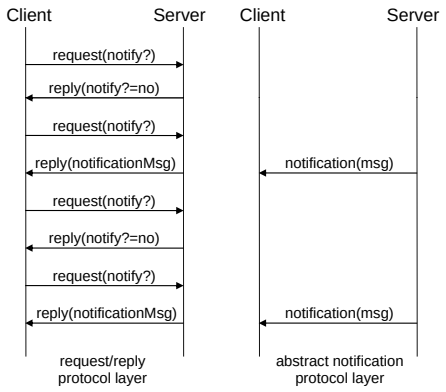

Figure 2: Protocol abstraction: Defining an abstract notification layer on top of the request/reply layer

We can stack abstract protocols in the same way regular protocols can be stacked. That is, we can define an abstract protocol on top of another abstract protocol. Indeed, even the request/reply protocol in our example could be an abstract protocol based on a yet lower protocol layer. This approach is open ended and can be used to build levels of abstraction in a stepwise fashion, thereby mirroring the abstraction process in the application itself. In order to teach a protocol analyzer abstract protocol layers, the analyzer must be programmable. That is, there needs to be a mechanism for describing stacks of protocol layers in a way that can be used to instruct the analyzer how to extract each layer from a lower layer. In Section 3 and 4, we will introduce CFR models as a way to define protocol stacks in such a way.

Abstract notification protocols such as the one described in this section are widely used in automotive networking. The MOST network uses request/reply mechanisms for the communication between clients, e.g. Human Machine Interfaces, and servers, e.g. a radio tuner [18]. For example, to ask the tuner for the current frequency, a message is sent with the content AMFMTuner.ATFrequence.Get() and the reply is sent back with the current value. On a notification layer, it is possible to subscribe a client to be notified if a property such as ATFrequency changes. CAN, on the other hand, is purely built upon broadcast of signal values. However, the Remote Transmission Request Bit that is part of the protocol header can be used to stimulate the transmission of a signal value. This is in effect a request/reply mechanism.

2.2 Complex Scenarios

Use cases are a widely-used technique for describing system functionality from a user-centric point of view. We facilitate use cases to identify scenarios in automotive systems. As an example for a use case, consider the adaptive brake light discussed in Section 1. If the user – a driver in this case – performs a braking maneuver that activates ABS (Antilock Breaking System), the break light might show some reaction that deviates from its normal behavior, depending on

speed, strength of breaking and the reaction of the driving assistance systems. The use case can be described as: *The driver performs an ABS braking maneuver.* At the level of ECUs, we derive the following – much simplified – scenario from the use case: The braking system registers blocking of tires and activates the anti-lock brakes. It notifies other components listening on the bus of the anti-lock brake activity. The control unit that controls the adaptive brake light registers the activity of the ABS and requests the speed of the car from, for example, the instrument cluster and decides to instruct the brake light system to blink. A second scenario associated with the same use case might be similar to the one just described, only that the speed is lower than a certain threshold and the adaptive brake light thus does not interfere with the normal operation of the brake light.

While the above description of scenarios is rather imprecise, scenarios can be precisely described at the level of the actual messages exchanged between components over the communication bus. This can be done at different protocol layers by associating a sequence of relevant messages with a scenario. By monitoring bus activity it is then possible to identify scenarios by looking for certain message patterns in the communication stream. We will elaborate on this in the following section and proceed instead with a description of the challenges posed by complex scenarios.

Complex scenarios involve communication between several different ECUs, possibly in different subnetworks. They pose a challenge for both specification and testing of ECUs. Typically, they are not built by only determined message sequences within the network. Vehicle communications systems are heavily impacted by stochastic triggers arising from the system's environment. For example, mobile phone calls may appear any time as well as warnings and recommendations from driver information and assistance systems. Even for moderate scenarios, the amount of communication to process may rise to an extent that makes monitoring and verification of scenarios a non-trivial task. Complex scenarios may cause peculiar errors e.g. overflow under stress conditions which may lead to the breakdown of a certain resource or the whole network. On the other hand, component and system developers may diverge in their assumptions of the system's macroscopic behavior. They are likely to anticipate different reactions of the components, depending on the system context taken into account.

While network standards often precisely define lower levels of communication and trivial scenarios, complex scenarios are often not specified properly. One reason for this is that the combination of all possible sequences of a given protocol surmounts the scope of a protocol standard. Especially for safety critical systems such as vehicles, this is not acceptable and calls for more complete approaches to specification and testing. In the following section we will show that our approach to protocol analysis can be the foundation for such approaches.

3. THE SOLUTION: CHANNELS, FILTERS AND RULES

In the previous section we introduced abstract protocols and complex scenarios as two main challenges when analyzing communication messages in a distributed system. Subsequently, we describe our approach in detail and indicate how we address these challenges.

3.1 Defining Abstract Protocol Layers using CFR Models

We already observed that in order to be able to introduce abstract protocol layers, we need some way to formally describe these new layers. As we have explained through the request/reply example, it is common to stack protocols on other protocols with the protocol layers interfacing adjacent protocol layers only. In this way, a protocol layer can be described by relating its communication patterns to the communication patterns on the next lower layer. Indeed, that is how we described the abstract notification protocol in Section 2: A request for notification message from a client followed by a positive reply from the server to the same client should be rendered on the next higher layer as a notification message from server to client. If we look more carefully at this natural language description of the protocol layer, we see that the following information is provided:

- Certain messages on the lower protocol layer are identified.
- The sequence in which these messages occur is given.
- A mapping of this message sequence to a message on the higher level protocol layer is defined.

We now present a conceptual framework that allows us to describe protocol layers in a machine-processable form. This framework is the foundation of the language we describe in the next section. Our framework is based on the following three basic concepts:

- A *channel* is a medium which transports messages. The messages obey a concrete or abstract protocol. A protocol consists of messages; it defines their syntax (i.e. the construction) and optionally a set of sensible message sequences.
- A *filter* is a passive and stateless unit that redirects messages from an incoming channel to one of two outgoing channels. Depending on a given pattern, the filter determines which message is to be delivered to which outgoing channel. Since both the input and output of a filter are messages, the output of a filter can be used as the input of another filter. Filters might be put in parallel or in sequence and may build up a filter-pipe-architecture. To reduce complexity, filter cascades can be encapsulated into an encapsulating filter. Filters are passive elements as they do not modify or create messages.
- A *rule* is an active and stateful message processing unit that consumes messages from one or more incoming channels and returns messages to one or more outgoing channels. In opposition to filters, a rule can modify incoming messages or create new messages. To do that, a rule contains defined logic in form of a state machine or a set of state machines with a selector function. Similar to filters, rules can be used to encapsulate other rules, channels or filters. Similar to filters, the output of one rule might be the output of another rule.

Protocols and their relationships to the next lower protocol layer can then be defined by combining these elements

using connectors or through containment. Such CFR models can be interpreted in two similar but distinct ways: As a specification that defines a higher protocol layer through a lower protocol layer, or as a specification for a protocol analyzer that can render communication on a higher layer by observing communication on a lower layer.

We represent the inputs and outputs of channels, filters and rules by input and output pins. An input pin can be connected to an output pin using a connector. The message flow is from output to input pins. Channels and rules have an arbitrary number of input and output pins, while filters have exactly one input and two output pins. We use regular expressions to define the selection criteria in filters. State machines are used to define the memory of rules. In some cases involving multiple components, a single state machine is not sufficient. In this case we use a set of state machines with a selector function defined over that set. This function then maps sender and receiver of a message to a distinct state machine. For reasons of readability, we require that filters might not be directly connected to rules and vice versa. This constraint does not reduce the expressive power of CFR models, as placing an otherwise unused channel between a filter and a rule is equivalent to connecting the two elements directly. An example of how CFR models can be used to define abstract protocols is described in the following. We define a CFR model to specify the abstract notification protocol from the request/reply example.

Let C_{rr} be a channel over which messages defined by a protocol rr – the request/reply protocol – are sent. These messages include request and reply messages related to notifications, but also other request and reply messages not related to notifications. Let C_{not} be a channel over which notifications messages are sent. Our aim is now to define a CFR model that describes how the message stream on C_{rr} can be transformed into a message stream on C_{not}. The following setup achieves this: The output pin of channel C_{rr} is connected to the input pin of a Filter F_{not}. The positive output pin of F_{not} is connected to a channel C_{rrf}. F_{not} is configured in such a way, that it sends all request/reply messages concerned with notifications to its positive output and all other messages to its negative output. Now, the communication on C_{rrf} consists only of the request/replies concerned with notifications and no other messages. The output pin of C_{rrf} is connected to the input pin of a rule R_{not}. The first output pin of the rule is connected to the input pin of C_{not}. The rule is configured in the following way: When a requests message from a client occurs, the rule memorizes the request. When a positive reply that fits a previously memorized request is received, a notification message wrapping the reply is created and sent through the output pin onto C_{not}. The messages on C_{not} are now notification messages sent from servers to clients. Omitting the filter and rule logic for the moment, the CFR model can be described in a textual notation as follows, whereby the $\rightarrow$ can be read as *connected to*, $R_{not}(out, 1)$ is the first output channel of R_{not}.

Channel: C_{rr}, C_{rrf}, C_{not}
Filter: F_{not}
Rule: R_{not}
$C_{rr}(out) \rightarrow F_{not}(in)$
$F_{not}(positive) \rightarrow C_{rrf}(in)$
$C_{rrf}(out) \rightarrow R_{not}(in)$
$R_{not}(out, 1) \rightarrow C_{not}(in)$

We will return to this example in Section 4 and show it in the notation of our visual language in Figure 5. Our simple example already indicates that the restriction on combining filters and rules enforces more readable designs. C_{rrf} is introduced as a helper because the output pin of F_{not} cannot be directly connected to the input pin of R_{not}. We thus have a way of describing intermediate steps in the definition of protocols. We can even view intermediate channels as being based on separate sub-protocols and the protocol definition itself as layering and combination of these sub-protocols. It should be noted, that in the example model and in general, both the given lower layer and the newly defined higher layer communication is represented as a channel transporting messages adhering to some protocol. This means that we can use the newly defined channel to define further channels and thus can use our approach for the stepwise definition of abstraction layers. In our example, it is transparent whether the protocol used by C_{rr} is a concrete protocol or an abstract protocol defined by another CFR model.

3.2 Specifying and Monitoring Complex Scenarios

As described in the previous subsection, the way in which we define a new protocol on a layer L_i is to identify certain communication patterns on the next lower protocol layer L_{i-1}. We then describe the abstraction of this pattern as messages on layer L_i. While the identification of patterns is done using cascades of filters and rules, the abstraction step is performed through the message generation capability of rules. If we view a scenario as a sequence of messages, the commonalities between abstract protocols and scenarios become obvious and it is apparent that our approach for specifying protocol layers can directly be mapped to the specification of scenarios. This can be done by describing a scenario as a specialized high-level abstract protocol. This protocol is defined by specifying a message and relating it to a unique sequence of messages at the next lower layer.

It is important to understand that the messages of a scenario do not necessarily obey the same protocol. In fact, it is rather unrealistic that they do. As explained earlier using the adaptive brake light example, complex scenarios are very likely to involve messages in different subnetworks, which makes specification using traditional techniques difficult. Using our approach based on abstract protocols, we can accommodate for the heterogeneity of the messages involved in a scenario: We first combine all relevant messages onto a single abstract protocol layer and then use this protocol layer for the definition of the scenario layer. To define this unifying protocol layer using CFR models, we represent each combination of bus and protocol as a channel. A cascade of filters and rules then select the relevant messages from each channel. Rules are then applied to these messages in order to convert them to messages obeying the protocol of the unifying channel.

We require that each scenario is uniquely identifiable via its message sequence. Ambiguities have to be resolved at the specification level. While there usually are a great number of protocol messages on the communication bus, the challenge is to identify only those messages that are relevant for a specific scenario. After these relevant messages have been singled out, they need to be assigned to a distinct scenario. This is the same procedure that we have already discussed

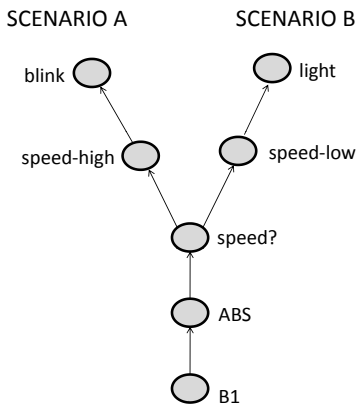

Figure 3: Two scenarios based on the adaptive brake light use case

when we introduced protocol abstraction. See Figure 2. In this case, the low-level protocol layer might be the message transport protocol, e.g. on a CAN bus, and the abstracted protocol layer is a scenario. We can then use the techniques explained in the previous section to relate these protocol layers.

We illustrate the application of CFR models for defining *scenario analyzers* with the two example scenarios we introduced in Section 2 in the context of the adaptive brake light. Figure 3 shows the message sequence of the two scenarios. Both scenarios are associated with the use case of a strong brake maneuver by the driver. Scenario A describes this use case under high speed, where the brake light blinks. Scenario B describes the case where the speed is low and the brake light operates normally. The two scenarios are depicted as sequences of messages, whereby both scenarios share a part of the message sequence. Message $B1$ marks the starting point for both scenario A and scenario B. If we observe the message flow sequentially, starting at $B1$ we have an ambiguity for the first three messages. Only after observing *speed-high* or *speed-low* we can uniquely identify scenario A or B. With each new message, it has to be checked whether this message is the beginning of a new scenario or the continuation of one or more scenarios under monitoring. Unless the path of consecutive messages is not unique, several scenarios remain as candidates.

Under real conditions, a large number of different concurrent scenarios might be monitored simultaneously, some of them still open to a final decision. For example, each occurrence of a $B1$ message in Figure 3 initiates a new instance of a scenario monitor. In CFR models, we use filters to sort out relevant messages and rules to assign sequences to scenarios. By describing scenarios through CFR models, we do not only specify them but at the same time also define the operation of an analyzer for the scenario level. Thus, we have a monitor for the scenario that enables us to verify it. We can refine our analyzers by assigning time intervals to the arrows connecting two messages. This defines how long the monitor waits for the continuation of a scenario. If the expected future is not confirmed within a given time frame, tracing of the scenario is canceled and an error is reported: Either the scenario has not been specified properly, or the flow of messages between two or more parties is in fact faulty due to misbehavior of one or more communication partners. We can identify under-specification by identifying message sequences that cannot be assigned to scenarios.

As we have already explained, scenarios can be viewed as abstract protocols and thus we can apply the same techniques to scenarios that we applied to protocols earlier. In particular, we can define sub-scenarios and abstract them using the technique of protocol abstraction. That is, we define an abstract protocol on top of the protocol at which the scenario messages occur. In this new layer, we indicate the occurrence of a scenario in a system by a single message. We can then use such layers to built new layers where messages represent compound scenarios. This approach is open-ended and allows stepwise definition of more complex scenarios through layering.

3.3 Test Automation through Reproduction of Complex Scenarios

In the previous subsection we described how to monitor and verify complex scenarios. We can use these techniques to reproduce complex scenarios with the aim to automate system tests by simulating the systems components. We assume that each functional component in a system has two interfaces: A protocol interface which refers to network based communication, and a service interface which refers to the application and possibly to the interaction with the environment. Assuming that each component of a system has thoroughly and successfully been tested, the system behavior can – to a certain extent – be derived from protocol messages. Thereby, the states of the components and their behavior in complex scenarios can be reproduced in a defined manner from observing the communication on the protocol interface. This opens the possibility to simulate resources and in this way also simulate complex scenarios.

Complex test scenarios are defined and verified on the level of the application on a use case basis. Functional components of the network under test can be replaced and simulated by message generators or stimulators, in order to trigger events and enforce specific reactions in the test scenario. In addition to that, stochastic stimuli, e.g. a user model, can be introduced by using our approach. In this way, triggers and events from the service interface can be kept to a minimum. The main benefit is that complex scenarios can be tested through their reproduction. While complex scenarios can consist of other complex scenarios, the reproduction of underlying scenarios for a higher level scenario is evident for automated testing.

3.4 Applications

Our *advanced protocol analysis* approach contributes to several different stages of the software development process. We present these contributions in the context of the V-Model, a widely used process model for software development in the automotive domain. The stages of the V-model are associated with our contribution in Figure 4.

The most apparent contribution of a protocol analyzer in the software development process is analysis on message level. The message level can be associated to the Module level in the V-Model. This is where Unit-Testing takes place, and where it is assured that system modules act in the desired way. So we can use a protocol analyzer to validate the

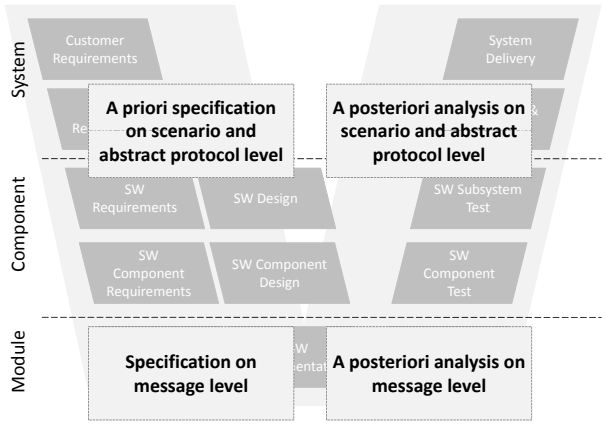

Figure 4: Identification of possible application areas in the V-Model

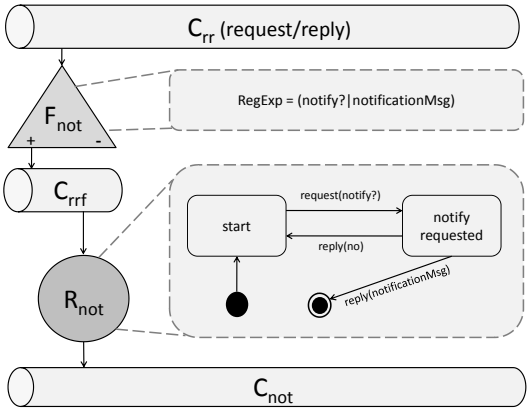

Figure 5: CFR model for the notification example in visual notation

correct communication behavior between several modules in different parts of the whole system network.

We have seen that our approach extends the scope of protocol analysis to complex scenarios, as we can define scenarios using abstract protocols. This way, our approach can be used at the system test level and also at the component test level of the V-Model. We can use analyzers for scenarios and abstract protocols to validate the correct behavior of system components and the system itself.

But there is more to our approach than just analysis. With the capabilities of monitoring message flows and learning from them – a task we aim to automate in future research – it is possible to document/specify different parts of a distributed system. Network protocols can be specified by observing the communication on the message level. Complex Scenarios can be specified by monitoring messages in a broader context, concerning components of a network system or the whole system itself.

So there are two different types of analysis that can be executed:

- Analysis with *a priori* knowledge

The challenge at this stage is to uncover under-specification in a given explicit specification.

- Analysis with *a posteriori* knowledge

Here we can uncover design intentions out of an implicit specification, e.g. an implementation.

We have seen that CFR models can be interpreted in two ways: As instructions of a protocol analyzer on how to perform protocol abstraction and as a way to specify protocol layers. The latter is another important contribution that we consider to be especially interesting for complex scenarios: Our CFR models can be used for the layered specification of communication behavior. We thus give an answer to the under-specification that we uncovered in Section 1 in conjunction with complex scenarios and abstract protocols. A side benefit that arises from the duality of interpretation of CFR models is that for each specification we define, we define at the same time an analyzer that can be used for verifying a system against the specification.

4. A DSL FOR PROTOCOL ANALYSIS

In this section we describe the implementation of our CFR approach in the form of a domain specific language (DSL) from which we generate full code for analyzers that target different platforms. We already introduced a textual notation for CFR models in Section 3. While this notation is suitable for simple models, complex models are better rendered using a visual notation that makes the arrangement and interconnection of elements immediately clear. In addition to that, we found that our visual language is less intimidating for experts in the automotive domain with possibly little background in general purpose programming.

4.1 Syntax and Semantics

In this section, we present abstract and concrete syntax of the CFR language. Figure 5 shows the CFR model for the notification example rendered using the visual notation. The state machine inside the rule is instantiated for different client/server combinations and is the rule's memory. The regular expression is a simple example for an expression that would be applied to the content part of a message. One of the benefits of our visual notation is encapsulation. It is in principle possible nest CFR models into components. Using this composition mechanism, reusable protocol analyzer components can be created and working at different levels of abstraction is encouraged.

Figure 6 shows a partial *metamodel* that defines the abstract syntax of our language. For space reasons, we omit the definition of state diagrams. The concept *model* within the meta model represents a container for the basic elements, so a model can be a CFR model itself. It may contain an arbitrary number of *processing units*. A processing unit is an abstract concept that generalizes any concrete message-handling entity. In other words, it is sub-classed by the concepts *channel*, *rule* and *filter*. A processing unit itself may contain at most one model. This containment relation represents the structural composability of our approach. As discussed earlier, processing units are connected through connectors by using the pins of the processing units.

Inherent to our design is the idea to keep the meta model itself as simple as possible and to express details with OCL (Object Constraint Language) [2] constraints. OCL Constraints are part of the Unified Modeling Language (UML) and are used to add detail to UML (meta) models. A con-

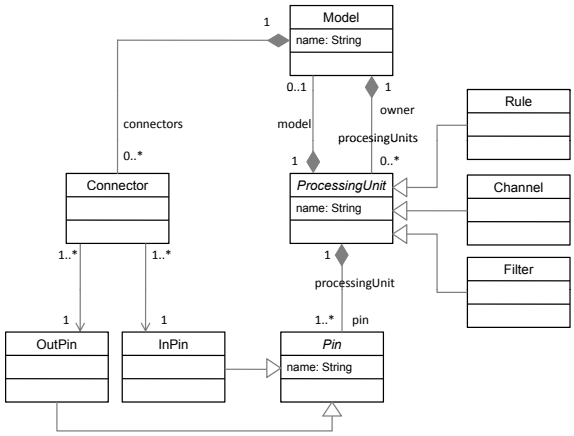

Figure 6: Partial meta model

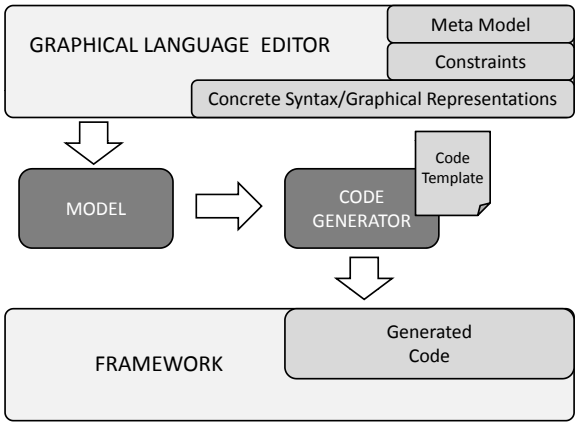

Figure 7: Implementation overview

straint is an expression that has to be validated to *true* at any time. We use OCL in conjunction with our meta model to complete the abstract syntax specification. Due to space reasons, we will only provide a simple example of a constraint and point the interested reader to the rich literature on the subject [8]. The constraint that prevents a CFR model from containing its parent model as a child can be defined in the following fashion:

```
context model:
inv noSelfContainment:
processingUnits->forAll(model <> self)
```

The first line defines that the following expressions are applied in the context of a model. The keyword `inv` in the next line stands for *invariant* and says that the following expression must be valid at any time in the system. Finally, the expression says: "For every processing unit in this model it is true that every model in that processing unit is not the model on which context the rule is based."

As mentioned above, the concrete syntax of our language is visual. We represent the three basic elements as follows:

- Channels as pipes; pipes are an intuitive metaphor, signifying the capability of messages to flow through channels
- Filters as triangles; a triangle is a natural shape for representing a component with one input and two outputs
- Rules as circles; circles are used because rules can have an arbitrary amount of outputs

Every element contains *inPins* and *outPins*. Connectors are represented by a line between two pins with an arrow head that indicates message flow direction. The pin concept is adopted from the electronics domain. Pins are represented as points on the edges of filters and rules. While a channel has an infinite amount of pins, these are not explicitly represented. The concrete syntaxes for the configuration of filters and rules can be divided into syntax for regular expressions and the syntax for state machines. Our syntax for state machines is adopted from UML state charts [10]. Regular expressions are represented as in the Perl programming language [22].

The informal semantics of the three basic elements of our language have already been defined in Section 2. In addition to that, the exact semantics are defined by a translator semantics for our two target frameworks.

4.2 Implementation

As a proof of concept, we created a prototype implementation of CFR using the meta modeling and code generation tools available for the Eclipse [24] platform. An overview of the different parts of the implementation is provided in Figure 7.

The definition of the abstract syntax of our language is based on a meta model created using EMF (Eclipse Modeling Framework) [7]. EMF provides an elegant, light weight modeling language for creating meta models. EMF offers a tree-style user interface to create *ecore-models* that are based on Essential MOF (Meta Object Facility) [3], a simple subset of the MOF. EMF also has several built in functionalities for generating code and for editing and creating models, based on a meta model representation. The generated code is in the form of a set of Eclipse plugins that offer a tree-based interface for model editing and creation.

To define the language constraints and in order to have error detection at edit-time for CFR models, we decided to use oAW (openArchitectureWare)[1], a modular MDA/MDD generator framework. Among other functionalities, it provides the definition and checking capabilities for constraints on ecore meta models. The definition of such constraints is done in a special language that is affected by and is similar to OCL. Besides the pure constraint definitions, error messages can be defined to address constraint violations to the user. oAW provides a convenient way to add the constraints to the model editor, created by EMF, and check the constraints at edit-time.

We used the GMF (Graphical Modeling Framework) [24] to create an editor for our language. GMF is based on EMF and provides functionalities for creating graphical editors for ecore-models. On the basis of an existing meta model the following needs to be defined:

- Graphical representations for the primitives of the language. That is, the concrete syntax of the language.

[1]http://www.openarchitectureware.org

- A toolbox for the later use of these graphical representations in an editor.
- The relationships between toolbox elements and graphical representations of the meta model elements.
- Settings for special behavior functionalities of the editor.

GMF also provides sophisticated code generation functionalities and creates on the basis of the meta model, the generated code from the meta model and the graphical definitions and settings, Eclipse plugins, that can directly be used as a graphical editor for models, based on the previously defined abstract syntax. The result is a graphical editor for CFR models that provides convenient and intuitive drag&drop editing. To navigate through different abstraction levels in the model or to open a rule and edit its logic, it is possible to simply double click a node. Another editor window will open and allow the user to edit the desired functionalities.

Our reason for choosing meta modeling and code generation techniques was our need for generating complete analyzers from CFR models. oAW provides a very powerful framework for this task. Based on a meta model it is possible to create code-templates that can then be used to produce runnable code out of models, based on the underlying meta model. The generator scans the model and processes it in a tree processing manner. Code fragments can be defined for different nodes reached in the model while processing it. For instance, the initialization of all channels in a model could look like this:

```
<<DEFINE main FOR Model>>
<<FOREACH procUnits.typeSelect(Channel) AS c>>
<<c.name>> = Channel("<<c.name>>")
<<ENDFOREACH>>
<<ENDDEFINE>>
```

In the context of a model all processing units are checked for being channels and if they are, a *Channel* object is initialized, with the name of the channel. The generated source code is Python code, more precisely configuration code for a Python-based framework [16]. We developed this Python-based simulation framework to explore the possibilities of our language and as a proof of concept. As a second platform, we target a framework that reads messages from a MOST network using a commercial protocol analyzer [17]. There are several benefits that our approach gains from using template based code generation:

- The target platform type is not restricted.
- Templates can be easily exchanged to create code for new target platforms.
- Template changes have no effect on the model.
- Implementation knowledge is in the template, so errors can be reduced.

5. RELATED WORK

We are currently unaware of other approaches that apply protocol analyzer techniques to abstract protocols and complex scenarios. However, our approach can be related to the large body of work done in forward engineering and analyzing protocols. Indeed, protocol analyzers are a standard technique for monitoring, testing and reverse engineering systems and many academic and commercial implementations are available [25], e.g. Ethereal [19]. The aim of our approach is not to implement another protocol analyzer that understands a set of predefined protocols. Instead, we aim to provide an elegant specification mechanism that makes protocol analyzers programmable and extends their scope to include abstract protocols and scenarios - not only standard protocols. Staffing a protocol analyzer with features to capture abstract protocols is essentially a reverse engineering activity. It is, so to speak, the reverse of approaches such as OSI [5]. Our claim that basically three concepts suffice to reverse engineer protocol-based systems is closely related to approaches in forward engineering protocols, protocol stacks and protocol-based services that aim to base formalisms on few basic concepts. For instance, in [13], the authors provide a formal approach to modeling layered distributed communication systems with a small number of concepts only. A mapping between their and our concepts seems possible and might help us substantiate our approach by a formal foundation. This is work left for future research. The importance of scenarios and associated heterogeneity challenges in automotive networks has been recognized by many authors, e.g in [21], [12] and [9].

6. CONCLUSIONS AND FUTURE WORK

In this paper we presented a novel approach that extends the scope of protocol analyzers to include application level communication abstractions and complex scenarios. We motivated our research by the growing complexity of networks in automobiles and the resulting risk of under-specification. We identified abstract protocols as a powerful basis for the specification, understanding and testing of systems. We identified complex scenarios as a key challenge for the development of reliable systems. We gave a precise definition of scenarios on the message level and related them to use cases. After pointing out similarities between protocols and scenarios, we concluded that analyzers for both can be specified using CFR models based on three basic concepts: channels, filters and rules. CFR models can be interpreted in two similar but distinct ways: As a specification that defines a higher protocol layer through a lower protocol layer, or as a specification for a protocol analyzer that can render communication on a higher layer by observing communication on a lower layer.

A side benefit that arises from the duality of interpretation of CFR models is that by defining a specification, we define at the same time an analyzer for verifying it. One of the strengths of our approach is its composability: Both the existing lower layer and the defined higher layer are represented as channels and there is no conceptual difference between the two. This means that our approach allows for layered definition of protocol stacks. In the context of complex scenarios, this allows us to accommodate for the heterogeneity of the messages involved, i.e. we can treat messages on different layers, using different protocols in different subnetworks as if they were sent using the same protocol on the same channel. In addition to the approach itself, we presented a DSL based on it and discussed its implementation. Our language can be used by experts in the automotive domain with little or no programming experience.

We are currently investigating several directions for future research. So far, our language implementation allows generation of analyzers for a Python-based simulation environment and for a framework that reads messages from a MOST-bus. We are working on an implementation for a framework that provides us with messages from other bus systems, including CAN, so that we can evaluate our approach for scenarios involving different subnetworks. Another direction we are currently pursuing is the application of machine learning techniques for the automatic configuration of filters and rules for complex scenarios. We expect this to be a valuable technique for re-engineering underspecified systems. Although the result might still not yield a complete specification of all possible scenarios between a set of communication partners, the documentation of valid scenarios is already highly valuable. Finally, we are aiming to give our approach a stronger formal basis. We are currently experimenting with the translation of CFR models into Finite State Processes (FSP) [15]. Using this formalism, we represent single messages as actions, message sequences as processes and use the parallel composition operator to express interleaving of messages.

7. REFERENCES

[1] CAN Specification Version 2.0. Technical specification, Robert Bosch GmbH, 1991.

[2] OCL 2.0 OMG Final Adopted Specification. Technical specification, OMG, Oct 2003.

[3] Meta Object Facility (MOF) Core Specification, v2.0. Technical specification, OMG, Jan 2006.

[4] Most specification rev. 2.5. Technical specification, MOST Cooperation, 2006.

[5] *Information Technology – Open Systems Interconnection – Basic Reference Model: The Basic Model*. ITU-T Recommendation X.200, International Telecommunication Union, July 1994.

[6] J. Botaschanjan, L. Kof, C. Kühnel, and M. Spichkova. Towards verified automotive software. In *SEAS '05: Proceedings of the second international workshop on Software engineering for automotive systems*, pages 1–6, New York, NY, USA, 2005. ACM Press.

[7] F. Budinsky, S. A. Brodsky, and E. Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[8] T. Clark and J. Warmer, editors. *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*. Springer-Verlag, London, UK, 2002.

[9] F. Corno, S. Tosato, and P. Gabrielli. System-level analysis of fault effects in an automotive environment. In *DFT '03: Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, pp. 529-536*, Washington, DC, USA, 2003. IEEE Computer Society.

[10] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[11] J. Hall. Multi-layer network monitoring and analysis. Technical report, University of Cambridge, July 2003.

[12] A. Hamann, M. Jersak, K. Richter, and R. Ernst. A framework for modular analysis and exploration of heterogeneous embedded systems. *Real-Time Syst.*, 33(1-3):101–137, 2006.

[13] D. Herzberg and M. Broy. Modeling layered distributed communication systems. *Formal Aspects of Computing, 28(4):751-763*, May 2005.

[14] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[15] J. Kramer and J. Magee. *Concurrency: State Models and Java Programs, 2nd Edition*. John Wiley & Sons, April 2006.

[16] A. Martelli. *Python in a Nutshell, 2nd Edition*. O'Reilly Media, Inc., Sebastopol, CA, USA, July 2006.

[17] A. Meroth and D. Herzberg. An open approach to protocol analysis and simulation for automotive applications. In *Embedded World Conference*, 2007.

[18] A. Meroth and B. Tolg. *Infotainmentsysteme im Kraftfahrzeug: Grundlagen, Komponenten, Systeme und Anwendungen*. Vieweg, Wiesbaden, 2008.

[19] A. Orebaugh, G. Morris, and E. W. G. Ramirez. *Ethereal Packet Sniffing*. Syngress, February 2004.

[20] A. Pretschner, M. Broy, I. H. Kruger, and T. Stauner. Software engineering for automotive systems: A roadmap. In *FOSE '07: 2007 Future of Software Engineering*, pages 55–71, Washington, DC, USA, 2007. IEEE Computer Society.

[21] C. Seybold, S. Meier, and M. Glinz. Scenario-driven modeling and validation of requirements models. In *SCESM '06: Proceedings of the 2006 international workshop on Scenarios and state machines: models, algorithms, and tools*, pages 83–89, New York, NY, USA, 2006. ACM Press.

[22] E. Siever, N. Patwardhan, and S. Spainhour. *PERL in a Nutshell*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[23] A. S. Tanenbaum. *Computer Networks*. Prentice Hall PTR, Upper Saddle River, New Jersey 07458, 4th edition, 2003.

[24] The Eclipse Foundation. Eclipse IDE. http://www.eclipse.org, 2007.

[25] E. Wilson. *Network Monitoring and Analysis*. Prentice Hall International, 2000.

A Domain Specific Language for Uncovering Abstract Protocols and Testing Message Scenarios

Tim Reichert[1], Dominikus Herzberg[2]

[1]School of Computing, Engineering & Inf. Sciences, Northumbria University
Newcastle upon Tyne, NE2 1XE, United Kingdom
tim.reichert@unn.ac.uk

[2]Department of Software Engineering, Heilbronn University
74081 Heilbronn, Germany
herzberg@hs-heilbronn.de

Abstract: We present CFR, a language for the specification of protocols and communication scenarios and show how this language can be used for systematic testing of distributed systems. Our language is based on three basic concepts, namely channels, filters and rules. Using our current implementation, we can fully generate sophisticated analyzers from CFR specifications.

1 Introduction

Many of the errors in today's distributed systems occur in the context of complex communication scenarios involving multiple distributed components. Such errors are intricate and cannot be found by component testing alone. System testing complex scenarios is, however, a difficult task, mainly because it is hard to specify correct behavior at the right level of abstraction and in a way that can be used for automated testing.

Protocol analyzers can be used for monitoring and testing system communication on different layers of a protocol stack. However, the parts of the communication design that are not specified as part of a protocol are invisible during analysis. Instead, only the effect a design has on protocol layers can be traced. This has severe consequences, as test cases have to be specified at an inappropriate level of abstraction, which makes them overly complex and likely sources of errors. We provide an example of this in Section 3.

We developed a language called CFR (Channel Filter Rule) that allows comprehensive specification of communication behavior and the full generation of specialized protocol analyzers that can monitor and test this communication behavior. The main purpose of the CFR approach is to uncover and document hidden design intentions in protocol use and to enable specification of communication behavior at appropriate levels of abstraction.

2 The CFR Language

In this section, we introduce CFR, a domain specific language for the specification of both protocols and message scenarios. CFR models can be interpreted in two ways: As a specification of a protocol or as a specification of a protocol analyzer. In our current implementation, we use them to generate analyzers that can monitor and automatically test even complex scenarios.

A protocol layer can be described by relating its communication patterns to the communication patterns on the next lower layer. CFR models do this in a machine-processable form by combining the following basic language constructs:

- A *channel* is a medium that transports messages. The messages obey a protocol.
- A *filter* is a passive and stateless unit that redirects messages from an incoming channel to one of two outgoing channels. Depending on a given pattern, the filter determines which message is to be delivered to which outgoing channel. Since both the input and output of a filter are messages, the output of a filter can be used as the input of another filter. Filters might be put in parallel or in sequence. Filters are passive elements as they do not modify or create messages.
- A *rule* is an active and stateful message processing unit that consumes messages from one or more incoming channels and returns messages to one or more outgoing channels. In opposition to filters, a rule can modify incoming messages or create new messages. To do that, a rule contains defined logic in form of a state machine or a set of state machines with a selector function. Similar to filters, the output of one rule might be the output of another rule.

We represent the inputs and outputs of rules by pins with connectors between them. The message flow is from output to input pins. Channels and rules have an arbitrary number of input and output pins, while filters have exactly one input and two output pins. We use regular expressions to define the selection criteria in filters. State machines are used to define the memory of rules. We provide an example of a CFR model in Section 4.

3 Uncovering Abstract Protocols

A concrete protocol is a specification of messages and a set of message sequences. An abstract protocol constitutes its own communication architecture based on a concrete protocol. The communication architecture may be layered. At application level, however, strict layering is often impractical and broken by tunneling. An example is due. HTTP is a stateless and connectionless concrete protocol. It consists solely of request and reply messages and only a client can initiate communication via a request message. An application using HTTP may compensate for HTTP limitations and define its own interaction scheme. For instance, it might be necessary for a web browser based application to receive

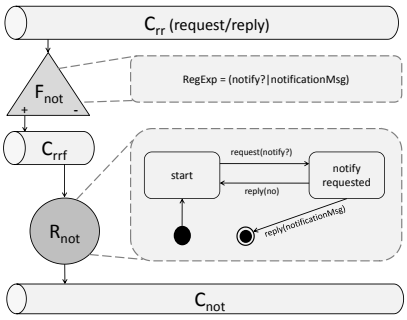

Figure 1: CFR model for the notification example in visual notation

notifications of important events from a web server. As the server cannot contact the client directly, an application programmer needs to design an abstract protocol to implement event notifications based on HTTP.

One way to realize the "illusion" of a server-based notification service is to let the client periodically send request messages to the server, asking whether there is a notification available. Upon receiving a request, the server replies by either sending a negative indication or it delivers the notification. This is exactly how the now popular AJAX web technology works. Other parts of the application might use the notification service via the abstract protocol – but they are not forced to do so. Application designers working with such a notification service certainly think in terms of notifications and probably do not even know about requests and replies. A conventional protocol analyzer on the other hand would only analyze the HTTP protocol, not knowing anything about the abstract protocol and its communication architecture. With CFR, we can specify abstract protocols and generate analyzers that understand these protocols.

Figure 1 shows a CFR model that defines the notification protocol just described. In our visual notation, Channels are tubes, filters are triangles and rules are circles. Channel C_{rr} transports all request/reply messages. The Filter F_{not} ensures that channel C_{rrf} transports only those requests and replies related to the notification service. The rule R_{not} uses a state machine to transform a request followed by a positive reply into a notification message and puts it on channel C_{not}.

4 Testing Scenarios with CFR Analyzers

Communication behavior in a distributed system can be described by a set of scenarios. A scenario for the aforementioned example might, for instance, be an update occurring on the server of which the client application needs to be notified. On the communication level, this involves at least two messages: An update message to the server followed by a notification message to the client. An effective method for testing the correctness of such a scenario is to stimulate the server by sending an update message and to check if the

expected notification message is sent. Using a conventional protocol analyzer, the stimulation and testing could in principle be done. However, the specification of the scenario would be at an inappropriate level of abstraction, namely at the request/reply layer.

If we view scenarios as sequences of messages, we can use CFR models to specify them as specialized high-level abstract protocols. By following a layered approach [HB05], we ensure that these specifications are always at the appropriate level of abstraction. We can specify even complex scenarios involving different protocols and subnetworks by combining all relevant messages onto a single abstract protocol layer and by then using this protocol layer for the definition of the scenario layer.

We require scenarios to be uniquely identifiable via their message sequences. Ambiguities have to be resolved at the specification level. This ensures that the step from merely monitoring a system to testing it is feasible. Based on a set of CFR models, we can use our analyzers to identify unspecified - and thus potentially faulty - communication behavior. In addition to that, we can use the analyzers to systematically stimulate components in order to trigger scenarios that are then verified. Under real conditions, a large number of different concurrent scenarios might be monitored simultaneously. We can refine our analyzers by assigning time intervals to the transitions of state charts within rules. This defines how long the monitor waits for the continuation of a scenario. If the expected future is not confirmed within a given time frame, tracing of the scenario is canceled and an error is reported.

5 Conclusions and Future Work

Our work is related to the large body of work done in forward engineering and analyzing protocols. Using a formalism based on channels, filters and rules, augmented with state charts and regular expressions, we are able to specify, monitor and test even very complex scenarios in distributed systems. Our layered approach ensures that specification can always be performed at the right level of abstraction. We have developed both a textual and visual notation for our language and extensively use code generation techniques to generate analyzers for different technology platforms. Currently, we are using our approach for testing AJAX based web applications and automotive systems [MH07] in order to gain more real-world experience. In future work, we will apply machine learning techniques for the automated extraction of CFR models from communication data and we are working on giving CFR a stronger formal basis by mapping CFR models to finite state processes.

References

[HB05] Dominikus Herzberg and Manfred Broy. Modeling layered distributed communication systems. *Formal Aspects of Computing, 28(4):751-763*, May 2005.

[MH07] Ansgar Meroth and Dominikus Herzberg. An Open Approach to Protocol Analysis and Simulation for Automotive Applications. In *Embedded World Conference*, 2007.

TEACHING LANGUAGE-DRIVEN SOFTWARE ENGINEERING

Tim Reichert

Northumbria University / Heilbronn University
Newcastle upon Tyne, UK / Heilbronn, Germany
tim.reichert@unn.ac.uk

Dominikus Herzberg

Faculty of Informatics, Software Engineering Department
Heilbronn University, Germany
herzberg@hs-heilbronn.de

Abstract

Language-driven software engineering requires that software engineers not only use but also design and implement computer languages. Teaching this new approach is difficult as it requires an understanding of a wide range of concepts usually taught in a number of different courses. We developed a novel concept for a course that aims to teach language-driven software engineering to undergraduate students in a unified, accessible and up-to-date manner. At the center of our course is XMF (XML Modeling Framework), an interactive teaching tool we created for modeling and meta-modeling. Our tool is based on web technologies and runs in a web browser. Using a pattern-oriented formalism, students define their own languages and modeling views. These definitions are used by a powerful transformation engine for syntax checking and bidirectional view transformations. Built-in constraint functionality makes the modeling experience interactive as it can be used by teachers for specifiying models and by students to check their designs against these specifications. We use XMF not only to introduce basics of creating and modelling languages but also to explain advanced concepts such as meta-level design, syntactic layering and program transformation.

Keywords - Teaching, Software Engineering, Programming Languages, Modeling, Tools

1 INTRODUCTION

The difficulty of programming a computer depends to a large degree on the use of the right programming language. It is thus no wonder that the biggest leap in programmer productivity was achieved with the introduction of “high level” programming languages. These languages abstract from the details of a particular machine and focus on the needs of the programmer. Raising the level of abstraction even further to solve today’s increasingly more complex programming problems demands a similar leap from languages that offer general solutions to highly problem-specific languages.

Modern approaches in software engineering, e.g. Model Driven Development (MDD) [1], Domain Specific Languages (DSLs) [2] and Generative Software Development (GSD) [3], represent this important trend in research and practice. The design, implementation and use of specialized languages have become a fundamental part of the software development process. The software engineer is no longer a language user only, but also a creator of languages.

For educators the challenge is how to teach this new understanding to software engineering students as it is based on a breadth of knowledge usually taught in a number of different courses. As part of the government-sponsored ”Novel Teaching Approaches” program we took up that challenge and designed a novel course that aims to teach Language-Driven Software Engineering (LDSE) to undergraduate students in a unified, accessible and up-to-date manner. The course we designed over the last two years combines concepts that are traditionally taught as part of formal language theory, compiler construction, modeling, programming and software engineering.

The basic idea underlying our course is an interactive teaching tool that we created specifically to satisfy the needs of our students. The tool, XMF (XML Modeling Framework), aims to make modeling (using languages) interactive and meta-modeling [4] (creating languages) comprehensible. XMF is easily accessible for students as it is based on familiar web technology. All editing of models and meta-models is done in a web browser, either in XML or through specific HTML views. Model validation through inner- and inter-model constraints leads to a modeling process that is controllable by teachers and highly interactive for students – with immediate feedback from the system in case of errors or design flaws. New languages can be designed with a built-in pattern language. A query interface allows students to ask XMF questions about models and thereby verify whether their design intentions are captured by their creations.

Aside from using XMF to create, relate and use various languages, an important part of our course is to look at "the big picture" of language-driven software development by relating XMF to other technologies. For example, we compare the meta-model of XMF to the four level architecture of UML (Unified Modeling Language), show the relationship between our pattern language and grammar formalisms such as EBNF (Extended Backus Naur Form) and compare XMF's transformational approach to macro mechanisms found in languages of the Lisp family.

The rest of this paper is organized as follows. The following section, Section 2, describes XMF from a high-level perspective, showing how the tool looks and how it basically works. Section 3 goes into more detail by introducing the mechanisms for defining and relating languages. Section 4 gives three examples of advanced concepts in the context of LDSE that we teach using XMF. In Section 5 we draw conclusions from our use of XMF in the classroom and outline future work.

2 XMF OVERVIEW

The most important goal when we designed XMF was to make modeling (using languages) and meta-modeling (creating languages) accessible and comprehensible for students. Our strategy consisted of two parts. Firstly, to lower the learning curve by basing the tool on a technological platform that our students are familiar with and, secondly, to make the modeling process highly interactive by providing automated feedback. Regarding the technological platform, browser technologies were a natural choice as they are already part of our curriculum. Thus, in XMF, models and meta-models are stored in XML. Editing is done either in XML or using a HTML-based view mechanism directly in a web browser.

One problem when teaching modeling in contrast to teaching programming is that models are not necessarily executable. Thus, students cannot run their models as they would run a program and react to feedback from the system. To enable such trial and error, i.e. to make the modeling experience interactive, we added a constraint mechanism to XMF. By using a JavaScript library teachers formulate constraints for a model; students can then check their models against these constraints, receiving valuable feedback on their validity.

A language in XMF consists of a language schema, a set of constraints and optionally a view definition. The schema defines the abstract syntax of the language. Since XML is the basis of all models, this is done by restricting the set of valid tag names, nesting and data values. Constraints may be used to refine the abstract syntax or to impose semantic constraints. Views define the concrete syntax of models, i.e. how a model is actually created or edited using a web browser. If no view definition is given, the default concrete syntax of models is XML.

Fig. 1 is a screenshot of XMF displaying two models at the same time. This is useful when there are relationships between two models as it is for example the case with class and an object models. The tabs above each model are for navigating between different views on the model and also for switching to the language definition. When displaying the model using a view, the rendering engine of the browser is used. The constraint box is for formulating model constraints and queries against one or more models using a JavaScript API. The log box shows the results of executing these constraints or queries on the models.

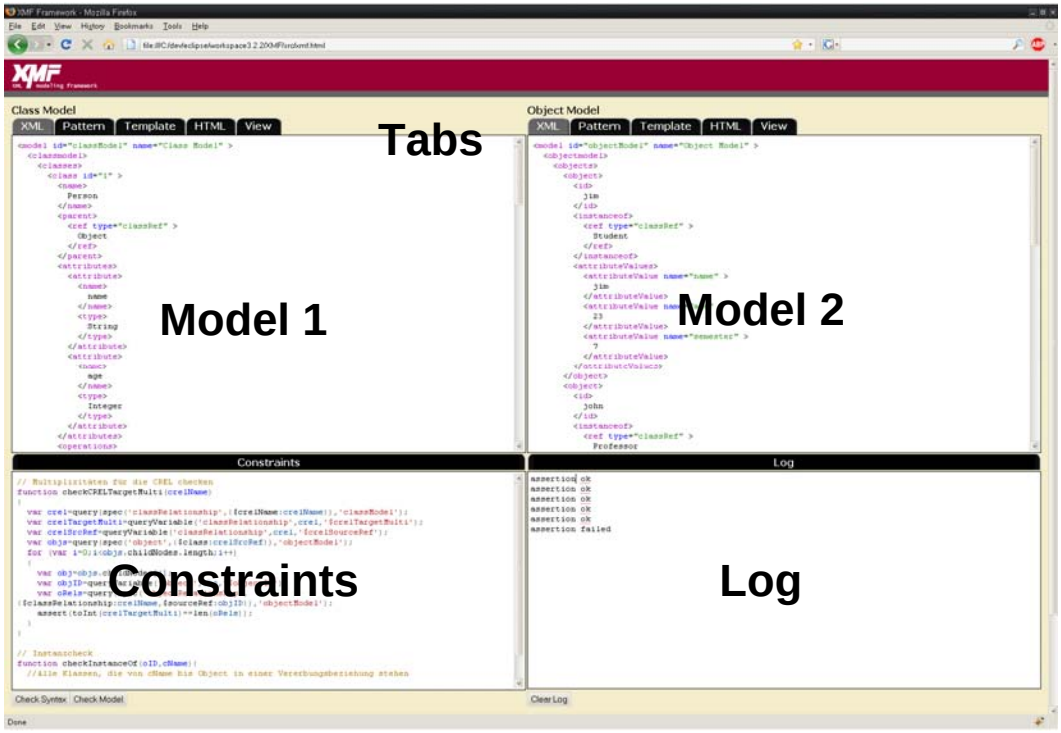

Figure 1: Screenshot of XMF running in Firefox

XMF has two example modeling languages built in, one for class and one for object modeling. The following XML-fragment is an example of a class definition in the class modeling language. It describes a class with id “1”, name “Person”, a parent class named “Object”, two attributes and one operation. In Section 3 section we will describe how such a modeling language can be defined using patterns.

```
<class id="1">
  <name>Person</name>
    <parent><ref type="classRef">Object</ref></parent>
  <attributes>
    <attribute><name>name</name><type>String</type></attribute>
    <attribute><name>age</name><type>Integer</type></attribute>
  </attributes>
  <operations>
    <method><name>saySomething</name><params>x</params></method>
  </operations>
</class>
```

From a technological point of view, an advantage of using XML is that it is straightforward to process, Through its rigid syntactic framework it provides clear guidance on how to define new model syntax. As XML is quite verbose, editing models in a different representation – a view – makes sense. Views in XMF are HTML descriptions of a model (possibly containing JavaScript) that are rendered in the browser. Fig. 2 shows a view on the class described in the XML fragment above. Both normal and editing modes are shown.

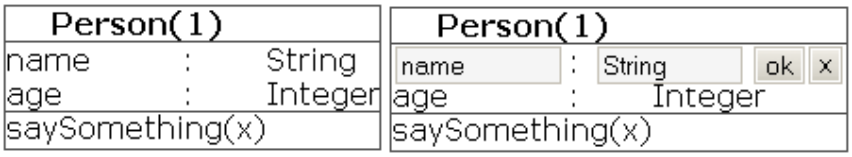

Figure 2: Example of a view on a class

3 XPLT LANGUAGE AND TRANSFORMATION ENGINE

A central component of XMF is the language XPLT (XML Pattern Language for Transformations) and its underlying transformation engine. This is also what students learn first when starting to use XMF to define languages. XPLT, which we developed specifically for XMF, is used for defining schemas and views through patterns. It also provides mechanisms for establishing relationships between schemas, e.g. the relationship between a model and its view. In Fig. 3 we use the example of a class modeling language to show the connection between schemas, models and views. An XPLT schema defines the basis of a class modeling language through patterns that state how class models may look like. A class model in this sense is an XML-document that matches this pattern. Similarly, the XPLT schema for a view defines how a class model might look like in HTML and a class model view is any matching description of the model in HTML. The relationship between the two schemas for models and views is established through variable names in the schema. If these relationships are properly set up, the transformation engine of XMF can automatically transform from model to view and back which allows editing the model through the view.

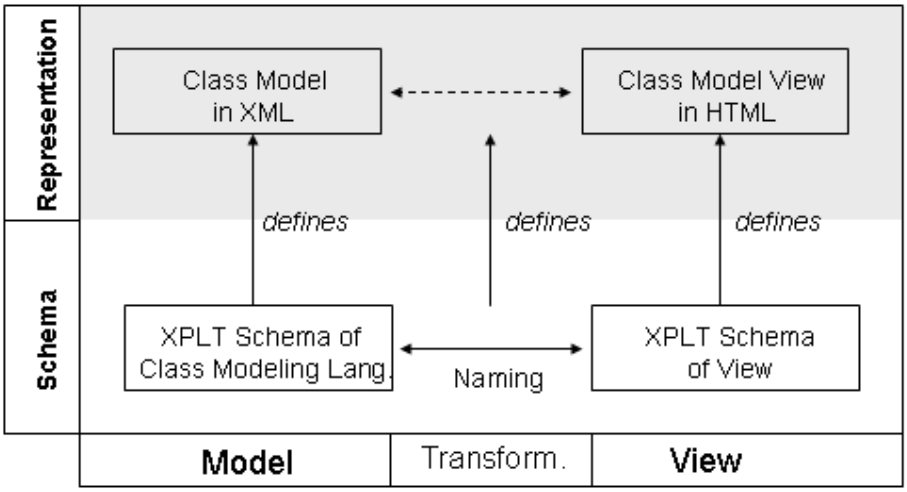

Figure 3: Relationship between schemas, models and views

An XPLT pattern is a mixture of concrete syntax of the modeling language and XPLT meta-operators. In the following section we will describe these operators.

3.1 Language Abstractions and Syntax

XPLT is comparable to XML schema languages such as W3C XML-Schema or DTD (Document Type Definition) in particular and in general to other grammar formalisms such as EBNF (Extended Backus-Naur Form). Similar to EBNF, it provides operators for expressing sequencing, alternation and repetition and also supports recursion. The most important constructs of XPLT are introduced in the following. It shall be noted that XML patterns should be read as defining restrictions on a Document Object Model (DOM) tree, not on XML strings directly.

Literal Writing an XML fragment without variables enforces that the XML-Fragment must be present in the model in exactly the way it is given in the pattern. For example, the pattern named “literally”

```
<pattern id=“literally“>
  <a>hello</a>
</pattern>
```

defines that the model needs to contain the element `<a>hello</a>`. In the following examples we will ignore the pattern tag that usually surrounds each pattern.

Variable As opposed to literals, variables allow arbitrary structures at the place in the model that corresponds to where the variable occurs in the pattern. Variables always start with a dollar character.

The pattern <a>$var simply defines that the model has an a-tag as parent and arbitrary children, for example <a>hello or also <a>hello.

Alternative To express that different structures are allowed in a particular place in a model, alternatives are used. For example, the pattern

```
<alternative>
  <a>$x</a>
  <b>$x</b>
</alternative>
```

allows the occurence of either the a- or the b-tag in a model.

Sequence If model elements are required to appear in a certain sequence, this is defined using the sequence-tag:

```
<sequence>
  <a>seq1</a>
  <b>seq2</b>
</sequence>
```

Repetition If elements in a model may occur an arbitrary number of times at a certain position this is expressed using repeat. For example, the pattern

```
<a>
  <repeat container=%c>
      <b>$x</b>
  </repeat>
</a>
```

expresses that “a” might have any number of children “b”. Providing a container name that always starts with the percentage character is relevant for using the patterns for transformations. An example of a fragment that would match the pattern above is:

```
<a>
  <b>test1</b>
  <b>test2</b>
  <b>test3</b>
</a>
```

Ignore To ignore certain parts of a model the ignore tag is used as shown in the example:

```
<ignore>
  <a>$x</a>
</ignore>
```

Reference By using references, patterns may be defined on the basis of existing pattern. This is the foundation for modularisation, reuse and also for recursive definitions in XPLT. For example, the pattern

```
<a>
  <pref id=”b”/>
</a>
```

references the pattern with id b. This corresponds to copying the pattern at the place of the reference.

To give an impression of how XPLT models look like, we give a more complete example of its use. The following pattern defines how classes are represented in XMF's built-in class modeling language. An instance of this pattern was introduced in Section 2.

```
<pattern id="class">
  <class>
    <name>$cname</name>
    <parent><ref type="classRef">$parent</ref></parent>
    <attributes>
      <repeat container="%attributes"><pref id="attribute" /></repeat>
    </attributes>
    <operations>
```

```
      <repeat container="%methods"><pref id="method" /></repeat>
    </operations>
  </class>
</pattern>
```

The pattern uses a pref-Tag to refer to existing patterns for attributes and methods. It contains a variable $cname at the position of the class name.

3.2 Transformation Engine

What sets XPLT apart from other schema languages is that XPLT-patterns are not only used for syntactic validation of XML documents, but are the basis for transforming and querying models. To verify that a model fragment is an instance of a pattern, the fragment is matched against the pattern. The result of a successful match is a set of bindings for all the variables in the pattern:

$$match(pattern, model) \Rightarrow bindings$$

Patterns cannot only be used to check data but also to generate data. During this process the variable parts of the pattern are filled with the information from the bindings:

$$instantiate(pattern, bindings) \Rightarrow model$$

To unambiguously instantiate an XPLT-pattern, a value for every variable in the pattern, the number of occurrences for every repeat and the choice taken for every alternation must be clear. While the first is critical, the XPLT instantiation engine tries to deduce the second and third. This is possible in some cases and the deduction strategy is as follows: The number of occurrences is the count of variable bindings in the respective repeat container. The choice taken is deduced from the variable bindings themselves. The strategy is based on the premises that here must be a variable in every choice, that variable names must be different across choices in an alternation, that there must be containers for every repeat and that there must be a binding for every variable.

Based on matching and instantiating patterns, a transformation relationship for patterns whose matching fulfils the above premises can be defined

$$transform(pattern_1,pattern_2,model_1) \Rightarrow model_2$$

Here, $model_2$ is an instance of $pattern_2$. transform relates to match and instantiate in the following way:

$$transform(pattern_1,pattern_2,model_1) \Leftrightarrow instantiate(pattern_2, match(pattern_1, model_1))$$

This means, that instead of defining transformations explicitly, as for example the W3C XSL Transformation language (XSLT) does, XPLT's transformations are defined implicitly through naming conventions in the source and target patterns. In other words: For many structurally similar patterns it is possible to compute transformations automatically.

3.3 Views and Bidirectional Transformations

Especially when editing data through a view it is useful to have transformations that work in two directions. Whenever the user switches views on a model, the data has to be transformed between the two representations creating the view. As XPLT transformations are not explicitly defined, there is no problem of the form "calculate an inverse transformation from a given transformation" as with other transformation engines. Instead, the following equation must hold for all instances $model_1$ of $pattern_1$ and all instances $model_2$ of $pattern_2$:

$$transform(pattern_2,pattern_1, transform(pattern_1,pattern_2,model_1)) \Leftrightarrow model_1$$

This is true when matching the patterns produces the same bindings. For such cases bidirectional transformations are guaranteed. The following example pattern enables a bidirectional transformation when combined with the pattern for classes in Section 3.1. It defines the view in Fig. 2:

```
<pattern id="classHTML">
  <TABLE cellpadding="2">
    <THEAD>
      <TR><TH colspan="2"><SPAN>$cname</SPAN></TH></TR>
    </THEAD>
    <TBODY>
       <repeat container="%attributes"><pref id="attributeHTML"/></repeat>
```

```
    </TBODY>
    <TBODY>
       <repeat container="%methods"><pref id="methodHTML"/></repeat>
    </TBODY>
  </TABLE>
</pattern>
```

Based on the two patterns, transformations from XML to HTML and from HTML to XML can be performed automatically by XMF's transformation engine.

3.4 Constraint and Queries

XMF allows for elaborate queries against a single model or a whole set of models. Querying in XMF is realized through matching. The query component tries to match every element in a model against a given pattern and returns a list of matches. This list of matches can then be used as target for further queries. Constraints in XMF typically take the form "for all objects of type T in the model M, condition C must be fulfilled" or "for all objects O_1 of type T_1 in model M_1 must exist an object O_2 of type T_2 in model M_2 where $O_1.x=O_2.y$". Such constraints can be formulated by querying models with partially instantiated patterns. For example, the following query returns all classes with name "Student" from a given model:

```
query(instantiate-partially(cpattern, {$name,"Student"}),cmodel)
```

Partial instantiation works as follows. Every XPLT pattern defines a set of structures that have some commonality expressed in the pattern and some variability. Instead of instantiating a pattern we can also refine it by fixing some of the variability. Whereas instantiation fails when bindings are missing for a variable in a pattern, partial instantiation yields a pattern with the variable for which no bindings were found un-instantiated:

$$\textit{instantiate-partially(pattern, bindings)} \Rightarrow \textit{refined-pattern}$$

Constraints in XMF are currently written against a JavaScript API that exposes functionality for matching, instantiating and querying. Putting a syntactic layer over the API could simplify constraint definition and remains future work.

4 USING XMF TO TEACH ADVANCED LDSE CONCEPTS

We introduce XMF to students not only as a tool for creating, relating and utilizing languages but also use it as a means to explain advanced concepts and technologies in the context of language-driven software engineering, e.g. different grammar formalisms [6], meta-architectures and program transformation. This is done by analysing how XMF works and by comparing it to these related technologies and approaches. In the following, we will do this for the UML (Unified Modeling Language) meta-architecture [7], the concept of syntactic layering and for program rewriting.

4.1 Meta-Architectures

We explain the meta-architecture of XMF by comparing it to the UML meta-architecture [5]. Our goal is that students become aware of the design choices underlying the UML meta-model and their consequences. In context of class and object models, the UML meta-architecture is often described as shown in Fig. 4.

ID	Level Name	Concept
M3	Meta-Meta-Level	Meta-Class
M2	Meta-Level	Class
M1	User-Level	Class-Model
M0	Data-Level	Object-Model

Figure 4: Levels of the UML meta-model for class and object models

For a concrete example of a class "Student" and an object "Jim" we would have the concept of a Meta-Class on level M3, that of a class on level M2, a Student on M1 and an instance of Student “Jim” on M0. This is shown in Fig. 5.

M3	Meta-Class
M2	Class
M1	Student
M0	Jim

Figure 5: Class and Object example using the UML meta-model

According to the UML-Architecture, the general linguistic relationship between a meta-model and a model (M3-M2 and M2-M1) and the specific relationship between class and object model (M1-M0) is shown in the same direction. According to our experience, this can be confusing for students. In XMF we entangle this by drawing linguistic instance-of relationships vertically and general inter-model relationships horizontally as shown in Fig. 6.

M2	XPLT-Pattern	
M1	Class	Object
M0	Student	Jim

Figure 6: Class and Object example using the XMF meta-model

A fourth level is not needed for defining the XMF meta-model. As in UML, level M(n+1) defines the concepts that are used on level Mn to define concepts on level M(n-1) and so on. For practical purposes, we can restrict the architecture to three levels. In UML-terms, meta-classes are used to define meta-classes. Infinite regress is replaced by circularity that somehow has to be resolved. In XMF, patterns take the role of meta-classes and are used on M1 to define what classes and objects are. From these definitions it can, for example, be deduced that “Student” and “Jim” are valid instances of class and object. Constraints define the relationships between Class and Object. In general, M1 is the level on which users work when they define languages. XPLT is used to define class and object models. On M0 the modeling languages defined that way are used in the form of

models. The relationships between M1 and M0 are defined through patterns. In our example, "Student" must match the pattern “Class”. The pattern “Class” must match the XPLT-pattern and then we run into circularity again. Horizontal relationships are optional. One model does not necessarily have a non-linguistic relationship with another model. In the case of class and object models, the horizontal relationship on M1 is the abstract relationship "object is of type class". The instance of this relationship on M0 is, for example, the concrete relationships "Jim is an object of type Student".

XPLT can be defined in a meta-circular way. This means that there is an XPLT pattern that describes how XPLT patterns look like. Because of this we can say that M2 points to itself. This circularity on the conceptual level is resolved by using an XPLT-JavaScript implementation for the actual system. Showing the definition and implementation of XMF to students makes an intricate concept such as meta-circularity understandable.

4.2 Syntactic Layering and Views

Syntactic layering is an advanced concept in the context of language syntax. We explain it using XPLT, the pattern language of XMF. The goal is to make students aware of the layers that exist when structured languages are used, how these layers restrict language syntax and how restrictions might be broken. This leads to the general concepts of views, an important idea in LDSE.

XPLT is a language based on XML for defining languages based on XML. This means that in XMF, there is no way around the syntactic boundaries of well-formed XML. All languages and views are XML-based. This means that every model could be viewed as being at the end of the following specialization-hierarchy:

Level	Description	Restriction
R3	Arbitrary Representation	limited only by machine
R2	Unicode String	Unicode character set
R1	Well-formed XML	XML Grammar
R0	Well formed language L	L XPLT Pattern

Figure 7: Syntactic layers for a language L defined with XPLT

Every level defines a set of possible representations. For every level, the representation on Rn is a subset of the representations on R(n+1). If we describe XMF using this architecture, we are concerned with the restriction from R1 to R0. An XPLT pattern answers the following question: How to restrict well-formed XML to the subset of XML that is a particular modeling language? As transformations are computed from patterns, patterns are restricted to valid XML, and there is no transformation without a pattern, all data is always on Level R0. Interestingly, the variables in XPLT patterns do not mean "anything", they mean "anything that is well formed XML". That is, variables are implicitly referring to R1. The following pattern makes R0=R1:

```
<pattern id="norestrictions">
  $anything
</pattern>
```

If all models in XMF are restricted to level R0 in this way, how is it then possible to view the data in ways that are outside of the restrictions imposed on Level R1? We have to bring the system to display the data in a different way. Some XML-Editors for example can display XML as nested boxes. However, such a mapping is defined for level R1 and thus for all kinds of XML-Strings. What we need is a way to define individual representations of data on level R0. Instead of an R1 definition such as "every element in an XML-Document is displayed as a box." we need R0 definitions such as "every class in a class model is displayed as in a UML-style box".

As we can only achieve this on level R0 and the maximum flexibility we have is R0=R1, we must have a process that can read a well-formed XML description and display it in the desired view. We can then translate our class models into the XML-based language that this process understands. In the case of

XMF there are two View-Processes for displaying models: The first one is the Code Mirror Firefox-plugin that displays all models as strings with syntax highlighting. The second one is the HTML rendering component of the browser that can render the full range of XHTML. In order to define a view for a class model we can define a pattern that complies with XHTML.

It should be noted here that with XMF we take the display processes and the languages they understand as fix. This means that we cannot change the internals of these processes and thus have to transform our modeling language to the language of the display process. The benefit is that we have an explicit description of how the data is displayed in terms of the display processes language. The drawback is that the view is volatile to changes within the display process.

Intermediate languages can be added between the processing components and the modeling language. For example, the following pattern abstracts from the XHTML table that is used to display a class by introducing the concept of an editable box ("ebox"):

```
<pattern id="classView">
  <ebox>
    <headgroup><sub><elem>$cname</elem></sub></headgroup>
    <group name="attributes">
      <repeat container="%attr"><pref id="attrView" /></repeat></group>
    <group name="methods">
      <repeat container="%methods"><pref id="methodView" /></repeat>
    </group>
  </ebox>
</pattern>
```

Typically, users do not only wish to display models according to a view definition, but also edit them using the view. In this case, the view process must map changes on the display back to the XML-based description of the view data. This view language description can then be mapped back to the modeling language description.

4.3 Program Rewriting, Lisp and Macros

Part of our curriculum is Scheme [8], a programming language of the Lisp family [9]. Comparing XMF to Lisp is valuable because there is a deep relationship: Lisp programs are written using symbolic expressions (s-expressions) in the same way that XMF models are written using XML. The special nature of Lisp can thus be explained by referring to the syntactic layering as defined in the previous section.

XPLT patterns can be compared to declarative macro systems used in some Lisp dialects. Indeed, the macro system of Scheme, especially its notion of ellipsis to express repetition of structural patterns was highly influential on the design of XPLT. The application of Lisp macros corresponds to model transformations in XPLT. Implementing a language in Lisp using declarative macros is similar to defining a view in XMF: A new syntax for the language is defined using a pattern and the relationship between the new language and an underlying language is established by a pattern using the same variables. The concept of pattern matching and instantiation that is underlying XMF is found in many areas of informatics and is the basis of several program rewriting systems. In our course we show examples of such systems and compare them to XMF in order to make students aware of the similarities.

5 CONCLUSIONS

XMF is a flexible tool for defining, using and relating languages. Our experience so far is that its highly interactive nature combined with the fact that it is based on familiar web technology significantly lowers the time it takes for students to successfully design, implement and use their own languages. Once students have reached this level, XMF becomes a reference point for explaining many of the principles necessary to understand language-driven software engineering. Intricate concepts such as self-referential meta-architecture and syntactic layering become much more concrete when we can discuss them based on XMF's design. The concept of editing models through views which becomes increasingly more important can be explained quite easily in XMF as it is based directly on pattern matching and instantiation.

Future work on XMF will include a language for constraints and queries that can be used instead of API calls. We plan to add an executable XML-based language so that models can be transformed into executable code by the existing transformation engine. This will also enable us to discuss the basic principle of program compilation in the XMF context.

References

[1] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons, 2006.

[2] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. SIGPLAN Notices, 35(6):26–36, 2000.

[3] Krzysztof Czarnecki. Overview of generative software development. In volume 3566 of Lecture Notes in Computer Science, pages 326–341.Springer, 2004.

[4] Tony Clark, Paul Sammut, and James Willans. Applied Metamodelling - A Foundation for Language Driven Development. Ceteva, 2008.

[5] Dominikus Herzberg, Tim Reichert, and Nick Rossiter. Towards modeling language interoperability – getting meta-level architectures right. In Rektor der Hochschule Heilbronn (Hrsg.):Forschungsbericht der Hochschule Heilbronn 2008, Hochschule Heilbronn, 2008

[6] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. ACM Transactions on Software Engineering Methodology, 14(3):331–380, 2005.

[7] Unified Modeling Language: Infrastructure, Version 2.1.2. Technical Specification, Object Management Group (OMG), November 2007.

[8] R. Kelsey, W. Clinger, and J. Rees (eds.). 5th Revised report on the algorithmic language scheme. Higher-Order and Symbolic Computation, 11(1), August 1998.

[9] Paul Graham. On Lisp. Prentice Hall, 1993.

Acknowledgements

This work was partially funded by the LARS project of the “Studienkommission für Hochschuldidaktik an Fachhochschulen in Baden Württemberg” and by the Thomas Gessmann-Stiftung.

CONCATENATIVE PROGRAMMING
An Overlooked Paradigm in Functional Programming

Dominikus Herzberg
Department of Software Engineering, Heilbronn University
Max-Planck-Str. 39, 74081 Heilbronn, Germany
herzberg@hs-heilbronn.de

Tim Reichert
School of Computing, Engineering & Information Sciences, Northumbria University
Pandon Building, Camden Street, Newcastle Upon Tyne, United Kingdom
tim.reichert@unn.ac.uk

Keywords: language-oriented programming, functional programming, concatenative languages

Abstract: Based on the state of our ongoing research into Language-Driven Software Development (LDSD) and Language-Oriented Programming (LOP) we argue that the yet relatively unknown paradigm of concatenative programming is valuable for fundamental software engineering research and might prove to be a suitable foundation for future programming. To be sound, we formally introduce Concat, our research prototype of a purely functional concatenative language. The simplicity of Concat is contrasted by its expressiveness and a richness of inspiring approaches. Concatenative languages contribute a fresh and different sight on functional programming, which might help tackle challenges in LDSD/LOP from a new viewpoint.

1 INTRODUCTION

One of our main themes of research is Language-Driven Software Development (LDSD) and Language-Oriented Programming (LOP). It is about how the creation and use of languages might help us in building and engineering complex software systems. As a matter of fact, LDSD/LOP is a growing field of interest as is manifested by the research on Domain Specific Languages (DSLs), Model-Driven Development (MDD), generative software development and software factories, to name just a few areas.

In order to experiment with language layers and domain specific specializations and to test our conceptions and hypotheses, we made use of languages which are regarded as flexible and easily adaptable. Among these languages were Lisp/Scheme, Prolog and Smalltalk. Their interactive nature and their late-binding features due to dynamic typing turned out to be helpful in the setting of a "laboratory situation" for language experimentation. Still, some experiments turned out to fail e.g. applying extreme refactoring or attempting to uncover hidden design intentions in code. We felt having something in our way; a problem we could neither clearly pinpoint nor sketch a solution for. There was something "wrong" with the languages we used.

When we made contact with so-called concatenative programming languages things began to fall into place. As a result, we developed our own concatenative language called Concat. We benefited a lot from using the concatenative paradigm and still do; our work on Concat is research in progress. Concat is a language that is "as simple as possible, but no simpler" – to paraphrase a quote attributed to Albert Einstein – but still useful and practical.

Our claim is that concatenative languages are (a) ideally suited for language experimentation and (b) worth to be applied in software engineering because of its unique features.

The features that characterize and distinguish Concat in particular and concatenative languages in general are:

- Concat is a *functional language* (no explicit states) with static types and type inference. A concatenative language can also be dynamically typed and work without type inference; some variants are also functionally impure
- Concat is a language one can interactively work with on the console; we regard *interactivity* as essential for an experimental approach to LDSD and LOP

- Concat is *homoiconic*, i.e. code can be treated as data and data as code
- Concat has a very *simple syntax*. Programs are created by concatenating words and so-called quotations, and there are just three tokens with special meaning: whitespace, [and]
- Similarily, Concat has very *simple semantics*. We distinguish the level of words and quotations from the level of functions processing stacks
- Both level of Concat maintain a relationship called a *homomorphism*; that means that there is a structure preserving mapping from the word/ quotation level to the function/stack level and vice versa

There are immediate implications that follow from these characteristics: (1) Concat has a sound mathematical foundation, which enables formal treatment and reasoning over programs. (2) There are no variable bindings in Concat, that means there are no structural ties beyond the homomorphism mentioned. And that has two other important consequences especially for code engineering: (3) Concat supports macros out of the box without further ado. (4) One can cut out any fragment of code at whitespaces. Presumed that you leave the code within squared brackets intact, any such fragment still represents a valid program. This is something, which is impossible in, say, Java, C#, Lisp or Haskell. Concat enables code reuse and refactoring of code to an extent unknown in other languages.

We think that Concat offers many interesting properties. We formally define Concat in Sec. 3 after we have briefly touched upon related work in Sec. 2. We hold the view that concatenative languages deserve much more attention than is the case. They are inspiring, usable and practical despite and because of their simplicity, see Sec. 4 – a position surely debatable. We draw some conclusions in Sec. 5.

2 RELATED WORK

Much of the foundational work on concatenative languages was done by Manfred von Thun in conjunction with the development of the Joy language.[1] Today, several implementations of concatenative languages exist. Cat is a purely functional language that unlike Joy and like Concat supports static type checking.[2] Factor is a programming language designed for use in practice. It has a concatenative core and supports object-oriented programming.[3]

[1] http://www.latrobe.edu.au/philosophy/phimvt

[2] http://www.cat-language.com

[3] http://factorcode.org

Concatenative languages are closely related to stack-based languages.[4] The former are characterized by the homomorphic relationship between words/ quotations and function, the latter by the use of a stack as the central concept in the execution model. A language may be both stack-based and concatenative, but this must not necessarily be the case. Forth (Rather et al., 1996) and PostScript (Adobe Systems Inc., 1999) are popular "high-level" stack-based languages that are not concatenative. Several assembly and intermediate languages also use a stack-based model of execution.

In a concatenative language, even those words that may intuitively be perceived as data, for example numbers and strings, denote functions. Thus, concatenative languages are not only functional in the sense that functions have no side effects, but also in the sense that "everything is a function". This form of purity and the non-existence of variables relates them closely to function-level programming as defined in (Backus, 1978) and the point-free style of functional programming (Gibbons, 1999).

3 FORMAL FOUNDATIONS

In this section we will define the concatenative language Concat. Due to space limitations we restrict our presentation to a dynamically typed version of Concat. Actually, Concat is statically typed enabling the programmer to define arbitrary types as encodings.

A specialty of concatenative languages is that there is the level of words and quotations (Sec. 3.1 and 3.2) and the level of functions and stacks (Sec. 3.3 and 3.4). Both levels have their own concepts and their own semantics. However, the levels are constructed in such a way that there is a close relationship between the two (Sec. 3.5).

3.1 Words and Quotations

On the level of words and quotations Concat is defined by only a handful of concepts: words, vocabularies, quotations, concatenation and substitution. The stack pool is just defined for convenience purposes on the word level, but it is needed later on on the function level.

Definition 3.1 (Vocabulary of Words) *A* vocabulary *is a set of elements* $V = \{w_0, w_1, \ldots, w_n\} \cup \{Id\}$*; its elements are called* words*. The word Id is the* identity word*.*

A quotation is recursively defined as:

[4] http://concatenative.org

Definition 3.2 (Quotation) *Let* [ ] *be the* empty quotation. *Given a vocabulary V of words, a* quotation *q using V is a finite sequence of elements written as $q = [s_1\ s_2\ s_3\ \ldots]$ with each element being either a word of V or a quotation using V including the empty quotation.*

Definition 3.3 (Stack Pool) *Given a vocabulary V, the* stack pool *S_V is the set of all possible quotations using V.*

A program in a concatenative language is a sequence – a concatenation – of words and quotations, respectively.

Definition 3.4 (Concatenation) *Given a vocabulary V, the binary operation $\oplus$ defines the* concatenation *of two words or quotations or combinations thereof: $\oplus : (V \cup S_V) \times (V \cup S_V) \rightarrow (V \cup S_V)$. The following properties hold with $w, w', w'' \in (V \cup S_V)$:*

$$w \oplus Id \Leftrightarrow Id \oplus w \Leftrightarrow w$$

$$(w \oplus w') \oplus w'' \Leftrightarrow w \oplus (w' \oplus w'') \Leftrightarrow w \oplus w' \oplus w''$$

The first property declares the identity word as the neutral element of concatenation, the second property is the law of associativity. Concatenation constitutes a monoid.

For the sake of a simpler notation, we replace the concatenation operator $\oplus$ by a whitespace character and do not use parentheses, since they are unnecessary because of associativity.

Definition 3.5 (Substitution Rule) *Given a vocabulary V, a* substitution rule *r is a unique mapping from one concatenation to another concatenation: $r : (V \cup S_V)^n \rightarrow (V \cup S_V)^m$ with $m, n \in \mathbb{N} \backslash \{0\}$.*

Now we have everything together to define a generic substitution system that rewrites concatenations. The execution semantics are fairly simple.

Definition 3.6 (Substitution Evaluation) *Given a sequence of substitution rules and a concatenation of words and/or quotations,* substitution evaluation *is defined as follows: walk through the sequence of substitution rules, rewrite the concatenation if there is a match (probing from right to left!) and repeat this process until no more substitution rules apply.*

Before we advance to the level of functions, we would like to provide some simple examples of substitution rules. The attentive reader might notice that the rules look very much like operators written in postfix position. This is for a good reason, which will become clear when we talk about the connection to the function level. The rightmost position on the left-hand side of a substitution rule almost always is a word. Substitutions essentially dispatch from right to left.

3.2 Examples of Substitution Rules

Substitution rules have a left-hand side (LHS) and a right-hand side (RHS). Inside substitution rules, capital letters prefixed by a `$`, `#` or `@` denote variables used for matching words and quotations on the LHS and for value replacement on the RHS. If prefixed by `$`, the variable matches a single word only. If prefixed by `#`, a single word or quotation is matched. If prefixed by `@`, any number of words or quotations is matched.

The following rule defines a swap operation. Remember that the concatenation operator $\oplus$ is replaced by whitespace for improved readability:

```
#X #Y swap ==> #Y #X
```

On the LHS `#X` and `#Y` match the two words or quotations preceding `swap` in a given concatenation. On the RHS, the recognized words or quotations are inserted in their corresponding places. Take for example the concatenation "`2 [ 3 4 ] swap`", which is resolved by applying the above rule to "`[ 3 4 ] 2`".

The following substitution rules might help get an idea how simple but powerful the substitution system is.

```
[ @REST #TOP ] call ==> @REST #TOP
[ @X ] [ @Y ] append ==> [ @X @Y ]
true [ @TRUE ] [ @FALSE ] if ==> [ @TRUE ] call
false [ @TRUE ] [ @FALSE ] if ==> [ @FALSE ] call
#X dup ==> #X #X
```

The first rule, `call`, calls a quotation by dequoting it i.e. by releasing the content of the quotation. Syntactically, this is achieved by removing the squared brackets. The second rule, `append`, takes two quotations (including empty quotations) and appends their content in a new quotation. The third and fourth rule define the behavior of `if`. If there is a `true` followed by two quotations, the quotation for truth is called, if `false` matches, the failure quotation is called. Apparently, quotations can be used to defer execution. The last rule simply duplicates a word or quotation.

Due to space limitations we cannot show nor prove that some few substitution rules suffice to have a Turing complete rewriting system. Substitution rules play the role macros have in other languages such as Lisp or Scheme.

3.3 Functions and Stacks

The concepts on the level of functions and stacks parallel the concepts on the word/quotation level. On the one hand there are words, quotations and concatenations, on the other hand there are functions, quotation functions and function compositions. The identity word is mapped by the identity function. We will discuss this structural similarity in Sec. 3.5.

Definition 3.7 (Pool of Stack Functions) *Given a stack pool S_V, a pool of* stack functions $\mathcal{F}(S_V, S_V)$ *is the set of all stack functions $f : S_V \to S_V$.*

Definition 3.8 (Quotation Function) *Given a quotation $q \in S_V$, the corresponding* quotation function $f_q \in \mathcal{F}(S_V, S_V)$ *is defined to be*

$$f_q(s) \to s \oplus q \oplus \text{append} \quad \forall s \in S_V$$

A function that throws its representation as a word onto a stack is called *constructor function* or *constructor* for short.

Definition 3.9 (Function Composition) *Given a stack pool S_V, the* composition *of two functions $f, g \in \mathcal{F}(S_V, S_V)$ with $f : A \to B$, $g : B \to C$ and $A, B, C \subseteq S_V$ is defined by the* composite function *$g \circ f : A \to C$. The following properties hold with $f, g, h \in \mathcal{F}(S_V, S_V)$:*

$$f \circ Id \Leftrightarrow Id \circ f \Leftrightarrow f$$

$$(h \circ g) \circ f \Leftrightarrow h \circ (g \circ f) \Leftrightarrow h \circ g \circ f$$

That means that Id is the neutral element of function composition and that function composition is associative. Function composition constitutes a monoid as well.

We use the same identifier for the identity word and the identity function. It should always be clear from the context, whether Id is a word or a function. From the above definition of function composition we can deduce the definition of the identity function:

Definition 3.10 (Identity Function) *For any vocabulary V, the* identity function *$Id \in \mathcal{F}(S_V, S_V)$ is defined as $Id(s) \to s$ for all $s \in S_V$.*

We can now compute results with a given composition of stack functions and quotation functions.

Definition 3.11 (Function Evaluation) *Given a stack pool S_V, the* evaluation *of a function $f \in \mathcal{F}(S_V, S_V)$ with $f : A \to B$ and $A, B \subseteq S_V$ is defined to be the application of f on some $s \in A$: $f(s)$.*

3.4 Examples of Function Definitions

In the context of functions, we refer to quotations as *stacks*. A function in Concat expects a stack and returns a stack. It will take some values from the input stack, do some computation and possibly leave some results on the input stack to be returned.

Function definitions look like substitution rules. The rightmost position on the LHS is a word denoting the name of a function. Everything else that follows to the left are pattern matchers picking up words and quotations from the stack. The position next to the function name stands for the top of the stack, then comes the position underneath etc.

```
$X $Y + ==> #<(+ $X $Y)>#
```

In the example, `$Y` expects a word on top of the stack and picks it up; `$X` picks up the word underneath. If there are more words or quotations on the stack, they are left untouched. The RHS of a function definition says that the top two values on the input stack are replaced by a single new word or quotation, which is the result of some computation enclosed in `#<` and `>#`.

Within these delimiters a computation can be specified in any suitable language; we use Scheme in this example. Before the computation is executed, the items picked up by `$X` and `$Y` are filled into the template at the corresponding places.

A function definition can also be read as having a so-called *stack effect* (and so can substitution rules): The function `+` takes two words from the stack and pushes a single word onto the stack. Looking at stack effects helps in selecting functions that fit for function composition. A function or composite function cannot consume more items from a stack than there are.

If a suitable language for specifying computations is used, function composition can be directly implemented by combining and rewriting the templates. That is one reason why we have chosen Scheme as the specification language for functions. To provide an example, take the definition to compute the inverse of a number:

```
$X inverse ==> #<(/ 1 $X)>#
```

The concatenation "`+ inverse`" can – on the functional level – be automatically derived as a composite function:

```
$X1 $X2 + inverse ==> #<(/ 1 (+ $X1 $X2))>#
```

Here, `$X1` and `$X2` are automatically generated by Concat. If template rewriting is too complicated to achieve in another target language, the behavioral effect of function composition can be simulated by passing a stack step by step from one function to another.

Any of the above substitution rules (Sec. 3.2) can also be defined as function definitions. One example, although rather trivial, demonstrates this for `dup`. The two values pushed onto the stack are "computed" on the function level.

```
#X dup ==> #< #X ># #< #X >#
```

3.5 Connecting the Levels

The previous section already indicates that the level of words and quotations and the level of functions and stacks are connected. As a matter of fact,

we established a bijective mapping between words and functions, quotations and quotation functions, and between concatenation and function composition. Mathematically speaking, this is called a *homomorphism*.

There is a subtle detail. The homomorphism implies that the search strategy looking for substitution matches must scan a concatenation from right to left. On each word or quotation we look through the list of substitution rules top down for a match. After a successful substitution has occurred, the search might continue to the left or start over again at the right. The first way (continuing) has the same effect, function composition has. The second way (starting over) is equivalent to passing a stack from function to function i.e. without really making use of function composition. Either way, the lookup for matching substitutions has to restart top down again.

When writing programs in Concat, we have the choice to either define substitutions that work on a purely syntactical level by rewriting concatenations of words and quotations. Or we define functions whose operational behavior is outside the reach of Concat – it is done in another computational world Concat has only an interface with but no more. For Concat, functions and composite functions are black boxes. Interestingly, we can seamlessly combine substitution and function evaluation.

The two approaches to interpret a given concatenation lead to two different readings. Take the following example, a simple addition:

```
3 0 +
```

Notationally, all there is are words and quotations. Assumed that there is the substitution rule "`$X 0 + ==> $X`", the result on the word/quotation level is mechanically retrieved as `3`. On the level of functions, all there is are functions and stacks. So `3` is a constructor function that takes a stack and pushes a unique representation of itself – the word(!) `3` – onto the stack and returns the changed stack. So does `0`. Taken together with the function `+`, function composition results in a function accepting some stack and leaving `3` on top. Only if we suppress function composition, we see a stack being passed from function to function. This is also called *trace mode*.

A slightly more complicated example is the following concatenation:

```
6 5 dup [ 3 > ] call [ + ] [ * ] if
```

The word `dup` duplicates `5` and `call` unquotes `[ 3 > ]`, leading to `6 5 5 3 > [ + ] [ * ] if`. Assumed that a function definition for > (greater than) is given, `5 3 >` results in `true`. Now `if` rewrites the concatenation to `6 5 [ + ] call`. The result of `6 5 +` is `11`.

4 THINKING CONCATENATIVE

The following subsections aim to inspire the reader of the richness that lurks behind the concatenative paradigm. We barely scratch the surface on a subject worth further investigation.

4.1 Pattern Recognition Agents

The input to Concat can be viewed as a static but possibly very long sequence of words and quotations. Substitution rules and function definitions could be viewed as agents working on the input. Each agent has some sensors that allow the agent to recognize a set of specific subsequences of words/quotations somewhere in a program. If a pattern is recognized, the agent takes the input sensed, transforms it into a new sequence of words and quotations and replaces the input by the transformation.

Essentially, there is a pool of agents ready to process any subsequence they find a match for. This model has some similarities with biochemical processes. Let us take protein biosynthesis in a cell of a living organism as an example. After a copy of the DNA has been created (transcription), complex organic molecules called ribosomes scan the code of the DNA copy in a way comparable to pattern matching. The ribosomes read a series of codons as an instruction of how to make a protein out of an sequence of amino acids (translation). These processes could be understood as numerous computing agents working together.

It is an interesting observation that Concat can be used in a way that is close to how nature works in creating complex living systems. This might inspire a lot of interesting and interdisciplinary research questions. Also cognitive processes rely very much on pattern recognition.

4.2 Stream Processing

Another, dynamic view is to regard the input to Concat being continuously filled with new words and quotations at the outmost left and added to the bottom(!) of the stack, respectively. A continuous stream of words and quotations flows in. With a certain lookahead Concat applies substitution rules and function definitions as usual. Any context information needed for processing the stream must be either left on top of the stack. Or we introduce a "meta-stack", with the stream-stack being on top, so that context information can be left somewhere else on the meta-stack.

We have built a system called channel/filter/rule (CFR) for advanced protocol analysis in computer

networks. The incoming stream of data stems from a recording or a live trace of a monitoring device intercepting the communication of two or more interacting parties. Stateless selection (filters) and stateful processing (rules) help in abstracting and extracting information that represent the information flow on the next protocol layer (channel). We suspect that such a stream processing system can be easily realized with Concat. We have not done a prototype, yet. This is research in progress.

4.3 Refinement & Process Descriptions

Refinement is a very important notion in computer science, especially in the formal and theoretical branch. As a matter of fact, refinement is a well-understood concept. The formalization of refinement dates back to the 1970s. Refinement is a means to reduce underspecification. A specification S_2 is said to refine the behavior of a specification S_1 if for each input the output of S_2 is also an output of S_1. Semantically, this notion is captured by logical implication. The denotation $[\![S_2]\!]$ implies $[\![S_1]\!]$.

Programming languages typically do not support refinement. However, it is trivial to provide refinement in Concat, because it is built in: the notion of refinement is captured by unidirectional substitution "==>".

Some researchers bring the notion of refinement and software development processes explicitly together; one prominent example is FOCUS (Broy and Stølen, 2001). In a yet unpublished paper we show that it is straight forward to formally describe development processes in Concat using refinement. We believe Concat to be well-suited for process modeling.

4.4 Flexibility & Expressiveness

Another area for which we cannot take credit for is a demonstration of the extreme flexibility of concatenative languages – this is best shown by pointing to Factor, a modern concatenative language implementation. Factor is dynamically typed and functionally impure (for practical reasons) though functional programming is a natural style in Factor. Its dominating programming model is a stack being passed from one function to the next.

Factor is powered by a kernel written in C of about 10.000 lines of code. Everything else is written in Factor itself. Factor's syntax is extensible, it has macros, continuations and a powerful collection library.

Factor comes with an object system with inheritance, generic functions, predicate dispatch and mixins – it is implemented in Factor. Lexical variables and closures are implemented as a loadable library – in Factor. An optimizing compiler outputs efficient machine code – the compiler is written in Factor. Bootstrapping the system helps all libraries in Factor benefit from such optimizations. Right now, Factor supports a number of OS/CPU combinations among which are Windows, MacOS and Linux for x86 and PowerPC processors.

Programs in Factor are extremely short and compact. Refactoring programs in Factor is easy as is in any concatenative language: any fragment of words can be factored out – hence the name "Factor". These features have helped the developers continuously improve the code base and its libraries. Factor outperforms other scripting languages like Ruby, Groovy or Python not only in runtime but also in the number of features supported by the language.

We are confident that we can achieve a similar level of flexibility, expressiveness and performance with Concat.

5 CONCLUSIONS

Remarkably, the definition of Concat fits on a single page of paper (Sec. 3.1/3.3). Yet, the concatenative paradigm shows a lot of interesting features and inspiring approaches (Sec. 4). We barely scratched the surface of a subject worth further investigation and research. We think that concatenative programming is a much overlooked paradigm that deserves wider recognition.

REFERENCES

Adobe Systems Inc. (1999). *PostScript language reference (3rd ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Backus, J. (1978). Can programming be liberated from the von Neumann style?: A functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641.

Broy, M. and Stølen, K. (2001). *Specification and Development of Interactive Systems:* FOCUS *on Streams, Interfaces, and Refinement*. Springer.

Gibbons, J. (1999). A pointless derivation of radix sort. *J. Funct. Program.*, 9(3):339–346.

Rather, E. D., Colburn, D. R., and Moore, C. H. (1996). The evolution of Forth. *History of programming languages*, II:625–670.

S

SOFTWARE ENGINEERING FOR TELECOMMUNICATIONS SYSTEMS

INTRODUCTION

Since the construction of the worldwide telephone network started more than a century ago, advances in communication systems and their widespread availability have been a source of profound change in societies and are an important part of what we call the "information society." Today, communication systems allow people to talk, see, and exchange data with each other almost independently of their physical location in the world. In the so-called developed countries, telephones are in virtually every household, cell phones are omnipresent, and half a billion computers are part of the Internet—together, these technologies form a gigantic network that allows anyone easy access to an enormous amount of information and to communicate easily with each other.

According to a broad definition, any system that makes communication over long distances (tele = distant) possible is a telecommunication system. Historically however, the term refers primarily to telephony networks for fixed and mobile communication. For the Internet and other interconnections of computers, the term computer network is used. The term communication system can refer to both computer networks and telecommunication systems. Because of the convergence of telecommunication and Internet technology, the once sharp line between computer networks and telecommunication systems has, however, been blurred in recent years. End users engaged in distant communication are indifferent as to whether their voice is transported via traditional telecommunication networks or over the Internet—as long as the service preserves high-quality demands expected from telecommunication systems: lost calls, too much delay in voice transmission, echoes, and so on are not tolerated.

Many definitions for software engineering exist in the literature. One definition that we feel is particularly suitable in the context of telecommunication systems is "the application of engineering to software" (1). In fact, (electrical) engineers built the first telecommunication systems. The key challenges for software engineering telecommunication systems developed from several aspects that make telecommunication systems special. These aspects relate to the system in general and the software components in particular and can be subdivided into industry practices, general technical challenges, and quality demands. Important industry practices in the telecommunications domain include the definitions of standards and protocols, the layering of systems, and intensive testing. General technical challenges develop mainly from the distribution aspect of telecommunication systems and the large amount of communication that these systems need to handle simultaneously in real time. For example, modern switching systems can handle several ten thousands of calls simultaneously. Thereby, high-quality demands must be fulfilled.

From a user's perspective, telecommunication systems must provide a high quality of service, namely the fulfillment of real-time and lossless requirements. In addition to that, they must satisfy high availability, reliability, and robustness demands. For example, delays greater than a tenth of a second or lost words in a telephone conversation are unacceptable; the expectation of users is that the telephone system "always works" and that especially emergency calls always go through, no matter the amount traffic. From the perspective of software engineers, telecommunication systems must be scalable, extensible, and portable. Scalability means that the code basis can be used for different traffic demands. For example, it is desirable that the same switching software can be used in a system that handles an average of 1000 simultaneous calls as well as in a system that handles 100,000 calls simultaneously. This consistency is especially important as the demand for telecommunication services is steadily increasing. Extensibility is important because the services that must be provided by telecommunication systems are constantly subject to enhancements. Portability has to do with the long lifetime of telecommunication software and the rapid advances in hardware platforms; one cannot afford to throw away the software developed for two decades just because of a switch to new hardware.

The next section provides an overview of relevant historic developments and crucial design decisions that led to today's telecommunication systems; important background information for understanding what makes software engineering for telecommunication systems special is given. The section entitled System Design in the Larger describes fundamental telecommunication concepts that are encountered when designing systems in the large. The basic notions of distribution and communication, as well as layering, planes and resource control are discussed. Systems design in the small is the topic of the section that follows. The last section surveys a selection of the literature on modeling telecommunication systems and introduces the Real-Time Object-Oriented Modeling ROOM language as an example of a modeling language for telecommunication systems.

BACKGROUND AND RATIONALE

Modern telecommunication systems were developed in the early 1960s. The new technology of computer control, called stored program control (SPC), started to substitute electromechanical systems (2). One of the main advantages introducing SPC was flexible systems, in which additions and changes could be introduced primarily through program modifications rather than through changes in the hardware (3). However, by the late 1960s, it was time for a review. At Ericsson, one had learned that the current generation of SPC, as it existed in the late 1960s, was expensive and way

too complex, with hindsight, for widespread use, except, to some extent, in the American Bell companies. The disadvantages were above all in the high costs of handling—design, testing, modification, fault-correction, production, installation, and operation and maintenance (4). What was needed was a new approach to structure and organize these complex systems. With the engineering techniques available at that time—"Structured Programming" is in the air (5), the principle of functional modularity was a promising approach. Within Ericsson, it was IVAR JACOBSON who made the important contribution of the "block concept" in 1967 (6), which included the structuring of the system into self-contained functional modules (blocks), with all interworking between blocks performed by software signals (7). The development of Ericsson's AXE switching system was based on these principles; it went into trial service late in 1976 and became and still is one of the most successful switching systems worldwide (4).

Hand in hand with this development, the study of new languages was initiated. The industry was in need of languages highly adapted to the demands of programming and designing telecommunication systems. The outcome of these efforts were Specification and Description Language (8) (SDL), Message Sequence Chart, (9) (MSC), CHILL CCITT High Level Language (CHILL) (10), and Man-Machine Language (MML) (11). All three languages have been standardized by Consultatif International de Télégraphique et Téléphonique) (CCITT) and are still in use today. In the early 1980s, SDL and MSCs were intended for system specification and design, CHILL for detailed design, coding and testing, MML primarily for operation and maintenance. Especially for coding, many companies developed their own variant of a programming language. For example, Ericsson developed Programming Language for EXchanges (PLEX) (7), Northern Telecom Procedure Oriented Type Enforcing Language (PROTEL) (12), both languages are block structured. More recently, new languages and paradigms have become part of the toolset of software engineers in the telecommunication domain. An example of a modern programming language for telecommunication systems is Erlang (13); Erlang can be classified as a functional programming language. It was developed at the Ericsson Computer Science Laboratory in the late 1980s and was released as open source in 1998. It has been used in industrial projects for the production of highly reliable and fault-tolerant telecommunication systems. For example, Ericsson's AXD301 switching system handles 30–40 million calls per week and node, and its reliability is measured at 31 milliseconds downtime per year. It contains 1.7 million lines of Erlang code (14).

In 1994, the ROOM language appeared. ROOM blends object-oriented and real-time concepts and techniques and is thus particularly well suited for modeling telecommunication systems. Elements of ROOM were added to the Unified Modeling Language (UML) (15,16) version 2.0 that was released in 2004. In 2006, the Object Management Group (OMG) released the specification for the Systems Modeling Language (SysML) (17), which is a modeling language for systems engineering that seems to be a promising addition to the toolset of software engineers in the telecommunications domain. A good chance exists that telecommunication systems engineering might benefit from the recent research and commercial interest in generative (18) and model-driven development (19). Domain-specific notations have been used by telecommunication engineers for a long time, and new technologies might enable the generation of systems based on descriptions using these notations.

The complexity of switching systems by sheer size of code is impressive. Already around 1980, several hundred programmers had produced over one million lines of code over a five-year period for the DMS-100 switching system family of Northern Telecom. This company represents over 15,000 procedures in 1500 modules (12). The systems of today are even more complex. A code base of several million lines of code is not unusual. Still, these systems fulfill high-quality demands on availability, reliability, fault tolerance, and so on. Such systems can be upgraded and maintained while being in operation! A downtime of some few minutes per year is already perceived as "bad quality."

Considering their complexity, it may come as no surprise that *architecture* is and always has been an important issue in telecommunication systems design. Architecture is and was a means to deal with complexity. Of course, the term "architecture" was not defined clearly, but it is absolutely in line with the design paradigm of the 1970s: The modularization of a system is regarded as its architecture. Architectures were not modeled, as we tend to say today, but rather described either informally, usually in some sort of box-line diagrams, or formally with Simple Declarative Language (SDL). It is interesting to read which design conceptions were identified for new software architectures in the 1980s: independent subsystems for call control (features), signaling, and hardware control; data abstractions partitioned for each subsystem; formal communication protocols; concurrent and asynchronous operation of each subsystem; terminal-oriented control; layered virtual machines; FSM Specifications; application programs; and systems programs (20)—the topicality of the list is astonishing.

Before "software engineering" was an established field, telecommunication engineers had already established a discipline of engineering highly reliable, scalable, and robust real-time systems, which are open and standardized—and it included software. When the telecommunication engineers included programmable devices into their systems, they integrated these devices into a hardware-driven environment. Thus, they applied a lot of their hardware principles to software. In effect, they made it transparent to the system, whether an entity is realized in hardware or software. Simply speaking, the software was and still is developed as seriously and effortful like hardware. Because failures and downtimes of telecommunication components are not an option, much energy is put into the design and architecture of those systems. The engineering aspect of the software part of telecommunication systems resembles many qualities of systems engineering: standardized interfaces, message-orientation, cascading, and composition as main design principles; exhaustive testing routines including load and stress testing, configuration management, process driven development—to name just a few—are best practices in the telecommunication domain.

The concepts, techniques, and requirements we describe in the historic overview above are still relevant to software engineering for telecommunication systems today. It is the way telecommunication engineers design their systems in the large and in the small that is special. That is why we put our focus on these two topics in the following sections. Other software engineering issues like requirements engineering and traceability, configuration and product management, software product lines and families, testing, project management and so on do not differ that much from software development practices in other domains such as large enterprise information systems.

Regarding software engineering for telecommunication systems, no established body of literature exists yet, which reflects a commonly agreed viewpoint on how telecommunication systems are (to be) designed in the large and in the small. However, if you spend some years in the telecommunication industry among systems designers and software developers and study existing publications, then you will notice that they somehow speak "one language" and design their software in similar ways. This article is an attempt to uncover the elements of design of telecommunication system engineers to provide valuable input for the interested reader. A more elaborated version of the systematics presented here can be found in Ref. 21.

SYSTEMS DESIGN IN THE LARGE

In telecommunications, systems design in the large must deal with the notion of distribution, layering, planes, and resource control. We will discuss issue in turn.

Distribution

A telecommunication system is made up of entities like switching systems, radio base stations, and mobile phones. These entities are physically distributed in space; they are either located in a fixed place (like switching systems) or are mobile (like mobile phones). These entities collaborate with each other to provide a service to end users. Thus, the most obvious characteristic of a communication system is its aspect of distribution. If two or more devices, processes, users or—more abstractly—entities are physically spread in space but want to collaborate, they somehow have to bridge spatial distribution and establish communication. We will give rather informal definitions of the concepts related to distribution in the following section. Formal definitions of these concepts can be found in Ref. 22.

Communication. "It is all about communication"—this slogan characterizes concisely the motto of telecommunications. We can classify three types of communication used in telecommunications. The classification scheme bases on the question "Who controls whom?" We can distinguish three basic combinations of the exertion of control between two communicating parties: (*1*) no side exerts control, that is no side has a state model of the other side to influence the other side's behavior in a controlled way, which we call *data-oriented* communication; (*2*) only one side exerts control, which we call *control-oriented* communication; (*3*) both sides exert control, which we call *protocol-oriented* communication.

Any communication type can be realized in a connection-oriented mode, a connectionless mode, or even other kinds of communications styles. We will come back to this in the discussion of communication services. Note that communication in telecommunications is message-oriented and that communication relations are strictly specified in form of protocols.

Decomposition and Remote Communication. What is distribution? With the eyes of software engineers, we tackle the notion of distribution in two steps: First, distribution is an issue of logical *decomposition*. Second, we need to consider the effects of *remote communication*.

We can view a telecommunication system as a logical entity that encapsulates some functionality and offers interfaces (often called "ports") for message-based communication with the environment. We assume that the behavior of the system is given, meaning that we know the set of allowed messages per interface, their format, how the system reacts on messages delivered to the interfaces and which messages it emits to the environment. The behavior of a telecommunication system is often said to provide services to its environment, usually its end users.

A first step toward distribution is that the entity under consideration can be logically split up ("decomposed") into separate parts, each part representing a new entity. The parts also communicate to each other via messages through their interfaces. The interfaces are connected via so-called channels, which are sometimes also called connectors. A channel is an idealized communication medium, which transfers messages faultless and in an instant. In other words, a logical entity gets refined by a network of separated but cooperating parts. From an outer perspective, the conglomerate of parts preserves the behavior that can be experienced at the interfaces of the single entity. The decomposition process is recursive.

The second step is to take into account that the communication over a channel is not ideal but suffers from the real-world effects of remote communication. When the decomposed parts get spread over, say, hosts or physical nodes, they require some sort of communication means to bridge the spatial separation. The interaction of the decomposed parts in a distribution network is *not* fault-free per se; it is sensitive to disturbances on the communication medium and dependent on the properties of the connection. We condense the whole communication medium in a model of a nonideal channel, which we call complex connector. The complex connector is a component that represents the properties of the communication channel and its effects on the transmission of messages. These properties are called QoS attributes and include all relevant characteristics, such as reliability, throughput, jitter, and delay (21,22).

To summarize: A telecommunication system is a distributed system. To its end users, a telecommunication system appears as a single, coherent, service-provisioning system. As a matter of fact, the system is decomposed into a number of physically separated but interacting parts, called nodes, which constitute a communication network.

The effects of remote communication are captured by the notion of a complex connector.

Network Topology and Communication Services. Readers might be familiar with the fact that communication systems are composed of a stack of layers. We will come back to layering in a subsequent article. In this section we view each layer in a communication system as a self-contained unit without any dependencies to other layers. Each layer unit consists of distributed entities communicating remotely to each other via a network that interconnects the entities. In short, we treat a layer as a distributed network in its own right.

Here, we are concerned with what kind of communication services and communication resources the distributed entities use to bridge their spatial distance; for the time being we are not interested in how it is achieved via a lower layer. That means our understanding of a network is an abstract model of distribution, which includes a *network topology* (who is permitted to communicate with whom) and the used *communication services*. A communication service can offer connection-oriented or connectionless communication means.

Connection-Oriented Communication Services. With the help of the complex connector, we can describe *static* configurations of distant connection-oriented communication. The complex connector concentrates all impacts that the transmission may have on the messages to be conveyed. In reality, connections are rarely static; they are rather a form of a long-lasting, dynamically created connection. Normally, connections are set-up and released on demand. Thus, we need something; we can ask for inserting and removing a complex connector between any two ports at some point in time. The connection-oriented communication service fulfills this role. In a telecommunication system, circuit switching is a connection-oriented communication services.

Connectionless Communication Services. A style of communication exists that requires no connection. Instead, messages include the address of the receiver. The sender hands the message over to a connectionless communication service, which distributes the message according to the address to a destination. If the sender wants to get a response on a delivered message from the receiver, the then sender has to include its source address in the message as well. In a telecommunication system, packet switching is a connectionless communication service.

Addressing. Addressing is crucial for communication systems in general and telecommunication systems in particular. In the following, we focus on addressing in the context of telecommunication systems. Generally speaking, an *address* denotes a concept to identify and locate objects in a defined scope. The scope is the so-called *address space* (or name space), which is an assembly of addresses with each address being unique in the assembly. An *address association* relates two addresses to each other; the association is directed pointing from one address (the source address) to another address (the destination address). Source and destination address must not belong to the same address spaces; we must make a difference between external address associations and internal address associations. External address associations relate addresses of different address spaces, internal address associations relate addresses of the same address space.

For example, the difference between connection-oriented communication and connectionless communication is basically different ways of working with address spaces. In connection-oriented communication, communication interfaces are associated with a fixed (better: temporarily fixed) communication partner. Information that is pushed to the interface is conveyed to the communication partner; reversely, information the communication partner wants us to notice, pops up at the interface. In that sense, the interface is a sort of representation of the other party, and the interface identifier is an internal address denoting the other party. So, talking to another party must either use another interface (that is bound to the other party) or newly bind the interface with the other communication party.

For connectionless communication, the general addressing structure looks different. The arrangement of associations is so that two communication partners do not maintain direct relations between their address spaces. Instead, local addresses are associated to a third party, which is an external address space. Consequently, users who communicate connectionless need to have an internal representation of the address space outside their locally addressable scope. They need to specify the destination of their messages. Users who communicate connection-oriented do not have to do that.

Remarks. Definitions on distribution to be found in literature suffer preciseness on the one hand and generality on the other hand. We think that at an abstract-level distribution is primarily a logical conception and that it is adequate to give a formal definition based on a proper model. Secondary, distribution has a technical dimension. A formal definition of distribution is given in Ref. 22.

No notion exists of a complex connector in open systems interconnection (OSI), but in practice, telecommunication engineers work with this concept. As an evidence for that statement, have a look at channel substructures in SDL (23, p.121), which basically captures the same intention as complex connectors.

Addressing is a delicate issue in modeling and an neglected issue in software engineering.

Layering

Layering is one of the oldest techniques in software engineering to structure a system. Possibly the first, who made systematically use of layering was Dijkstra, he used layering for the design of the THE operating system (24). Layering is also a key structuring principle in the design of communications systems, be they telecommunications or computer networks. In the previous section, we intentionally left out the issue of layering. We simply said that one can look at each layer of a distributed communication system individually. Now it is time to explain, how several

layers of communication networks are interconnected and make up a layered system.

In the next section, we will briefly outline the seven layer reference model (RM) of open systems interconnection(25) (OSI). We will then distill the key idea that underlies layering. This progression will naturally lead us to two viewpoints one can have on a communication network: a network-centric or a node-centric perspective. Finally, we discuss the concept of planes.

The OSI Reference Model. Layering is a means to stepwise provide higher-level services to a user or the next "upper" layer, and to separate levels of services by precisely defined interfaces. This overall design principle is reflected by the use of protocol stacks. The OSI RM is the most prominent framework for a layered communication architecture. We do not repeat OSI RM to the full extent, we just would like to remind the reader of the basic outlook, see Fig. 1: Several network layers are stacked on each other, each layer realizing a complete network of its own. Higher-layer network services rely on lower layer services until a physical layer is reached. Additional introductory information can be either retrieved from the X-Series of the ITU-T recommendations or from textbooks. Almost any textbook on computer networks and/or data communications gives an introduction into OSI RM, for example Ref. 26.

We wish to highlight one important point. OSI RM clearly distinguishes two communication relations: layer-to-layer ("vertical") communication from peer-to-peer ("horizontal") communication. "Vertical" communication refers to the exchange of information between layers (that is levels of services usually within the same physical entity) in the form of *Service Data Units* (SDU). "Horizontal" communication refers to the exchange of information between remote peers. Remote peers are physically distributed and communicate with each other according to a protocol in the form of protocol messages, also called *Protocol Data Units* (PDU), thereby sharing the same level of protocol conventions. PDUs are the vehicles for SDUs. A single SDU may be packaged into one or more PDUs. Such PDUs are also called *data* PDUs. Nondata PDUs are called *control* PDUs. In a multilayer communication architecture, a service provisioning layer becomes the service user of the next lower layer.

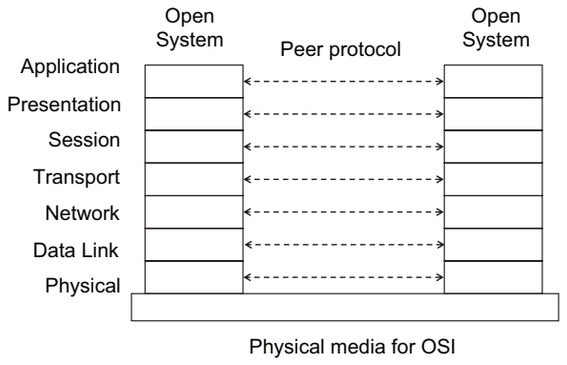

Figure 1. OSI seven layer reference model; see Ref. 25, p. 31.

Communication Refinement. To understand how two different networks of service levels are connected through layering, one has to know that Fig. 1 unveils only half the truth. The dotted lines labeled with "Peer protocol" do not represent protocol relations only. Each double-headed arrow "hides" a complete infrastructure of a communication service for this specific layer. The communication service per layer is an abstract model of the style of communication (connection-oriented, connectionless), properties (delay, reliability, etc.), topology, and addressing schema. This abstract model can be refined into a more concrete model, which is in effect the next lower layer of the protocol stack. The next lower layer includes the communicating entities (the boxes next to the double-headed arrow) and the communication service of that layer. In essence, layering is the result of refining communication services; we call this *communication refinement*.

Communication refinement leads to two different viewpoints on distributed layered communication systems. Both viewpoints are important for systems modeling in software.

Network-Centric Viewpoint. If we regard the communication service as an abstract model of the means of communications, suppressing all the details of lower layers, then we just observe a network of communicating entities of one layer using the communication service. This view is the network-centric viewpoint on communication systems. This view Let's one to look onto a network as a distributed system ignoring layering. We made use of this technique in the section about distribution.

Node-Centric Viewpoint. If all communication services are resolved by a refinement, which represents the next lower layer, we end up with a situation similar to Fig. 1: We have a communication service at the very bottom, a physical media, which cannot be resolved more. The pile of boxes labeled "Open System" on the left and on the right represent the entities that, together, make up the software or hardware, which resides on a physical node in a network. This view is the node-centric viewpoint on communication systems.

Planes

One concept that has turned out to be extremely useful is the concept of *planes*. The concept was introduced in Integrated Services Digital Network (ISDN) (27), taken over in Global System for Mobile communication (28), and currently shapes the network architecture of Universal Mobile Telecommunication System (UMTS) (29). The distinction is usually in three planes, namely the control plane, the user plane, and the management plane.

A plane encapsulates service functionality and may have internally a layered (protocol) structure. Planes are an organizational means on top of layering and communication refinement, respectively. In telecommunications, the user plane provides for user information flow transfer (data PDUs), along with associated controls (e.g., flow control, recovery from errors); the control plane performs call and connection control functions (control PDUs), dealing with the necessary signaling to set up, supervise, and

release calls and connections; the management plane takes care of (*1*) plane management functions related to the system as a whole including plane coordination and (*2*) functions related to resources and parameters residing in the layers of the control and/or user plane (30).

OSI RM is not prepared to handle planes (nor is the Internet architecture), which is also one of its major deficiencies. The control and user plane are not separated. In software engineering, the organization of a system in planes is almost unknown. On a case-by-case basis, designers had and still have to invent individual solutions to handle planes in their models. For example, in ISDN the engineers introduced a synchronization and coordination function (SCF) as a major component of the management plane. The SCF is connected to the highest layer of the user plane and to the highest layer of the control plane to coordinate and synchronize the required collaboration of planes (27).

Resource Control

A field that is largely ignored in computer networks but is of importance in telecommunications is the issue of resource control—most popular textbooks on computer networks and distributed systems do not touch on the subject at all. The term *resource* does not only include physical resources such as adaptors, switchboards, echo cancellers, codec converters, and so on, but also resources implemented in software. On a software level, resources can be combined, added by some functionality, and offer value added services that make a user believe to access a "new" kind of resource that is more than the sum of its physical components. Take for example an alarm clock and a radio, add a composing layer, and you will get a clock radio. The new feature, that the radio turns on at a certain alarm time, is more than any of the resources could provide in isolation.

We recognize a need to pay some special attention to resource control. As was mentioned previously, telecommunication systems are sliced in a control and a user plane; basically, it is the control plane that controls the user plane. In most cases this control relationship breaks down to resource control. The control plane controls resources of the user plane. Although the control plane and the user plane may operate as largely independent networks, the combining spots are locations of resource control. Usually, the node hosting the resource brings together the control and the user plane. Traditionally, the aspect of resource control has been a local, internal issue. Often, inside such a node, the border between controlling and controlled behavior is blurred and not fully separable. At best, the designers define a proprietary application programming interface (API) for the resource.

One of the intentions of UMTS has been to clearly separate the control and the user plane and to avoid the blur of the control\slash user plane inside nodes hosting resources; this idea is the so-called *architectural split* introduced with UMTS. As a result of that, the telecommunication sector of the international telecommunication union defined a protocol, a control-oriented protocol in our terminology, that describes how a user can control a switching center. This protocol is called media gateway control protocol, it is specified in H.248 (31) and has been taken over as a standard by IETF as well, see RFC 3015 (32). With the definition of a protocol and the separation in a resource user and a resource provider all prerequisites are given to aim for physical separation of both roles. In the UMTS architecture, these two roles are logically fulfilled by the media gateway controller and the media gateway. It is up to a manufacturer to produce two individual nodes or a single combined node. Important is that the distinction has been made logically.

Remarks

We mentioned the OSI RM. The Reference Model of the Internet Architecture, is related loosely to OSI RM, see Ref. 26. Other frameworks, which address the topic of distributed communication system and propose a terminology, a set of conceptions, and a system architecture organization. The most important frameworks to mention are the reference model for open distributed processing (33,34), the telecommunications information networking architecture (35), and the object management architecture (36,37), (38,39) which is the basis for the Common Object Request Broker Architecture. Basically, all three frameworks specify an environment to develop, install, and maintain distributed applications.

SYSTEMS DESIGN IN THE SMALL

When it comes to systems design in the small, the most apparent issue a software engineer is confronted with is that telecommunication systems are real-time systems. An understanding of real-time systems and a suitable approach for designing such systems is required. The use and the understanding of the term "real-time system" is not consistent in the literature. It is a mixture of characterizing attributes and structural properties of a system. For example: On one hand, it is said that a real-time system fulfills timing constraints, (i.e., a real-time system has to react to a stimulus in a certain time frame); in this example the guaranteed response time is an attribute, which characterizes a real-time system. On the other hand, real-time systems are often classified as "embedded systems." An embedded system is recognized as a specific part of a larger system, which is a structural aspect. Besides this lack of clarity in terminology, there is not even common agreement on the word "real-time." The following paragraphs summarize findings from studying the literature.

What is a Real-Time System?

Real-time systems are defined as those systems in which the correctness of the system depends not only on the logical result of computation, but also on the time in which the results are produced (40). After more than a decade this definition still seems to be the greatest common denominator. Here, "real-time" is an attribute to "system." Because of their specific field of application, additional attributes are usually associated with real-time systems. Included in this category are e.g., reliability, fault tolerance, adaptability, and speed (41).

Hard versus Soft Real-Time

The most popular classification is the distinction in *hard* and *soft* real-time systems. Hard real-time systems are under deadline constraints. Passing a deadline is considered unacceptable. A soft real-time system retains some tasks that are still valuable for execution even if they miss their deadlines (41). Several (42) telephony systems belong to the class of soft real-time systems: Passing of deadlines is accepted as long as the number of failures is below a defined threshold. Although this explanation might be true in general, some components in telecommunication networks, have to fulfill hard real-time constraints. For example, the time delay perceived as acceptable for voice transmission in a speech conversation places tough time limitations on a mobile phone for speech encoding- and decoding including cyphering and channel coding.

Rough Structure

A very rudimentary structure of the basic elements of a real-time system is given by Ref. 42: It consists of hardware, *sensors* and *effectors*, the environment, and software. The sensors and effectors interact with the environment; the software controls the actions of the hardware via a hardware interface. A similar description using different terminology can be found in Ref. 43: A real-time system consists of a controlling and a controlled system. The controlling system interacts with its environment using information about the environment available from various sensors and activating elements in the environment through "actuators." The controlled system can be viewed as the environment with which the computer interacts.

A loose reasoning describes why timing aspects and structural issues of a real-time system are related: Timing correctness requirements develop because of the physical impact of the controlling systems' activities on its environment. That means that the environment needs to be monitored periodically and sensed information needs to be processed in time (41). This finding implies that we have to distinguish the environment from a controlling part, and detecting and acting devices are needed.

What is an Embedded System?

The definition of an embedded system is vague; it mainly describes a structural aspect. In its most general form, an embedded system is simply a computer system hidden in a technical product (44). A more concrete definition is that most embedded systems consist of a small microcontroller, and limited software situated within (e.g., an automobile or a video recorder) (45). Three issues seem to be important here: (*1*) size matters, (*2*) an embedded system is part of a technical system, and (*3*) it serves the purpose of the technical system and not vice versa. Issue (*3*) especially helps delimitate nonembedded systems from embedded systems. A Personal Computer (PC) for instance is a general purpose computing machine, the software and the central processing unit (CPU) are an integral part of it. This eliminates a PC from being an embedded system. A counterexample might be a mobile phone. The digital signal processing chip and its software serve a single purpose: to offer phone functionality. Embedded systems may or may not have real-time constraints (43), but many real-time systems are embedded systems (45).

To summarize: The special character of systems, that have a physical impact on the "real" world by means of reactiveness is most significantly described by the requirement on the timing constraints to be met by the system. Such systems are called *real-time systems*. Additional properties, which reflect other aspects of the physical impact character, include reliability, fault tolerance, stability, safety and so on. As yet no commonly agreed list of properties exists that constitutes is a real-time system. Moreover, the physical impact nature of such systems implies a rough structure: a controlling part interacting with the environment (the controlled part) through sensors and effectors. The hardware mediates between the sensors\slash effectors and the software of the system. Many real-time systems are embedded systems, which means they serve a specific purpose in a technical system, which is actually the case for all nodes in a telecommunication system.

Despite these various aspects of real-time systems and partly confusing definitions from the literature, designing real-time systems is a well-established domain. When designing telecommunication systems in the small, it becomes obvious that just a few key design concepts are required, such as active objects for modeling threads and message-orientation for modeling asynchronous communication. Interestingly, these concepts can also be used for systems design in the large. This statement means if carefully selected, then one can use the same language for both systems design in the small and in the large. In the section on Real-time object-oriented modeling, we describe a language that can be used for both tasks.

MODELING TELECOMMUNICATION SYSTEMS

In this section, we describe different modeling approaches by surveying the available literature. We then go on to describe ROOM, which is a language in widespread use in the telecommunications domain. ROOM can be used for both systems design in the large and in the small.

Modeling Approaches

Since the UML has been standardized by the object management group (OMG) and published in many books, *modeling* is on everybody's lips. Also, the importance of the architecture level in software systems is more and more respected, see for example OMG's initiative on Model Driven Architecture (46). However, when it comes to modeling telecommunication systems the fundus of literature is even smaller.

In Some books, the object-oriented paradigm has been used to model communication systems. One example is Object-Oriented Networks: Models for Architecture, Operations, and Management (47). The book uses not only conventional object-oriented modeling concepts but also advanced concepts from specialization theory. The syntax used to capture the semantics of models is the Abstract Syntax Notation One (see Ref. 48). The author develops a classification scheme adapted to the needs of

communication networks that enables a designer to develop understandable and meaningful object and class diagrams. The approach is descriptive and the techniques presented seem to be suited for modeling product architectures. The risk is that given "facts" are just schematically modeled (it is relatively easy to note down an object diagram for almost anything) without any reflection about the actual functioning and the actual meaning for the architecture.

Another example is Object-Oriented Network Protocols" (49). The book's intention is to provide a foundation for the object-oriented design and implementation of network communication protocols. Although modeling of communication systems is not the topic of the book, it is worth to have a look at the modular communication system framework developed by the author. It gives an insight how protocols could be modeled and that object-orientation is a practical approach in protocol design.

A completely different approach is taken by "Modeling Telecom Networks and Systems Architecture: Conceptual Tools and Formal Methods" (50). This book condenses more than 20 years of experience gained on the subject within Ericsson. It presents a method and a language for modeling telecommunication system and is based on the processing system paradigm (51). The whole field of communication systems is covered, and a stringent methodology and classification scheme is discussed. The interested reader might also look at Ref. 52.

Real-Time Object-Oriented Modeling

Subsequently, we briefly present the ROOM language to give the reader a notion of what kind of concepts software engineers in the telecommunication system domain work with. Even though the publication of ROOM dates back to 1994, it is still modern and a rare example of a well-documented design language, see Ref. 42. Many features of the ROOM language have been incorporated into the UML (15,16). Nonetheless, we have chosen to describe ROOM, because it represents a coherent set of features required for designing (embedded) real-time systems in the telecommunication domain; the UML is just a rich set of modeling concepts a designer can choose from. In the following section, we will briefly discuss structural elements of ROOM, behavioral elements and mention model execution.

Structural Elements. ***Actor, Port, Message, Protocol.*** The ROOM language is built on the notion of an *actor*. An actor represents a physical device or a software unit; it is a sort of active object that clearly separates its internals from the environment. Everything inside the actor, meaning the actor's structure and behavior, is not visible to the environment. Only at distinct points of interaction, so-called *ports*, the actor interfaces the environment. A port is somewhat comparable to an interface as known for example, in the UML but the comparison blurs two important facts. First, ports in ROOM are not method interfaces but message interfaces. A *message* consists of a message name, priority, and data. Messages may be incoming and/or outgoing at a port (the direction is always defined from the viewpoint of the actor). So, ports are message exchange points between the actor and its environment. Secondly, a port is not only an interface that tells the environment how to use the actor but also is a definition of the actor's expectations on the environment. Therefore, a *protocol* is always associated with a port, which defines the set of incoming and outgoing messages that may pass the port. An actor is specified by means of an *actor class*. An actor class is symbolized by a rectangular box with a thick black border. A port is figured by a small squared box that appears on the border of an actor class symbol. An example is shown in Fig. 2.

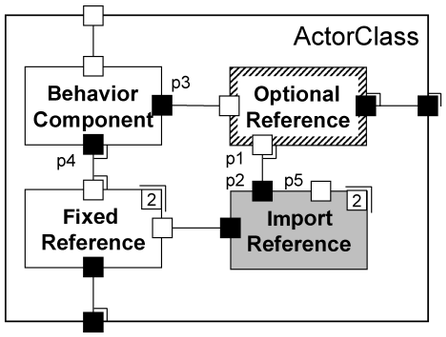

Figure 2. Actor class containing all types of actor references.

Actor References. An actor can be composed of other actors. In ROOM, references describe compositions. That means, an actor class specification may reference zero or more other actor class specifications. Such a reference is called *actor reference*; it is a way to include other actors into the name space and life-time context of an actor. Per actor reference, a *replication factor* determines the maximum number of valid actors of the referenced actor class that can be put in context. By default, the replication factor is set to one. The following types of references can be distinguished: an actor reference may be fixed, optional, imported or substitutable. These types specify run-time relations. For a *fixed* actor reference, actors of the referenced actor class are incarnated along with the incarnation of the composing actor. If the actor reference is declared as *optional*, then actors of the referenced actor class can be dynamically created and destroyed during the life time of the composing actor. The maximum number of allowed actors (given by the replication factor of the actor reference) may not be exceeded. If declared as *imported*, then an actor that already exists in another context of another composing actor is plugged-in at incarnation of the composing actor. That means a single actor instance may act in two or more contexts of a composing actor: in the context of the "original" composing actor that created the actor and owns the permission to destroy it and in the context of one or more other composing actors which imported that specific actor. Imported actor references are a powerful tool to define different roles for different contexts of an actor and thereby to define patterns of collaboration. A *substitutable* actor reference means that any actor instance of that actor reference can be replaced by another actor, provided that the other actor's class specification is compatible with the referenced actor class of the actor reference. Here, compat-

ibility means that the other class specification supports at least the same set of ports (with the same message schema).

Binding, Contract. To build up complete structures of actor references, some means to interconnect their ports must exist. This connection is done by so-called *bindings*, sometimes also referred to as *connectors*. A binding connects a port of an actor reference either with the port of another actor reference or with a port of the composing actor class. Bindings define communication relationships on class level. The auxiliary concept of a *contract* consists of a binding and the two interface components (ports) that the binding connects.

Example. An example of an actor class specification that encompasses all the discussed modifications of an actor reference is shown in Fig. 2. Actor references are symbolized by a rectangular box with a thinner black border and can only appear "inside" the context (also called *decomposition frame*) of an actor class specification. Names for actor references begin with a small letter. Names for bindings begin with a small letter by convention. Sometimes, to avoid visual clutter, the names of bindings and ports are not displayed in the diagram. The replication factor of a replicated actor reference is displayed inside a box in the upper right-hand corner. Optionality is indicated by stripes. If imported, the actor reference is colored grey. Substitutability is indicated by a "+" symbol in the upper left-hand corner.

The Behavior Component. A component specifies the actor class' behavior. In fact, the behavior component is invisible; the behavior component's border is colored in grey just for demonstration purposes. Thus, all ports of an actor class specification that are not connected somewhere else are actually connected to the actor's behavior component; they are called *end ports*. Otherwise, they are called *relay ports*. All other ports ((p3, p4) that "hang around" are also implicitly connected to the behavior component. *Reference ports* (the name for ports of actor references) that are not involved in a contract are actually not in use, see p5.

Layer Connection, Service Provision Point, Service Access Point. The notion of layers is a built-in concept in ROOM. Layering is a form of abstraction that is used to define "islands" of self-contained functionality that provide services to another "island" of functionality. In contrast to the *horizontal* structure of peer-to-peer communication between ports, layers represent a *vertical* organization of a system. The terms "horizontal" and "vertical" are apparently vague and indicate the difficulty for giving a precise definition of layers. Actually, the sort of interfaces used to describe layers are very close to ports. The interface of an actor that provides (layer) *services* towards another actor is called service provision point (SPP). The SPP may be replicated; the number of replications is given by a replication factor. Its counterpart, the interface that accesses services of an SPP is called service access point (SAP). SAPs can be replicated as well but they need to. The SPP and the SAP each have a protocol associated with it that determines the interface type. Similar to a binding, a SPP and a SAP are connected to each other by a *layer connection*.

Behavioral Elements. ***ROOMcharts, Scheduler.*** We already mentioned the behavior component of an actor. In ROOM, behavior is specified in form of state machines, so-called *ROOMcharts*, a variant of Harel's statechart formalism (53). Actors in ROOM are reactive objects with their own thread of execution, which is a typical characteristic for real-time systems. All incoming messages at the behavior component are *events* that may trigger a *transition* to leave a *state*, of perform some *action* and enter the same or another state. For a state, entry and exit actions can be specified. Actions are specified in a detail level language such as C, C++, or Java. A *guard* (a boolean condition) can be attached to a transition, which that prevents the transition from firing if the condition evaluates to false. The concept of *composite states* enables the modeler to nest states within states. Once all actions have been executed (ROOM follows the "run-to-completion" processing model), the actor "falls asleep" waiting for additional events to process. Because incoming events are queued, the actor may immediately become busy again until the event queue is empty. Events can also be deferred (i.e., the processing is postponed). Message priorities change the order of event processing usually to "the more important, the more up front in the event queue." In principle, the scheduling semantics of the *scheduler* can be adapted to any other scheme. ROOM flexible in that respect to cover a wide range of real-time applications. For example, time-based scheduling ("the more urgent, the more up front in the queue") may be an alternative.

Data Classes. Complex data structures can be modeled using the concept of *data classes*. Data classes correspond to traditional classes: they define data and methods that operate on them. In contrast to actors, data objects do not have their own thread of control; they are extended state variables that are encapsulated within the actor and are accessible by the behavior component. Typically, data classes are based on classes provided by the detail-level programming language. That means within an actor the modeler can use and stick to a traditional object-oriented design paradigm. In addition to their role as variables, data classes are used to define the data carried in messages. Remember that a message consists of a name, a priority, and data, or more precisely, a data object. This single data object is an instance of a predefined or user defined data class. The basic requirement put on data objects is that they must be serializable for message transfer by the ROOM virtual machine.

Model Execution. In principle, two possible methods exist to execute ROOM models: (*1*) the model is accompanied by an interpreter called the *ROOM virtual machine*, which is a hypothetical platform implemented in software that interprets ROOM models; and (*2*) the elements of the model are mapped to their functional equivalents in the target environment, which usually is a real-time operating system.

BIBLIOGRAPHY

1. IEEE Standard Glossary of Software Engineering Terminology. Standard 610. 12-1990, Piscataway, NJ: IEEE Standards, 1990.
2. J. Meurling and R. Jeans, *A Switch in Time—An Engineer's Tale*, Chicago, IL, Telephony Publishing Corp., 1985.
3. F. S. Viglinate, Fundamentals of stored program control of telephone switching systems. *Proceedings of the 1964 19th ACM National Conference*, 1964, pp. 142.201–142.206.
4. J. Meurling and R. Jeans, *The Ericsson Chronicle: 125 Years in Telecommunications*, Stockholm, Sweden: Införmationsförlaget Heimdahls, 2000.
5. O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*, New York: Academic Press, 1972.
6. I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard, *Object-Oriented Software Engineering*, Reading, MA: Addison-Wesley, 1992.
7. D. Herzberg, UML-RT as a candidate for modeling embedded real-time systems in the telecommunication domain, in R. France and B. Rumpe, (eds.), *UML '99 —The Unified Modeling Language: Beyond the Standard; Second International Conference, Fort Collins, CO*, 1999, LNCS 1723, Springer, 1999, pp. 330–338.
8. Specification and Description Language (SDL), ITU-T Recommendation Z.100, International Telecommunication Union, November 1999.
9. Message Sequence Chart (MSC), ITU-T Recommendation Z.120, International Telecommunication Union, November 1999.
10. CCITT High Level Programming Language (CHILL), ITU-T Recommendation Z.200, International Telecommunication Union, October 1996.
11. Introduction to the CCITT Man-Machine Language, ITU-T Recommendation Z.301, International Telecommunication Union, November 1988.
12. B. K. Penny and J. W. J. Williams, The software architecture for a large telephone switch, *IEEE Trans. Communicat. Communicat. Software*, COM-**30**(6): 105–114, 1982.
13. J. Armstrong, *Programming Erlang: Software for a Concurrent World*, Raleigh, NC: The Pragmatic Programmers, 2007.
14. J. Armstrong, Concurrency oriented programming in erlang, *Proc. of the German Unix User Group's Frühjährsfachgespräch (FFG)*, 2003.
15. Unified Modeling Language: Superstructure, Version 2.1.1, Technical Specification, Object Management Group (OMG), February 2007.
16. Unified Modeling Language: Infrastructure, Version 2.1.1, Technical Specification, Object Management Group (OMG), February 2007.
17. OMG Systems Modeling Language (OMG SysML) Version 1.0, Technical Specification, Object Management Group (OMG), September 2007.
18. K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, New York: ACM Press/ Addison-Wesley Publishing Co., 2000.
19. T. Stahl and M. Völter, *Model-Driven Software Development*, London: John Wiley & Sons, 2006.
20. D. A. Lawson, A new software architecture for switching systems, *IEEE Trans. Commun. Communication Software*, COM-**30**(6): 17–25, 1982.
21. D. Herzberg, Modeling telecommunication systems: From standards to system architectures, PhD thesis, Aachen University of Technology, Department of Computer Science III, 2003.
22. D. Herzberg and M. Broy, Modeling layered distributed communication systems, *Formal Aspects Comput.*, **17**(1): 1–18, 2005.
23. J. Ellsberger, D. Hogrefe, and A. Sarma, *SDL—Formal Object-oriented Language for Communicating Systems*, London: Prentice Hall, 1997.
24. E. W. Dijkstra, The structure of the "THE"-multiprogramming system. *Commun. ACM*, **11**(5): 341–346, 1968.
25. Information Technology—Open Systems Interconnection—Basic Reference Model: The Basic Model, ITU-T Recommendation X.200, International Telecommunication Union, July 1994.
26. A. S. Tanenbaum, *Computer Networks*, 4th edition, Upper Saddle River, NJ: Prentice Hall PTR, 2003.
27. ISDN Protocol Reference Model, ITU-T Recommendation I.320, International Telecommunication Union, November 1993.
28. J. Eberspacher and H.-J. Vögel, *GSM—Switching, Services and Protocols*, New York: Wiley, 1998.
29. B. Walke, M. P. Althoff, and P. Seidenberg, *UMTS—Ein Kurs*, J. Schlembach Fachverlag, 2001.
30. B-ISDN Protocol Reference Model and its Application, ITU-T Recommendation I.321, International Telecommunication Union, April 1991.
31. Gateway Control Protocol, ITU-T Recommendation H.248, International Telecommunication Union, June 2000.
32. F. Cuervo, N. Greene, C. Huitema, A. Rayhan, B. Rosen, and J. Segers, Megaco Protocol Version 1.0. Standard RFC 3015, Internet Engineering Task Force, November 2000.
33. Information Technology—Open Distributed Processing—Reference model: Overview. ITU-T Recommendation X.901, International Telecommunication Union, 1997.
34. J. R. Putman, *Architecting with RM-ODP*, Englewood Cliffs, NJ: Prentice Hall, 2001.
35. M. Chapman and S. Montesi, Overall Concepts and Principles of TINA—Version 1.0, Tina baseline, TINA-C, February 1995.
36. R. M. Soley and C. M. Stone, Object Management Architecture Guide—Revision 3.0. Document ab/97-05-05, Object Management Group (OMG), June 1995.
37. R. M. Soley and C. M. Stone, *Object Management Architecture Guide*, 3rd edition, New York: Wiley, 1995.
38. T. J. Mowbray and W. A. Ruh, *Inside CORBA: Distributed Object Standards and Applications*, Reading, MA: Addison-Wesley, 1997.
39. Common Object Request Broker Architecture: Core Specification—Version 3.0. Specification formal/2002-11-03, Object Management Group (OMG), November 2002.
40. J. Stankovic, Misconceptions about real-time computing: A serious problem for next generation systems, *IEEE Comput.*, **21**(10): 10–19, 1988.
41. A. B. Tucker, Real-time and embedded systems, in *The Computer Science and Engineering Handbook*, Boca Raton, FL: CRC Press, 1997, pp. 1709–1724.
42. B. Selic, G. Gullekson, and P. T. Ward, *Real-Time Object-Oriented Modeling*, New York: John Wiley & Sons, Inc., 1994.
43. J. A. Stankovic, Real-time and embedded systems, *ACM Comput. Surv.*, **28**(1): 205–208, 1996.
44. D. E. Simon, *An Embedded Software Primer*, Reading, MA: Addison-Wesley, 1999.

45. J. A. Stankovic et al. Strategic directions in real-time and embedded systems. *ACM Comput. Surv.*, **28**(4): 751–763, 1996.
46. J. Miller and J. Mukerji, Model driven architecture (MDA). Technical Description ormsc/2001-07-01, Object Management Group (OMG), 2001.
47. S. Bapat, *Object-Oriented Networks—Models for Architecture, Operations, and Management*. Englewood Cliffs, NJ: Prentice Hall, 1994.
48. J. Larmouth, *ASN.1 Complete*, San Francisco, CA: Morgan Kaufmann, 1999.
49. S. Boecking, *Object-Oriented Network Protocols.*, Reading, MA: Addison-Wesley, 2000.
50. T. Muth, *Modeling Telecom Networks and Systems Architecture: Conceptual Tools and Formal Methods*, Berlin: Springer, 2001.
51. T. Muth, D. Herzberg, and J. Larsen, A fresh view on model-based systems engineering: The processing system paradigm, in *Proc. of the 11th Annual International Symposium of The International Council on Systems Engineering (INCOSE 2001)*; Melbourne, Australia, 2001.
52. T. Muth, *Functional Structures in Networks: AMLn—A Language for Model Driven Development of Telecom Systems*, Berlin: Springer, 2005.
53. D. Harel, Statecharts: A visual formalism for complex systems, *Sci. Comp. Prog.*, pages 231–274, 1987.

DOMINIKUS HERZBERG,
TIM REICHERT
Department of Software Engineering, Heilbronn University, Germany

Towards Modeling Language Interoperability: Getting Meta-Level Architectures right

Dominikus Herzberg and Tim Reichert
Department of Software Engineering
Heilbronn University
74081 Heilbronn, Germany

herzberg|reichert@hs-heilbronn.de

Nick Rossiter
School of Computing, Engineering & Information Sciences
Northumbria University
Newcastle Upon Tyne, United Kingdom

nick.rossiter@unn.ac.uk

Abstract: The growing interest in Domain Specific Modeling (DSM) languages and the increasing demand for model driven approaches (like OMG's vision of a Model Driven Architecture, MDA) suggest that modeling languages should strive towards interoperability. In short, modelers could benefit from a symbiotic coexistence of DSM and GPM (General Purpose Modeling) languages, like the UML. Key to modeling language interoperability is the underlying meta-level architecture, which we studied exemplarily on UML. First, we unveil a design flaw in the presentation of UML's meta-level architecture. Second, formal considerations show that the meta-level architecture is limited to closed interoperability, which is a constraint from which approaches such as MDA suffer.

1 Introduction

There are two main approaches to modeling in software engineering: one either makes systematic use of a General Purpose Modeling (GPM) language or one chooses to use a Domain Specific Modeling (DSM) language. Today, the Unified Modeling Language (UML) [OMG06b, OMG06c] is the most prominent GPM language used in software development favoring an object-oriented modeling paradigm. The UML is standardized by the Object Management Group (OMG), has become widely spread in academia and industry and is regarded as the *lingua franca* among software engineers. On the opposite, there are many different DSM languages, each language addressing a specific field or domain. By nature, these languages usually resist standardization because of their specialization and limited scope of use. Standardization is replaced by advocacy for DSM and tool sets to ease the creation of new DSM languages. Among others, Microsoft is a strong advocate of DSM[1].

[1] http://msdn.microsoft.com/vstudio/DSLTools

The appearance of commercial and open source DSM environments (such as MetaEdit+ from MetaCase[2] and the Eclipse Modeling Project[3] from the Eclipse Foundation) show that DSM and DSM languages are becoming more and more popular and maturing.

Domain Specific Languages (DSLs), be they classified as modeling languages or not, have always played an important role in the toolbox of software developers and engineers. In programming, some DSLs are or are almost de facto standards available in many programming languages. Take, for example, SQL (Structured Query Language) and Regular Expressions. Both languages cover highly specialized domains: SQL is for data manipulation, retrieval and creation in relational database systems, Regular Expressions are for text pattern matching. Such DSLs are often either accessible via libraries or are directly embedded in a host language. Perl and Tcl, for instance, have the syntax for Regular Expressions built into their language. Simply speaking, in programming, general purpose and domain specific languages live a symbiotic coexistence.

In modeling, the situation is similar but has not evolved to the same level of maturity. The motivation to invent or choose a DSM language is a modeler's experienced lack of "purposefulness" of a GPM language in terms of preciseness and expressiveness. A DSM language serves the purpose to have a precise (or at least a better) means of communication about models within a specialized field or domain with its own conceptions and relationships. Using a GPM language would be of no benefit, since it would neither support the notation nor the language semantics required to call for a valid and meaningful model. A DSM language incorporates domain and methodological knowledge about a subject or field, whereas a GPM language does not [Her03]. Examples of DSM languages are ERD (Entity Relationship Diagrams) [Che76] for conceptual data modeling, ROOM (Real-Time Object-Oriented Modeling) [SGW94] for architecture and real-time systems modeling and AMLn for modeling telecommunication systems [Mut05].

Like in programming, a GPM language can host a DSM language – and vice versa. Because of the level of abstraction provided by some DSM languages, it might make sense to use a DSM language, say, for high-level architecture modeling (ROOM could be taken as an example) and embed a GPM language like the UML for class modeling within the architectural units of composition of the DSM language. Similarly, a GPM language such as the UML can be used for structural modeling embedding Petri Nets, to give one example, in form of a DSM language for behavior modeling. In short, a modeler could benefit from a symbiotic coexistence of modeling languages.

The state of affairs is that modeling language interoperability is of much practical use but not common practice, mostly due to a lack of research and tool support. This comes to some surprise, because the UML is designed with a lot of foresight in this respect. The designers of the UML structured the language into several layers of design, each upper layer being a sort of tool set for the lower layer directly underneath. This design constitutes a n layered meta-model architecture, with $n = 4$ in the case of UML. Language extensions, so called *profiles*, can be plugged into the language at defined points in the meta-architecture of the UML, thereby providing maximum flexibility for language integration. For that purpose, the OMG has even created a framework for meta-data management, the Meta Object

[2] http://www.metacase.com
[3] http://www.eclipse.org/modeling

Facility (MOF) [OMG06a].

In our research, we strive for modeling language interoperability, which includes language integration – a goal of much practical value. We want to achieve two main goals: First, we would like to create an infrastructure, wherein a modeling language designer can define modeling languages and their interoperation. We aim for a formally sound approach based on a meta-layer architecture and will use Category Theory for that. Consequently, we do not rely on concrete frameworks like MOF. Second, with the help of such an infrastructure, we want to improve a modelers capabilities to reuse existing languages for certain challenges providing a seamless way of interoperation of syntax and semantics of concrete modeling languages. As an example, such features are required for a sound realization of OMG's vision of Model Driven Architecture (MDA) [OMG03]. We see these features as a crucial prerequisite for the model and meta-model transformations as sketched in [OMG03]. The novelty of our approach lies in the strict and systematic use of meta-modeling for language interoperability.

In our current state of the project, we learned that the meta-model architecture is broadly misunderstood. This misunderstanding hinders formal considerations and tool implementations. It is partly also responsible for the lack of automated tool support for UML extensions. So, first, we want to get this part right in this paper, which we value as our main contribution. Subsequently, in section 2, we will first introduce the common understanding and presentation of the meta-level architecture. After that, in section 3, we will present a more detailed and more complete view on an informal level. In section 4, we will complement our discussion by a formal view in categorical terms. Section 5 discusses related research. Finally, section 6 provides our conclusions and an outlook on further research issues.

2 Common Presentations of the Meta-Level Architecture

As if it were a confirmation of the emerging trend to take care of the design of modeling languages, some new textbooks are around, which discuss the overall architecture of the UML. Quite often, these books present the meta-level architecture (also called meta-model or meta-data architecture) more or less detailed in the spirit of Figure 1a); as an example, take the UML book by Chris Rupp et al. [JQZ+05]. There are four layers, usually labeled M0 to M3 from bottom to top, each lower layer being an instance of its direct upper layer. Figure 1a) is also the viewpoint of the current and previous versions of the UML standard, see e.g. chap. 7.10 in [OMG06b].

In 1999, one of the authors (Herzberg) had seen another depiction of the meta-level architecture, see Figure 1b) [HvW99], which left him in irritation: Why is there an "instanceOf"-arrow pointing from M0 to M2? There was no reason to identify, why this "extra" arrow was needed.

Today, some years later, it is clear, why Figure 1a) is incomplete. The additional arrow in Figure 1b) is not superfluous. On the opposite, it is needed to get the meta-level architecture right.

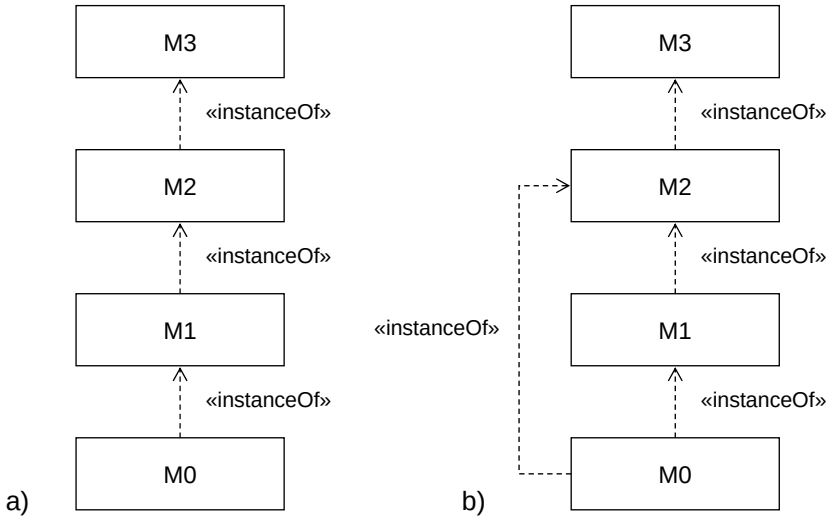

Figure 1: Two competing presentations of the 4-level meta-data architecture

3 An Informal View on the 4-Level Meta-Architecture

Let us have a closer look at the architecture of meta-levels on an informal level first. We will start at the top left, see Figure 2. To make the discussion concrete, we refer to conceptions as you know them from modeling with classes and objects. The abbreviation CD stands for Class Diagram. A class diagram consists of classes, associations, inheritance and so on. These conceptions, the conceptions you are allowed to use for a CD are defined by CM, the Class Model, as we call it. The CM is a specification of conceptions, of which a concrete CD is an instance of. That's why there is an arrow labeled with "instanceOf" between CD and CM. The Class Meta-Model (CMM) specifies the language constructs available for use on CM. In other words, the CM is one concrete specification of a modeling language, whose specification concepts are defined by CMM. In that sense, CM is an instance of CMM. All this is pretty straight forward and not in conflict with the common understanding of meta-level architectures.

However, and this is often overlooked, the same argumentation holds on the row below. An Object Diagram (OD) consists of objects, values, references etc. These conceptions are defined by the Object Model (OM). The Object Meta-Model (OMM) specifies the language constructs available for use on OM. Again, this is pretty straight and clear. Now comes the interesting part.

Of course, the world of classes and the world of objects are somehow interconnected. This interconnection is defined through a relationship between the Class Model (CM) and the Object Model (OM). This relationship determines how OD and CD are related. Objects are instances of classes, meaning that a certain class is the input to a factory, which "produces" an object, whose type property is a pointer to the class and whose values are data stores

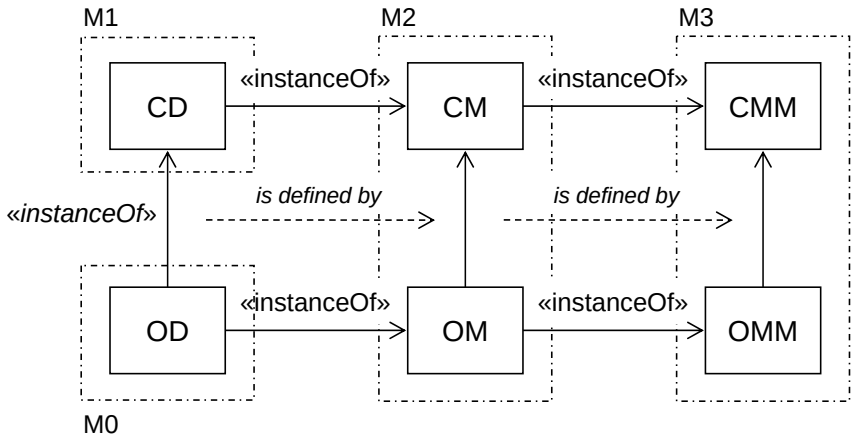

Figure 2: A detailed, but informal view on the meta-level architecture

including pointers to the class attributes. That is what we call an “instanceOf” relationship. This relationship is defined by the OM/CM arrow.

The means to describe an interconnection between OM and CM is defined by the arrow between OMM and CMM. Otherwise, one could only describe self-contained models of OM and CM, but not interrelate these models, which – in turn – would prevent to specify the “instanceOf” relationship between OD and CD. As one can observe, the reasoning is absolutely “symmetric”.

Now, let’s do some grouping. Let us refer to OD as meta-level zero (M0) and to CD as meta-level one (M1). CM and OM together constitute meta-level two (M2), CMM and OMM constitute meta-level three (M3).

We are done and have come up with a completely coherent description of the meta-level architecture, which – in principle – could be extended by further, higher levels. As you might have noticed, Figure 1b) is a condensed form of Figure 2: The arrow pointing from M2 to M3 in Figure 1b) actually comprises two arrows. Obviously, Figure 1a) is incomplete.

One might ask, why we have not grouped OD and CD in the very same way, as we have done it for OM and CM and for OMM and CMM, respectively. Such an argument would call for a three-level meta-architecture instead of a four-level meta-architecture. However, there is finer point in here. Of all arrows labeled with “instanceOf”, there is only one arrow, whose semantics can be arbitrarily defined in the meta-level architecture: it is the arrow between OD and OD, which is in fact defined on M2. In that sense, the “instanceOf” arrow between OD and CD is of a different kind than that all the other “instanceOf” arrows. Seen from this point of insight, even Figure 1b) is not 100% precise. It should point out the different quality of the “instanceOf” relation between M0 and M1.

By the way, could there be reasons to introduce higher levels in the meta-level architecture, like M4, M5 etc? Yes, there could. The arrow between OMM and CMM on M3 is a

necessity in order to have means to define the relationship between CM and OM in M2. If you want to define the relationship between OMM and CMM yourself, you need a higher level, which provides the infrastructure to do so. The four-level meta-architecture is the smallest number of levels needed by a modeling language designer in order to have means to specify M0, M1 and their interrelationship of instantiation. In practice, higher levels are possible but rarely needed.

We have a hypothesis, why Figure 1a) is such a widely spread viewpoint on the meta-level architecture of the UML and on meta-level architectures in general. M2 also covers the execution semantics of M0 (we refer now to Figure 2) including the interplay with M1 for e.g. instantiation processes. The execution semantics are a weak point of the UML and have always been a target of criticism. We speculate that because of this weakness people have become unaware of the relation between M0 and M2. Still, it is a bit worrying that experienced language designers have overseen this flaw for many years in the UML and MOF specification. Without this complete understanding, the basis for modeling language interoperability is weak as well.

4 A Formal, Categorical View on the 4-Level Meta-Architecture

Category theory [ML98] offers many facilities for representing information systems. It has been extensively applied to problems in Computer Science [BW99] and Software Engineering [Fia04]. The basic constructions of category, functor and natural transformation enable mappings to be represented between objects, categories and functors respectively. Cartesian categories enable products and exponentials (connectivity) to be represented, within the limits of initial and terminal objects. In commuting diagrams, different paths between the same two objects must yield the same result, enabling equations to be derived for equality for the composition of the arrows in each path.

For interoperability previous work has shown that the property of adjointness is particularly valuable for modelling situations where there is a relationship between levels but such relationship is not as simple as equivalence or isomorphism [RHN06]. An adjoint relationship is represented by a 4-tuple $< F, G, \eta, \epsilon >$ where F, G are functors $F : \mathbf{A} \longrightarrow \mathbf{B}$ and $G : \mathbf{B} \longrightarrow \mathbf{A}$, $\mathbf{A}$ and $\mathbf{B}$ are categories, η is the unit of adjunction and ϵ is the counit of adjunction. In an equivalence relationship, η is 0 and ϵ is 1, indicating respectively that $GF(A) = 1_A$ and $1_B = FG(B)$, where A is an object in $\mathbf{A}$ and B is an object in $\mathbf{B}$. Thus with equivalence the application of the pair of functors to an object returns the starting object.

In a more general relationship, there may be a change in state through the application of the pair of functors. This change is represented by the following two mappings:

$$\eta : 1_A \longrightarrow GF(A)$$

$$\epsilon : FG(B) \longrightarrow 1_B$$

If two functors, say F and G, are adjoint, they are written as $F \dashv G$. In such a case the application of the functors provides a unique solution for the relationship between them. Neighbouring adjoints can be composed in a natural manner. So for the functors $F : \mathbf{A} \longrightarrow \mathbf{B}$, $H : \mathbf{B} \longrightarrow \mathbf{C}$, $G : \mathbf{B} \longrightarrow \mathbf{A}$ and $I : \mathbf{C} \longrightarrow \mathbf{B}$, we can write the 4-tuple as $< HF, GI, \eta, \epsilon >$ if there is adjointness $F \dashv G$ and $H \dashv I$. The values for η and ϵ in such a case are a composite of those for each single adjunction [RHN06].

In previous work [RHN06], a four-level architecture has been developed to handle interoperability. This has assumed a relatively simple structure as shown in Figure 3 with four levels of categories connected by three levels of mappings (functors) with the relationship of adjointness holding for each pair of two-way functors and every composed pair of two-way functors.

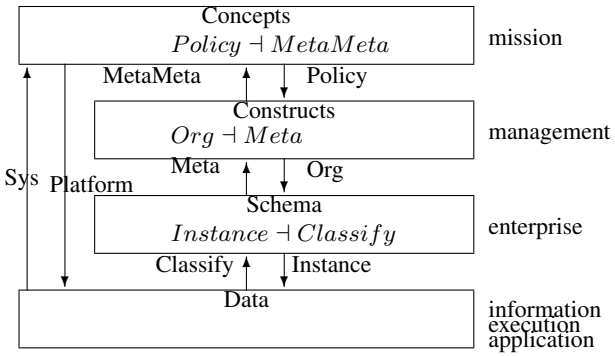

Figure 3: Interpretation of levels as natural schema in general terms

With UML the simplest categorical diagram, corresponding to Figure 1a), is shown in Figure 4a). This shows that at level M0 we have the Object Diagram (OD), at level M1 we have the Class Diagram (CD), at level M2 we have the Conceptual Model (CM) and at level M3 we have the Class Meta-Model (CMM). There is some apparent difference in the naming of the levels. OD in UML corresponds to Data in the four-levels and CD to Schema but CM does not match Constructs and CMM does not match Concepts. However, CM describes the concepts available to make an OD and on this basis seems close in purpose to Constructs, which describes the data structuring facilities available to a database designer, such as Table and Primary Key in a relational system. The content of CMM is clearly critical. If it refers to object-oriented abstractions such as inheritance, then it is similar to Concepts in the sense of Figure 3. Otherwise there is a difference here in the definition of the levels.

The functors connecting the levels in Figure 1a), $IN01 : \mathbf{M0} \longrightarrow \mathbf{M1}$, $IN12 : \mathbf{M1} \longrightarrow \mathbf{M2}$ and $IN23 : \mathbf{M2} \longrightarrow \mathbf{M3}$, correspond to *Instance of* relationships in Figure 1a) and to Classify, Meta and MetaMeta respectively in Figure 3. Categorically, Figure 1a) is not very interesting, lacking the arrows Policy, Organisation and Instance where Instance refers to Instantiation, the opposite of Instance of. The absence of two-way arrows means that adjointness cannot be tested in Figure 4a), which means that it is a much less constrained

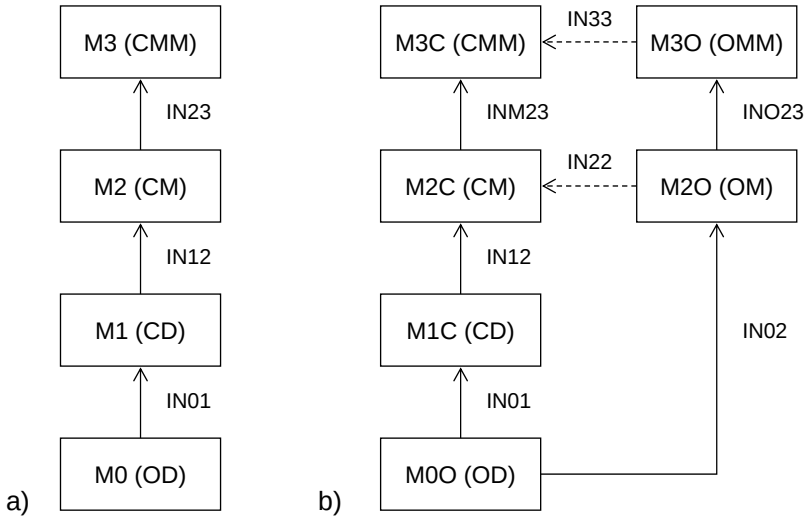

Figure 4: a) simple diagram relating OD to CMM, b) more complex diagram relating OD to CMM, with addition of OM and OMM

structure than that envisaged in Figure 3. It is though possible to compose the functors:

$$Sys = IN23 \circ IN12 \circ IN01$$

so that Sys represents the overall mapping from the object diagram to the class meta model.

Figure 4b) is the equivalent of Figure 1b). The situation here is more complex and there is more scope for useful categorical reasoning. In particular some commuting diagrams can be constructed. The relationship $IN22 : \mathbf{M20} \longrightarrow \mathbf{M2C}$ between the Object Model (OM) and the Class Model (CM) can be captured by the equation:

$$IN12 \circ IN01 = IN22 \circ IN02$$

The dashed line in the diagram in Figure 4b) for $IN22$ indicates that this arrow is not known directly, but is inferred from the commuting requirements. The relationship $IN33 : \mathbf{M30} \longrightarrow \mathbf{M3C}$ between the Object Meta Model (OMM) and the Class Meta Model (CMM) can be captured by the equation:

$$IIN33 \circ IN023 = INM23 \circ IN22$$

where again the dashed line in the diagram in Figure 4b) for $IN33$ indicates that this arrow is not known directly, but is inferred from the commuting requirements.

While the comparison of the object model with the class model in Figure 4b) does offer valuable extra insight into the interoperability potential of UML, the lack of two-way arrows (for adjointness) suggests that UML has not been designed with anything but local interoperability in mind. The object-oriented paradigm itself does have some downward arrows. For instance a constructor instantiates an object for a class in the same way as the Instance arrow in Figure 3. Imported classes (as in Java utilities) might be interpreted as constructs to be organized by the designer. At the top level, the type of object-oriented system might be declared as a Policy mapping from information system abstractions to the constructs to be made available.

5 Related Work

Since the late 1960s, the majority of the research on interoperability has been done in the area of databases. Of this work, especially the approaches which deal with language translation are relevant [CG98, ZCT91]. In comparison to our work, these approaches are focused on database languages, where for example an object oriented general purpose language hosts a domain specific database query language, while our focus is on modeling languages. More recent research on interoperability has been done in the context of the Semantic Web [BLHL01]. Here, languages that are used for modeling ontologies have to be interoperable so that distributed knowledge described with different languages can be utilized. [SLW$^+$04] is an example of a formal approach to ontology language interoperability based on lattice theory. Interoperability approaches based on category theory are model management [AB01] and the approach by Goguen [Gog05] that is based on institutions [GB92]. Model management is a generic approach to schema integration that is based on generic operators for data model translation, transformation and merging, but it does not cover language interoperability in the sense that we described above. The work by Goguen is important for interoperability between logical description languages, such as F-Logic or the family of Description Logics, but less relevant for modeling languages in general. Programming Language interoperability is a concern in approaches such as the .NET framework by Microsoft, where interoperability is achieved by translating program code to an intermediate language [BKR04]. A very similar approach has been taken for modeling languages [BM05], where the intermediate language is a hypergraph to which all schemas are translated in order to achieve interoperability. A closer investigation into the four-level-architecture of the UML and the connections between levels and models is given in [RFBLO01], where a UML virtual machine based on the UML's four levels is implemented. However, even this approach is in the realm of Figure 1a).

6 Conclusions and Future Research

In this paper, we brought forward the argument that the common understanding of meta-level architectures is oversimplifying, thereby hindering modeling language interoperabil-

ity. We believe that modeling language interoperability between GPM and DSM languages is crucial for the further development of software engineering and key to visions like MDA. We presented a detailed and consistent view on the meta-level architecture, first informally then formally. Formal considerations gave further argument that a GPM language like the UML is prepared for internal interoperability but not for external interoperability. This insight was deduced from a generic meta-level architecture for interoperability.

Our conclusion is that modeling languages need an extended meta-level architecture similar to the Natural Schema shown in Figure 3. Otherwise, approaches like the MDA remain closed and limited to the use of UML and languages integrated into the UML infrastructure only. We think that only open approaches are future-proof and stimulate further development of DSM languages.

Currently, we are investigating a very generic approach on interoperability, which includes modeling languages, database schemas and ontologies. It looks like that a unifying approach is feasible. It would help us transfer research results on e.g. database interoperability to modeling language interoperability and vice versa.

References

[AB01] S. Alagic and P. A. Bernstein. A Model Theory for Generic Schema Management. In *8th International Workshop on Databases and Programming Languages*, pages 228–246, 2001.

[BKR04] Nick Benton, Andrew Kennedy, and Claudio V. Russo. Adventures in interoperability: the SML.NET experience. In *PPDP '04: Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 215–226, 2004.

[BLHL01] Tim Berners-Lee, James Handler, and Ora Lassila. The Semantic Web. *Scientific American*, 284:34–43, 2001.

[BM05] M. Boyd and P. McBrien. Comparing and Transforming Between Data Models via an Intermediate Hypergraph Data Model. *Journal on Data Semantics*, 4:69–109, 2005.

[BW99] Michael Barr and Charles Wells. *Category Theory for Computing Science*. CRM, 3rd edition, 1999.

[CG98] Bogdan Czejdo and Le Gruenwald. Schema and Language Translation. In *Management of Heterogeneous and Autonomous Database Systems*, pages 157 – 174, 1998.

[Che76] Peter P. Chen. The Entity-Relationship Model – Towards a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[Fia04] José Luiz Fiadeiro. *Categories for Software Engineering*. Springer, 2004.

[GB92] Joseph Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the ACM*, 39:95–146, 1992.

[Gog05] Joseph A. Goguen. Data, Schema, Ontology and Logic Integration. *Logic Journal of the IGPL*, 13:685 715, 2005.

[Her03] Dominikus Herzberg. *Modeling Telecommunication Systems: From Standards to System Architectures*. PhD thesis, Aachen University of Technology, Department of Computer Science III, 2003.

[HvW99] Dominikus Herzberg and Lars von Wedel. Erweiterungsmechanismen der UML. *OBJEKTspektrum*, pages 56–59, Juli/August (4) 1999.

[JQZ+05] Mario Jeckle, Stefan Queins, Barbara Zengler, Chris Rupp, and Jürgen Hahn. *UML 2 glasklar: Praxiswissen für die UML-Modellierung und -Zertifizierung*. Hanser, 2nd edition, 2005.

[ML98] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer, New York, 2nd edition, 1998.

[Mut05] Thomas G. Muth. *Functional Structures in Networks: AMLn – A Language for Model Driven Development of Telecom Systems*. Springer, 2005.

[OMG03] MDA Guide Version 1.0.1. Technical Report, Object Management Group (OMG), June 2003.

[OMG06a] Meta Object Facility (MOF) Core Specification, Version 2.0. Technical Specification, Object Management Group (OMG), January 2006.

[OMG06b] Unified Modeling Language: Infrastructure, Version 2.1. Technical Specification, Object Management Group (OMG), April 2006.

[OMG06c] Unified Modeling Language: Superstructure, Version 2.1. Technical Specification, Object Management Group (OMG), April 2006.

[RFBLO01] Dirk Riehle, Steven Fraleigh, Dirk Bucka-Lassen, and Nosa Omorogbe. The Architecture of a UML Virtual Machine. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 327–341. ACM Press, 2001.

[RHN06] Nick Rossiter, Michael Heather, and David Nelson. A Natural Basis for Interoperability. In *Proceedings of I-ESA'06, Interoperability for Enterprise Software and Applications Conference*, LNCS. Springer, 2006.

[SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, Inc., New York, 1994.

[SLW+04] Baisheng Shi, Zongtian Liu, Yuqing Wang, Hong Yu, and Meili Huang. The lattice approach to ontology language interoperability. In *Computer and Information Technology, 2004. CIT '04. The Fourth International Conference on Computer and Information Technology*, pages 265–272, 14-16 Sept. 2004.

[ZCT91] Roberto Zicari, Stefano Ceri, and Letzita Tanca. Interoperability between a rule-based database language and an objet-oriented database language. In *First International Workshop in Multidatabase Systems*, pages 125–134, April 1991.