Mapping Data Flow Models to the Palladio Component Model

Stephan Seifermann stephan.seifermann@kit.edu
Dominik Werle dominik.werle@kit.edu
Mazen Ebada utxqw@student.kit.edu
Karlsruhe Institute of Technology (KIT)

Abstract

Predicting quality properties such as privacy are reasonable use cases for Data Flow Models (DFMs). For other use cases such as performance prediction, component-based software architecture models focusing on control flows are more suitable. Designers can derive a Control Flow Model (CFM) from a DFM but they have to make numerous design decisions like defining operation signatures. Currently, this derivation is a creative process without a clear design space and without guidelines for navigating this space. In this paper, we present design alternatives for given data flow examples and derive mapping rules that allow to choose between reasonable alternatives. Our results are a first step towards a catalogue of rules for deriving CFMs from DFMs in a systematic way and providing semi-automated transformations.

1 Introduction

Structured Design [4] uses Data Flow Models (DFMs) to bridge the gap between requirements and software design, i.e. designers derive software designs from DFMs. Many definitions of DFMs exist but always contain fundamental concepts that DeMarco [1, p. 51] describes as follows: *Sources* start a data flow and *sinks* end a data flow. *Processes* take data, transform it, and yield modified data. *Files* are temporary stores of data. *Data Flows* are pipelines transporting pieces of information (so called data) between those entities.

Besides serving as a documentation or transition model between development phases, models are valuable for quality analyses. DFMs are particularly useful for analyzing privacy as already discussed by the authors [9]. Control Flow Models (CFMs) can predict performance properties amongst other things. We see CFMs as models describing the realization of systems with respect to structure, procedures, and sequences of procedures. The Palladio approach [8] is one of these prediction approaches. Architects benefit from using models tailored for specific quality analyses that might not be feasible in one single model.

Deriving software designs from DFMs is a way of using both models together. The main source of complexity is that DFMs are analysis models lacking design information, i.e. DFMs specify what shall be done and CFMs specify how it shall be done. DFMs and CFMs must not hold contradicting information.

Several approaches for transforming DFMs into CFMs exist. However, handling ambiguities in DFMs is still difficult. For instance, the DFM to Object Oriented Design (OOD) mapping *Tad* [3] is closely related to our work but struggles in giving easy step-by-step guidelines for solving ambiguities because of its genericness. Approaches that extend DFMs avoid such ambiguities by additional specifications. In real-time systems modeling [2], additional control flow descriptions can solve ambiguities in execution orders. However, this breaks the separation of views and roles between analysis time, i.e. requirements engineering, and design time. Therefore, information from design time might not be available during analysis time. Other formal approaches like the one proposed by Larsen, Plat, and Toetenel [7] are tailored to a specific formalism, which impedes reuse. However, some introduced concepts like specifying relations between input and output data [5] are valuable. To the best of our knowledge, all approaches lack strong guidance in collecting of and deciding for design alternatives while transforming DFMs into CFMs.

In this paper, we provide an initial set of mapping guidelines that support designers in deriving a Palladio Component Model (PCM) model from a DFM. We exploit restrictions of the design space imposed by PCM to make the guidelines of *Tad* [3] more precise and better applicable to PCM. However, we do not claim this set to be complete but rather see it as a starting point for discussions and further research.

The remainder of the paper is structured as follows: In Section 2, we describe DFMs requirements and a minimal extension in order to specify the relations between input and output data in an unambiguous way. We explain our running example in Section 3. In Section 4, we describe the mapping guidelines. We conclude the paper in Section 5.

2 Data Flow Model Requirements

We formulate requirements for DFMs to allow the definition of useful mapping guidelines. In general, we require properly modeled DFMs as defined by DeMarco [1, part 2]. This means that a DFM only consists of sources, sinks, processes, files, and data flows. It is beneficial if the modeler applied the refinement mechanism of DFMs that allows to use another DFM to describe a process or set of processes in more detail

while maintaining the incoming and outgoing data flows. DeMarco calls this leveled DFMs. The data types of DFMs are hierarchically defined as done in PCM by composite and collection data types.

In addition, we require a definition of the data dependencies between input and output data of a process, source, or sink as suggested by Brunza and Weide [6]. We define a triple (E, D_{in}, D_{out}) with E as entity, i.e. process, source, or sink, D_{in} as a set of input data, and D_{out} as a set of output data. The semantics of the triple is that as soon as all input data D_{in} is available, the entity E can yield output data D_{out}. There can be multiple triples involving an entity E. Especially, this means that not always all data inputs have to be available in order to send or produce data.

3 Running Example

We use the example shown on the left of Figure 1 to illustrate the challenges in mapping DFMs to PCMs. The figure shows a simple ordering system consisting of an order and a payment process: A customer orders and pays goods and a shipment department sends a package containing the goods and a bill. The right hand side of the figure shows a part of the PCM model resulting from the mapping procedure described in the next section. Data dependencies of processes, sources, and sinks are given in the equation list below and annotated in the figure. *ccd* means *creditCardData*.

$$(\text{Customer}, \emptyset, \{\text{order}\}) \quad (1)$$

$$(\text{Customer}, \{\text{orderConfirmation}\}, \{\text{ccd}\}) \quad (2)$$

$$(\text{order}, \{\text{order}\}, \{\text{bill}, \text{orderConfirmation}\}) \quad (3)$$

$$(\text{order}, \{\text{paidBill}\}, \{\text{paidBill}, \text{inventoryItems}\}) \quad (4)$$

$$(\text{pay}, \{\text{ccd}, \text{bill}\}, \{\text{paidBill}, \text{payConfirmation}\}) \quad (5)$$

$$(\text{Shipment}, \emptyset, \emptyset) \quad (6)$$

4 Mapping Guidelines

The mapping of DFMs to PCM models consists of a fixed and a variable part. The fixed part contains mappings that always apply to models. The variable part requires the software architect to make design decisions. We base our mapping guidelines on the transformation from data flow diagrams to OOD introduced by Alabiso [3]. We replace OOD concepts with PCM concepts and elaborate on design decisions.

In this paper, we focus on the variable parts. Therefore, we only explain the fixed part briefly: DFM data types map to PCM data types. DFM sources and sinks are external entities and therefore map to users, i.e. usage scenarios. DFM files map to components that provide data access services. DFM processes map to PCM services, i.e. operational signatures and corresponding Service Effect Specifications (SEFFs).

The variable part consists of two mappings: The architect has to 1) group services to interfaces, as well as decide which component provides that interface, and 2) map data flows to service calls and parameters.

In the following subsections, we describe how to derive stubs for the PCM usage, system, and repository models, as well as how to handle ambiguous situations for our running example in Figure 1. By stubs, we mean reasonable structures that have to be parameterized and further specified to predict quality. For instance, we cannot derive resource demands because this information is not part of a DFM.

4.1 Operational Interfaces

DFMs characterize interfaces by incoming and outgoing data. PCM characterizes interfaces by callable operational interfaces with input and return parameters. Intuitively, each executable action of a process, source, or sink maps to an operation signature. Consumed and provided data maps to the parameters.

In a first step, we have to identify executable parts and corresponding triggers, i.e. callers. As specified in Section 2, execution does not always require all input data to be present. Therefore, data dependencies describe executable parts most closely. In our running example, the customer and the order process have two data dependency triples each. The shipment department and the pay process have one data dependency each. All data dependencies of sources and sinks map to system services to be called. For instance, there have to be two system services to be called by the user. Data dependencies of processes map to services specified in internal interfaces. For instance, there have to be two services offered by the order process.

Mapping data to parameters is only straight forward if all incoming and all outgoing data come from and go to the same entity. In any other case, additional control flows have to fulfil the data flow specification. In our running example, mapping the user data dependencies (1), (2), and (6) is straight forward as can be seen for the interfaces *IOrderCustomer*, *IPayCustomer*, and *IOrderShipment* in Figure 1.

In contrast, the data dependency of the order process taking an order (3) returns two data items, of which one goes to the user and one to the pay process. In that case, we group returned data by recipients. All data going to the triggering entity is a return parameter. For all remaining groups, we can either push the information to the recipients or let the recipients pull it. In this paper, we only consider pulling. Therefore, we create an operation in *IPullBill* that returns a bill.

The data dependency (5) of the pay process requires credit card data from the user and a bill from the order process. To solve this, we identify the triggering entity, keep data of the trigger a parameter, and acquire remaining data via pulling. Only architects can clearly identify triggers. However, a good heuristic is to identify data that directly or transitively comes from users because this is the origin of the control flow in PCM. Therefore, we create the *IPayCustomer* interface that receives credit card data from the user as shown in Figure 1. To acquire a bill,

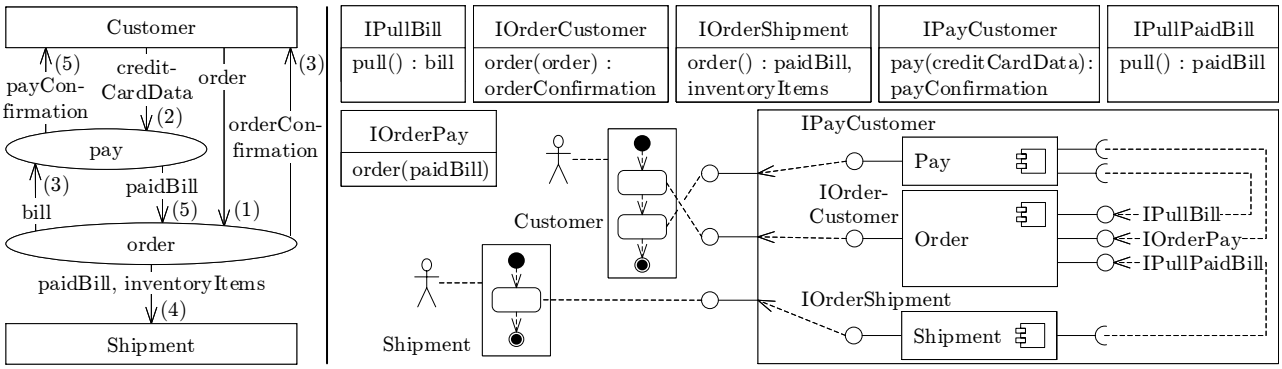

Figure 1: Simple DFM of ordering process (left) and PCM model resulting from mapping (right).

we use the previously created *IPullBill* interface. To determine output parameters, we apply the approach described in the previous paragraph, which introduces the *IPullPaidBill* interface.

The same approaches apply for the second data dependency (4) of order. This yield the *IOrderPay* and the *IOrderShipment* interfaces.

4.2 Usage Model

Each source and sink maps to a usage scenario. Therefore, we create a usage scenario for customer and shipment. In usage scenarios, users call system services identified in the previous step. The sequence of entry level system calls depends on the data dependencies. In our running example, the customer calls the service not requiring any data (1) before the service requiring the order confirmation (2). The shipment department only has one service to call.

4.3 Components and Interfaces

Each process maps to a component. The interface mapping process of Subsection 4.1 already provides us with all required information for required and provided roles: The component provides the interfaces that we derived from its output data. The component requires the interfaces that replaced input parameters in its data dependencies.

4.4 System Model and SEFFs

We create an assembly for each component in the system model. The wiring is unambiguous because our mapping yields exactly one component providing and exactly one component requiring an interface. The same holds true for the system provided roles in PCM.

SEFFs only consist of external service calls. First, the component pulls all required data not contained in input parameters. We already determined pull sources when creating operational interfaces. Second, the component calls all services that require data from the component via an input parameter because this means that the component has to trigger the service. As already said, we cannot provide resource demands because this information is missing from DFMs.

5 Conclusion

DFMs are valuable for quality analyses but are hard to transform into CFMs like PCM. To ease transformation, we introduced mapping guidelines tailored for PCM. The guidelines require an extended version of classic DFMs defined by DeMarco that allows specifying relationships between input and output data. We demonstrated ambiguous situations and initial heuristics for solving them in our running example. The resulting set of mapping guidelines is not complete yet. As soon as the set is exhaustive, automated mappings can support using both DFMs and CFMs in parallel without contradicting information in the models.

In future work, we plan completing the mapping guidelines by constructing examples, evaluating real world models, and deriving new guidelines. We plan to automate the transformation from a DFM to PCM and evaluate our guidelines against case studies.

Acknowledgements. This work was partially funded by the the German Federal Ministry of Education and Research under grant 01IS17106A (Trust 4.0).

References

[1] T. DeMarco. *Structured analysis and system specification.* Prentice-Hall, 1979.

[2] P. T. Ward and S. J. Mellor. *Structured development for real-time systems.* Vol. 3: Implementation modeling techniques. Yourdon Press, 1986.

[3] B. Alabiso. "Transformation of Data Flow Analysis Models to Object Oriented Design". In: *OOPSLA'88.* ACM, 1988, pp. 335–354.

[4] M. Page-Jones. *The Practical Guide to Structured Systems Design.* 2nd ed. Prentice-Hall, Inc., 1988.

[5] T. H. Tse and L. Pong. "Towards a Formal Foundation for DeMarco Data Flow Diagrams". In: *The Computer Journal* 32.1 (1989), pp. 1–12.

[6] P. Brunza and T. van der Weide. *The Semantics of Data Flow Diagrams.* Technical Report TR 89-16. Dept. of Information Systems, University of Nijmegen, 1993.

[7] P. G. Larsen, N. Plat, and H. Toetenel. "A Formal Semantics of Data Flow Diagrams". In: *FAOC* 6.6 (1994), pp. 586–606.

[8] R. H. Reussner et al. *Modeling and Simulating Software Architectures – The Palladio Approach.* MIT Press, 2016.

[9] S. Seifermann, R. Heinrich, and R. H. Reussner. "Data-Driven Software Architecture for Analyzing Confidentiality". In: *ICSA'19.* IEEE, 2019, pp. 1–10.