Compositional Performance Abstractions of Software Connectors

Misha Strittmatter
Karlsruhe Institute of Technology
Am Fasanengarten 5
76131 Karlsruhe, Germany
misha.strittmatter@student.kit.edu

Lucia (Kapova) Happe
Karlsruhe Institute of Technology
Am Fasanengarten 5
76131 Karlsruhe, Germany
kapova@kit.edu

ABSTRACT

Typically, to provide accurate predictions, a performance model has to include low-level details such as used communication infrastructure, or connectors and influence of the underlying middleware platform. In order to profit from the research on inter-component communication and connector design, performance prediction approaches need to include models of different kinds of connectors. It is not always feasible to model complex connectors with all their details. The choice of suitable abstraction filter, which reduces the amount of detailed information needed with respect to the model purpose, is crucial to decrease modelling effort. We propose an approach by which an abstract connector model can be augmented with selected adaptations and enhancements using model completions to result in a more detailed connector model. As the purpose of our models is performance prediction, we designed a suitable abstraction filter based on the *Pipes & Filters* pattern to produce performance models of connectors. Thus, we need to characterize only a small set of compositional and reusable transformations. The selection of applied transformations is then based on the feature-oriented design of the connector's completion.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems—*Modeling techniques*; D.2.8 [**Software Engineering**]: Metrics; D.2.11 [**Software Engineering**]: Software Architecture—*Patterns*

Keywords

Connectors, Model Completions, Performance Abstractions

1. INTRODUCTION

In Model-Driven Software Performance Engineering (MDSPE), applications are composed from prefabricated components. The modelling of the system is done at a high abstraction level. Adding non-functional information to the component specifications enables a system-wide analysis. This is the main intention of the Palladio Component Model (PCM) [2]. A design-time performance prediction using component-based models requires plenty of implementation details to be sufficiently accurate. Model Completions [5] are one possible approach to reduce model development effort and keep the model free from implementation details (especially in early development stages), by adding components with performance-relevant details to the model transparently for the user and automatically by model transformations.

The separation of concerns is essential to avoid construction of large and monolithic models, which are hard to maintain or reuse. Reusability of such models is limited especially because they are often designed for one purpose, and as such do not consider possible enhancements when the purpose of the model changes and new domain-specific details have to be introduced. For example, a component-based architecture model could be used to predict performance. However, the same model could be used to analyse reliability, as well. Both of these purposes require additional domain-specific details, i.e. performance or reliability specific implementation details.

Model Completions support the reuse of purpose-specific abstractions of specific implementation details, which could be configured and used for different purposes. Moreover because the integration of the completions is automated, they are also a technique of experimenting with different settings without having to adapt the whole model manually. The goal of this paper is to create a basis for the implementation of completions which insert connector models into PCM model instances. Although, we have used PCM as modelling language, the analysed connectors and their feature models are applicable for all component-based systems in general. This work introduces: i) compositional performance abstractions that define model transformations to enhance models of connectors; ii) feature models describing possible configuration variants of the connectors; and iii) descriptions on how the configurations of these features influence the system architecture and performance. Subjects of analysis are the connectors, such as procedure calls, messaging, streaming and blackboards analysed by [3]. We focus in this work on performance of the software systems and thus we provide only performance abstractions of the features in the feature models.

The paper is structured as follows. After discussing the foundations in Section 2, we provide a discussion on the suitable connector abstractions in Section 3. The result of this analysis is an architecture and feature-oriented design of an

ICPE'12, April 22-25, 2012, Boston, Massachusetts, USA

exemplary connector. Section 4 discusses the implementation of the completions. Section 5 gives an overview of work related to our approach. Finally, Section 6 concludes the paper, highlights future research directions, and additionally, we discuss the limitations and the validity of the approach.

2. FOUNDATIONS

Software performance engineering (SPE) enables an early performance evaluation of software systems. It allows software architects and developers to identify potential performance problems, such as bottlenecks, in their software systems during the design phase. For this purpose, SPE integrates performance predictions directly in the software development process.

MD-SPE [1] aims to improve the prediction quality of performance models while requiring little manual effort. Software architects describe their system in a language specific to their domain and annotate these models with configurations of performance abstractions (completions) or other performance-relevant information. *Completions* [5] are configurable transformations which add performance-relevant details about the implementation, execution environment etc. to the models. Executing all completions results in the refined model, which then can be transformed into a performance model. Typical models for performance analysis are queueing networks, stochastic Petri nets, or stochastic process algebras. Solving the performance models by analytical or simulation-based methods yields various performance metrics for the system under study. In practice, tools encapsulate the transformation and the solution of the models and hide their complexity.

A *Feature Diagram* is a tree with its nodes representing features (except the root). The connection between nodes carry annotations, which state rules on how features can be selected. E.g. there are optional or mandatory features, alternative (or) and mutually exclusive (xor) feature sets. A tree can be instantiated by selecting features (following the rules) and annotating additional data, where demanded. We use such instances as configuration for completions [5].

We apply our approach in the area of *Component-base Software Architecture*. The implementation of our approach is based on an architectural modelling language PCM [2]. It is specifically designed for performance prediction of component-based systems, with an automatic transformation into a discrete-event simulation of generalised queuing networks. Software *components* are the core entities of the PCM. Basic components contain an abstract behavioural specification called Resource Demanding-Service Effect Specification (RD-SEFF) for each provided service. RD-SEFFs describe how component services use resources and call required services using an annotated control flow graph. *AssemblyConnectors* connect required interfaces with provided interfaces. *Resource containers* model the hardware environment in the PCM. They represent nodes, e.g., servers or client computers, on which components can be allocated. They provide a set of processing resources, such as CPUs and hard disks, which can be utilized by the hosted components.

3. CONNECTOR ABSTRACTIONS

The purpose of our models is performance prediction, therefore we aim to model only performance abstractions of connectors, thus we can abstract from the functional details and concentrate on the performance-relevant dependencies. From a performance point of view, a connector is a chain of components producing load dependent on the size of data being sent through it (and sometimes on additional properties, e.g. entropy). In such a highly abstract connector model, we can simplify the basic building blocks of connectors to two types of activities: buffering of transferred data and computation or I/O activities involving the data. This is very similar to the *Pipes & Filters* pattern, which is an architecture pattern for data stream processing systems [4]. As we break down the connector's overall task into several independent incremental processing steps, we can use the Pipes & Filters pattern and consider filters as processing steps, which execute their subtask, while pipes connect adjacent filters and provide buffering, linking and transport. A filter is defined by the load it generates (dependent on the properties of data) and if and how it changes the properties of data (e.g. compression reduces bytesize, but increases entropy). The filters of a connector also form a chain of producer-consumer systems. A filter is a consumer with regard to his predecessor and a producer with regard to his successor, while the pipes are the bounded buffers in between. This internal mechanism is modelled in the RD-SEFFs of the filter and pipe components. Filters also have a definable amount of worker threads. One challenge we faced was to model the behaviour of these active components (opposed to reactive components, which just pass a call along) using the PCM. This was achieved by adding worker management interfaces, so that the pipe could acquire and release workers of adjacent pipes.

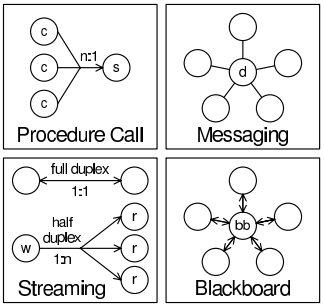

Figure 1: Connector Layout

The settings in which a particular connector can be used are determined by its topology. In the scope of this work we built several exemplary connectors (including their feature trees) with varying topology and arbitrary cardinality, whose underlying architectures were designed by [3]. These connectors are *Procedure Call*, *Messaging*, *Streaming* and *Blackboard*. Their layout is shown in Fig. 1. The Procedure Call and half duplex Streaming connectors feature unidirectional, while Full Duplex and Blackboard connectors feature bidirectional communication. Components connected to the Messaging connector can either be sender, receiver or both. Considering one direction of communication, a call can either be synchronous (i.e. caller will wait for response) or asynchronous (i.e. caller can resume as soon as the call was accepted by the first pipe).

Fig. 2 shows the sub feature tree which is used to configure the behavior of one generic filter. The declarations of the sizes of worker pool and buffer of the preceding pipe are

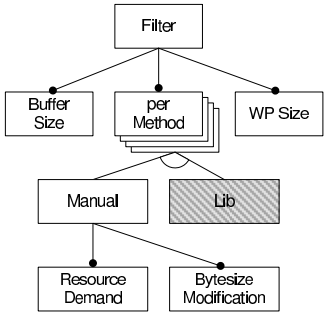

Figure 2: Filter (Sub) Feature Diagram

mandatory. For each method, which is contained in the signature of the interface on which the connector is being applied, one can either define the further behaviour of the filter manually or use predefined formulas.

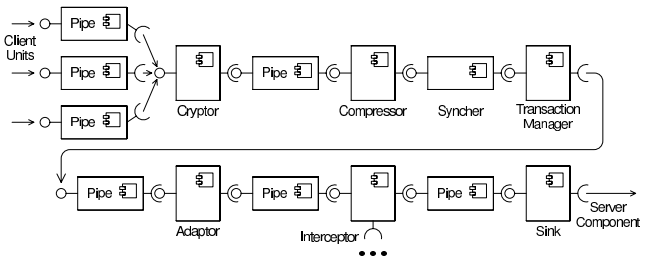

Figure 3: Procedure Call Connector Server Unit

A connector is composed of multiple units (a unit is nothing more than an assembly of components). The most simple 1:1 procedure call connector e.g. is composed of a client and a server unit. Should the two components (which are connected by the connector) be deployed on different resource containers, the client unit will be deployed on the client's container and vice versa. As an example of how the connector looks like when realized within the PCM, Fig. 3 shows the server unit of a procedure call connector (though in this case it has 3:1 cardinality). The unit is fully featured (all features which map to server side filters are selected). Most processing steps are realized as simple filters, while the interceptor can be linked to an external interface and the transaction manager may be inserted by another (nested) completion.

4. COMPLETION IMPLEMENTATION

To adapt models corresponding to the real system, we implemented a transformation realizing connector completions and composing them from basic elements. It is a model-to-model transformation using QVT Relational [7]. The tool used to run the transformation is the Medini QVT Engine. The transformation can be divided into a number of basic steps. As the first step of the transformation, the source model has to be copied completely into the target model, but without the annotated *AssemblyConnectors* (because it will be replaced by our connector abstraction). Then the *Connectors* have to be created. The creation of a *Connector* can be divided into the following steps: (a) find elements in the target model, (b) create new elements, (c) connect elements, (d) allocate elements, and (e) place elements. First, the location (*pivot element*) and the elements that are directly connected to this location, where the *Connector* has to be inserted into the copied system, has to be identified. The new elements and their interconnections are defined by the feature diagram. For each completion, a completion transformation is generated. Second, this generated transformation is executed and all the new elements that are part of the connector are integrated. Then all the elements get allocated to hardware resources depending on the allocation in the source model. At the end all created elements have to be placed inside the system or the allocation element to be at the right place in the PCM model.

The resulting transformation then integrates the selected model primitives based on the configuration of the feature model (cf. Figure 4). Moreover, the transformation calibrates the feature model for a certain platform by adding the resource demands to the connector primitives modelled as filters (cf. Figure 4: Resource Demands (RD)).

In the performance abstractions of connectors, a feature is a discrete design element of connector and a model (i.e., transformation) primitive is a relation from an abstract connector to a more detailed connector. These transformation primitives operate on the connectors and their parameters. Each transformation primitive introduces a part of connector and their composition builds a whole valid connector. In order to achieve generality and flexibility, we have to achieve appropriate granularity of transformation primitives. However, in most of the approaches the additional effort that is required to compose a large number of these small parts is a problem. The means to implementing the primitives and derive the transformation were chosen with this concern in mind. Thus, we implement the primitives as relational transformation fragments (i.e., in QVT-R) with a very high compositionality. The Figure 4 illustrates a few simple compositional and reusable transformation primitives used to implement the procedure call connector.

5. RELATED WORK

Mehta et al. [6] present an approach to a taxonomy of connectors. They feature several levels of categories each of which provides a connector instance on the lowest levels. New connector instances can be composed by combining multiple features from the lowest levels. A technical report of Bures and Plasil [3] explains how connectors can be composed of elements and illustrates the connector generation process. In preparation to our work, we researched how other component systems treat connectors. While they are mainly not used for design time performance predictions, there are still important aspects for our work in these systems. As surveyed in [1], *ROBOCOP*, a component model with a performance prediction framework, has its application in the field of embedded systems. In addition, to passive components it also supports active and mixed-mode components. As an outcome of the active/passive consideration it also explicitly distinguishes between synchronous and asynchronous calls. *OLAN* is an environment for architecture description of distributed systems using middleware. It also features explicit synchronous and asynchronous calls. The *SOFA* component model, treats connectors as first class entities and they can be described by their architecture.

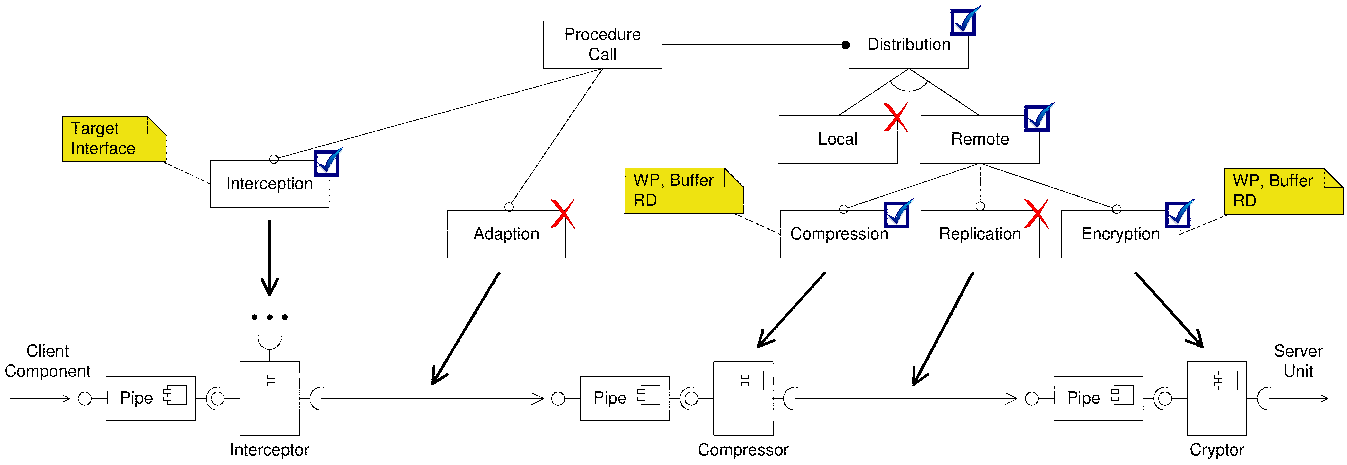

Figure 4: Resulting active fragments of transformation in the completion.

6. CONCLUSIONS AND VISIONS

In this work-in-progress paper a suitable performance abstraction for Procedure Call Connector, its design and performance impact, was analysed. As part of the definition, connector feature models summarizing the configuration options were identified and systematically analysed. An important aspect of this work was to examine the potential of the PCM regarding its ability to model connectors. As a conclusion, almost everything was possible to implement with the PCM's own means. The behaviour of active components can be emulated with passive resources representing thread-pools. Buffering is done through the use of blocking on passive resources, asynchronous calls through forking. Connectors do not have to be considered as first class entities since they can be utilized using completions. Only the lack of an internal component state was a constraint (e.g. when considering GC effects). Without it, the blackboard connector cannot be used to communicate exact values. Instead it responds with statistical data. The resulting connector models were composed from reappearing and reusable parts, which consequently eased the development of the required model transformations. In this work, we showed that starting with general models and deciding about the purpose of models later in the model development is acceptable, when the models' purposes can be handled in isolation. In other words, an early usage of modelling abstractions designed for one specific purpose only mirrors in the later usage of models and the effort necessary to automate modelling actions.

As such this paper is a starting point for a wide variety of further work. A more comprehensive case study should be conducted to evaluate the effect of the whole concept and the whole set of possible features. The results can also be used to calibrate the architecture, its configuration as well as predefined values, formulas and libraries. Most valuable would be a complete integration of automated measurements and regressions benchmarks (e.g. Performance Cockpit [8]) to calibrate completions. Such approaches can provide multi-dimensional parametrised resource demands needed for completions with less-effort. Especially already developed completions (e.g. middleware completions) have to be coordinated or combined with the connector-completion.Finally, the matured completion can be included in a completion library or be directly integrated into the PCM IDE.

7. REFERENCES

[1] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-Based Performance Prediction in Software Development: A Survey. *Transactions on Software Engineering*, 30(5):295–310, May 2004.

[2] S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. 82:3–22, 2009.

[3] T. Bures and F. Plasil. Composing connectors of elements. Technical Report 3, Dep. of SW Engineering, Charles University, Prague, May 2003.

[4] F. Buschmann. *Pattern-oriented software architecture*, volume 1: A system of patterns. Wiley, Chichester [u.a.], repr. edition, 2007.

[5] L. Kapova and T. Goldschmidt. Automated feature model-based generation of refinement transformations. In *Proceedings of the 35th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2009.

[6] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 178–187, New York, NY, USA, 2000. ACM.

[7] Object Management Group. *MOF 2.0 Query/View/Transformation, version 1.0*, Apr. 2008.

[8] D. Westermann, J. Happe, M. Hauck, and C. Heupel. The performance cockpit approach: A framework for systematic performance evaluations. In *Proceedings of the 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2010)*. IEEE Computer Society, 2010.