Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 264 (2010) 73–90
www.elsevier.com/locate/entcs

Systematic Refinement of Performance Models for Concurrent Component-based Systems

Lucia Kapová [1]

Chair for Software Design and Quality
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany

Steffen Becker[2]

FZI Research Center for Information Technology,
Karlsruhe, Germany

Abstract

Model-driven performance prediction methods require detailed design models to evaluate the performance of software systems during early development stages. However, the complexity of detailed prediction models and the semantic gap between modelled performance concerns and functional concerns prevents many developers to address performance. As a solution to this problem, systematic model refinements, called completions, hide low-level details from developers. Completions automatically integrate performance-relevant details into component-based architectures using model-to-model transformations. In such scenarios, conflicts between different completions are likely. Therefore, the application order of completions must be determined unambiguously in order to reduce such conflicts. Many existing approaches employ the concept of performance completions to include performance-relevant details to the prediction model. So far researcher only address the application of a single completion on an architectural model. The reduction of conflicting completions have not yet been considered. In this paper, we present a systematic approach to reduce and avoid conflicts between completions that are applied to the same model. The method presented in this paper is essential for the automated integration of completions in software performance engineering. Furthermore, we apply our approach to reduce conflicts of a set of completions based on design patterns for concurrent software systems.

Keywords: Component-based Software Engineering, Software Performance Engineering, Performance Prediction.

1 Introduction

In software performance engineering, abstract design models are used to predict and evaluate response time, throughput, and resource utilisation of the target system

[1] Email: kapova@ipd.uka.de
[2] Email: sbecker@fzi.de

doi:10.1016/j.entcs.2010.07.006

during early development stages. To provide accurate predictions, performance models have to include many low-level details. For example, the configuration of a message-oriented middleware (e.g., a size of a transaction) can affect the delivery time of messages [11]. While most of the implementation details are not known in advance, a rough knowledge about the design patterns that are to be used might be already available. This knowledge can be exploited for further analysis, such as performance and reliability prediction, and for code generation.

However, the architectural models that accurately reflect the performance of the system under study can become very complex and hard to understand. This problem of high complexity, lack of standardisation, and lack of automation for performance modelling has been clearly stated in [23]: "*There is a semantic gap between performance concerns and functional concerns, which prevents many developers from addressing performance at all. For the same reason many developers do not trust or understand performance models, even if such models are available. Performance modelling is effective but it is often costly; models are approximate, they leave out detail that may be important, and are difficult to validate.*"

In literature, the above issues are addressed by model refinements that integrate performance-relevant details into software architectural models. In the remainder of this paper, we call model refinements that specifically address quality attributes of software systems *completions* [24]. In the original approach of Woodside et al. [24], performance completions have to be added manually to the prediction model. The difficulty of automation is a result of the flexibility needed for performance completions [23]. In order to provide tool support and to apply performance completions, we have to address this problem. Model-driven development can provide the needed automation by means of model transformations. For example, the authors of [8] analyse design patterns for Message-oriented Middleware. They use the selected combination of messaging patterns as configuration (also called mark model) for model-to-model transformations. Basically, existing solutions [24,8,10] focus on the integration of only one completion at a time. In scenarios where more than one completion is applied to model element, conflicts between different completions are likely.

In our approach, completions are realized by model-to-model transformations that can be configured by a mark model [10]. The configuration provides the necessary variability. The transformations are applied to model elements specified by the software architect. To automate the integration of completions, we need a method to specify the order in which completions are applied. For this purpose, we propose a method that reduces the number of potential conflicts and allows the explicit specification of the execution order. To evaluate our method, we developed a number of completions based on design patterns for concurrent software systems, e.g., Thread Pool, Replication, Publisher-Subscriber Connector, Message-Oriented Middleware [8] and Barrier. A detailed description of these completions is out of the scope for this paper. Additionally, our previous work [7,10] describes the details related to the definition of model refinement and completions.

In this paper, we address the issue of dependencies between completions. These

dependencies define where and when certain completions can be woven into the model. The execution order of the completions may affect the result model in a way that the following completions are not applicable anymore or that the analysis results are altered. Therefore, the order of completions must be unambiguously specified. Furthermore, we have to clarify responsibilities if additional information to reduce conflicts is necessary. For this purpose, we specify the roles in the development process responsible for specific completions.

The contributions of this paper are (I) a systematic approach to investigate dependencies and conflicts between multiple completions applied to the same source model, (II) guidelines to avoid and minimise potential conflicts, with a goal to reduce conflicts in transformation generators, and (III) an application of our approach to the domain of design patterns for concurrent software systems.

This paper is structured as follows. In the section 2, we describe the basic concepts of model-driven development and completions. Section 3 provides a motivating example to illustrate the problem of conflicting completions. Based on the introduced concepts, section 4 introduces our approach to specify sequences of completions execution and minimises possible conflicts. In section 5, the analysis of completions for concurrency patterns demonstrates the applicability of our approach. Section 6 summarizes related work. Finally, section 7 concludes this paper and highlights future research directions.

2 Foundations

In the following, we introduce the technologies and architectural languages for specifying software architectures and their extra-functional properties. We apply our approach in the domain of performance engineering. For this purpose, we use a performance prediction approach called Palladio Component Model (PCM) [17,12,2]. The PCM is a modelling language specifically designed for performance prediction of component-based systems. Figure 1 illustrates a system model with performance annotations in PCM. It consists of four models created by four developer roles in a parametric way, which allows the models to be updated independently on each other. Component developers specify the behaviour and performance properties of components, software architects combine components into component assembly with defined system interfaces, system deployers define execution environment and allocation of software components to system resources, and domain experts specify the scenarios of system usage that drives system execution. A model-to-text transformation maps the architectural model into a discrete-event simulation which resembles a generalised queueing network. The simulation predicts various performance metrics such as resource utilisation or response time distributions of the system and of individual components. The figure shows an assembly of components forming a system. In the following sections, we introduce the foundations related to model-driven architectures and performance completions.

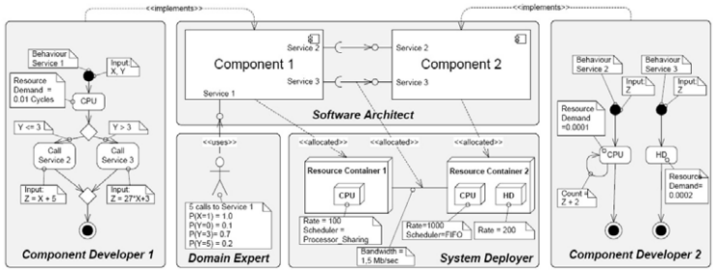

Fig. 1. Illustration of a PCM model.

2.1 Model-Driven Engineering

In model-driven software development processes like the OMG's Model-Driven Architecture (MDA) [16] process, models serve as input for transformations to generate the system's implementation. In Figure 2, the refinement process is distributed

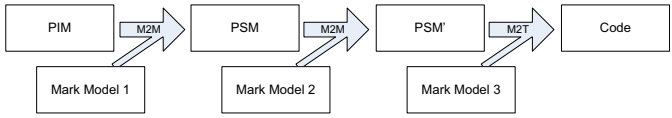

Fig. 2. MDA models and transformations.

among a number of transformations forming a transformation chain. Each transformation takes the output of the previous transformation and adds its own specific implementation details to the model. When refining high-level concepts of transformations into concepts on lower abstraction levels, sometimes different options are available for mapping such high-level model elements. For example, if different applications communicate via messaging, different patterns for realizing the message channels can be used, e.g., with or without guaranteed delivery. If developers want their transformations to be flexible to express these options, they can parametrize them by so called mark model instances. Mark model allows users of a transformation to decide on mapping variations themselves. Czarnecki and Eisenecker [5] used so called feature diagrams to capture different variants in the possible output of model or code generators. In Figure 3, a feature model describes the possible configurations of the Message-oriented Middleware (MoM). The MoM Feature Model captures different configurations for a Messaging system. The configuration includes the type of Messaging Channel as well as characteristics of the Sender and Receiver. For example, a Messaging Channel can be configured as a Point-to-Point Channel if only a single Receiver is needed. The Message Size is a property of the Sender and expresses the amount of data transfered. Furthermore, the number of Competing Consumers at the Receiver's side can be specified. The choice of either of these features results in a change of the architectural model. The effect of these changes varies from setting a parameter, through structural changes, to changing the deployment of a system.

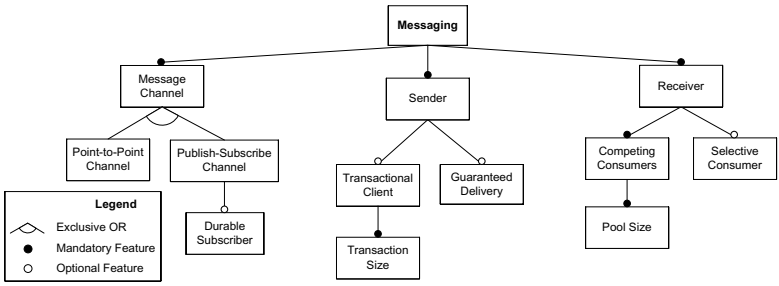

Fig. 3. MOM Adaptor Feature Model.

2.2 Performance Completions

When doing performance predictions in early development stages, the software model has to be kept on a high level of abstraction. The complexity and the specific knowledge about the implementation required to create the necessary models would dramatically increase the modeling effort. The complexity of such models reduces the variability of the design models and, thus, increase the effort to evaluate and compare design alternatives. However, detailed information about the system is necessary to determine the performance of the modeled architecture correctly. Performance completions, as envisioned by Woodside [22,24], are one possibility

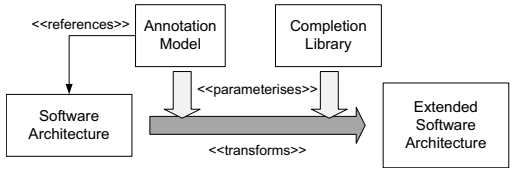

Fig. 4. Transformation integrating performance completions.

to close this gap. They are components added to the prediction model that add performance-relevant details to a performance prediction model but which are not of interest when designing the system's application logic. For example, details about the design patterns or platform are not included within the design model and therefore should be added by completions. These performance completions extend the software model with annotations (or rules) whose refinements (such as additional components, execution environments, or communication design patterns) are added to the original software architecture.

Figure 4 shows how performance completions can be realized using the MDA concepts described in the section 2.1. Elements of a software architecture model, such as components or connectors, are annotated by elements of a *Mark Model* using, for example, feature diagrams. Mark models annotate elements in the architecture which are to be refined and provide the necessary configuration options. For example, if a connector is to be replaced by message-passing the mark model can provide information about the type of the messaging channel, e.g., using guaranteed delivery. Model-to-model transformations take the necessary components from the *Completion Library*, adjust them to the configuration, and insert them in the software architecture prediction model. The result of the transformation is an ar-

chitecture model whose annotated elements have been expanded to its detailed performance specifications. Figure 5 illustrates the changes of a model resulting from specific configuration options. This step of model refinement has been automated by [7,10] where refinement transformation generators based on actual completion configuration are introduced. The resulting model is not performance-equivalent to the input model. To provide more accurate performance prediction the resulting model includes more low-level details. The accuracy of these predictions was discussed in [8]. The approach presented in this paper uses performance-specific

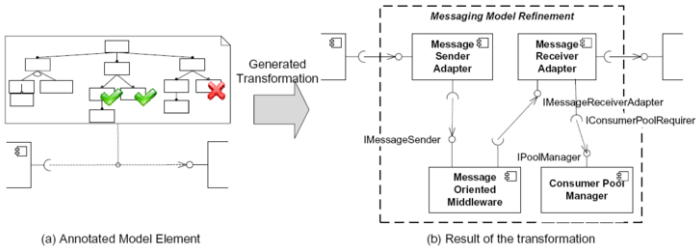

Fig. 5. Transformation integrating MOM Adaptor.

completions for concurrency patterns to enable software architects to easily predict the performance properties of different architectural alternatives. The patterns can be configured with different parameters to analyse the influence of concurrent component interactions on performance. In this section presented completion theory allows to integrate one completion to the model, we will discuss further the execution of the completion sequences.

3 Motivating Example

To motivate our method, we present an example system of a supply chain management (SCM) for supermarkets. In particular, we are interested in the performance of a business reporting system for a subset of supermarkets. Figure 6 shows the

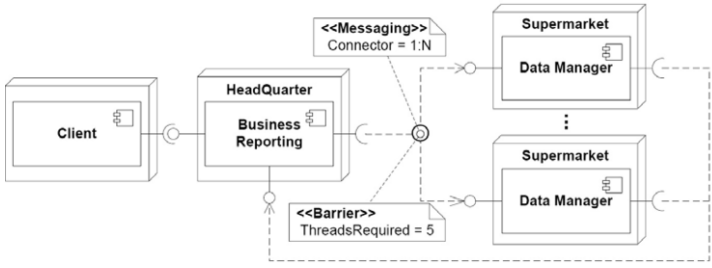

Fig. 6. Annotated Architecture.

part of the system's architecture relevant for business reporting. The main part of the business reporting is running on HQ's server system. However, the data is distributed among the company's supermarkets and managed by `Data Managers`. In order to generate a report for a particular set of supermarkets, the `Business Reporting` sends a request to the supermarkets of interest. The data managers of

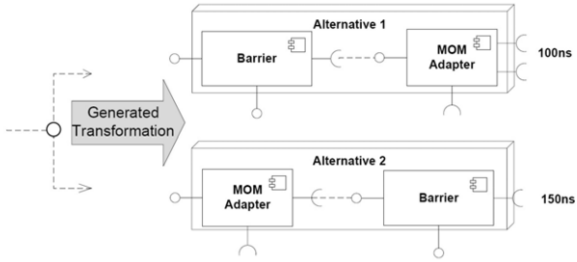

Fig. 7. Completion Alternatives.

each supermarket retrieve the necessary data and send it back to HQ. As soon as all data is available, the `Business Reporting` generates the report and returns it to the client.

In this example, one connector (line connecting required interface of a client to provided interface of a server) is annotated by two completions: firstly, by a `Barrier` pattern configuration and, secondly, as *Message-oriented* connector (`MOM Adapter`, cf. Figure 6). Both of these completion annotations refine the performance prediction model with certain properties. The sequence of completion execution affects the model structure and its validity. In the illustrated example the completion execution order results in different semantic of the `Barrier` component (cf. Figure 7). In a first case the `Barrier` component waits for a number of replies from different `Data Managers`. By changed order the `Barrier` component waits for replies from one `Data Manager`. Additionally, the results of performance prediction could be influenced as illustrated by our example (cf. Figure 7). The whole set of completions for this example could involve different concurrency patterns: message-based communication with publish-subscribe (1:N) connector, barrier, strategized locking, and thread-specific storage. These design patterns were introduced by Schmidt in [19]. To identify a valid completion execution order in such complex system is a non-trivial task.

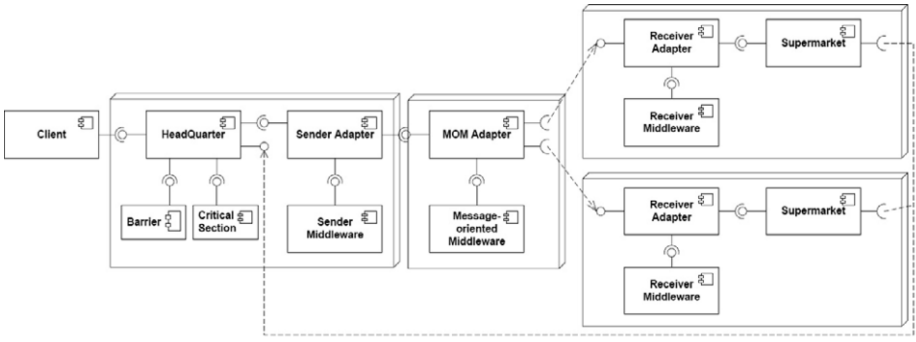

Fig. 8. Resulting Architecture.

In the presented example, the sequence of completion execution should result in the full architecture model illustrated in Figure 8. The method introduced in this paper provides step-based reduction of completion executions order and their conflicts.

4 Completion Conflict Reduction

The architecture is described at design time by the means of an architectural language suitable for specifying architectural elements, like components, connectors, etc. The goal of such architecture models can be, for example a performance analysis or code generation. Models suitable for this purpose are build by a sequence of refinements mirroring the real implementation. Applying the configurable refinements [10] (completions) on an abstract level using model-driven development based on transformations is an adequate instrument. This way, we can create refined architecture models on a lower-level of abstraction. To know when and where (on which model element) to execute a model transformation for completion integration, we have to analyse the completion and the architecture model too. Each time a new completion is introduced, we have to analyse its dependencies to other already known completions. Therefore, we have to focus on a related group of completions where conflicts are more likely.

The introduced approach for minimising and avoiding conflicts between executed performance completions builds on a few systematic steps of architecture model analysis. In the following we describe the problem of minimising and resolving conflicts between executed performance completions on the model level formally.

4.1 Formal Description of Completion Conflict Reduction

In our approach for a component-based systems, we understand a *completion* as model refinement. For example a component A could be refined by a locking strategy, a monitor or a state manager. These completions provide additional details about the components functionality. They also include so called *performance completions* which integrate parametrised resource demands [8] (for example middleware properties) into the model. Such detailed information about a software system is a basis for the analysis of non-functional properties. In the following we will introduce the concept of completions formally.

Let now $C = \{c_i | i \in I\}$ be a finite set of available completions, that we call a *completion library*, where I and J are finite index sets giving each element a unique label. Furthermore, let $V_i = \{v_j | j \in J\}$ be a set if possible variations of completion c_i, which are assumed to be a countable set. To continue with our example we assume that the variation domain of a completion $c_{locking}$ (refining component A with a critical section locking strategy) is $V_{locking} = \{v_{scoped}, v_{double-checked}, ..., v_{strategized}\}$. On this basis we can define a set of tuples $T = \{(v_j, c_i) | i \in I, j \in J, v_j \in V_i\}$, that defines a set of completion instances. The *completion space* CS is defined as a subset of a powerset $\mathcal{P}$, $CS \subset \mathcal{P}(T)$ where for each subset holds: $\forall (v_j, c_a) \neq (v_k, c_b) : a \neq b$.

A *completion chain* $cc_i =< t_1, t_2, ..., t_N >, i \in I, t_i \in T$ is a permutation of t_i for one element from *completion space* $cc_i \in CS$. The *completion chain* cc_k is in *conflict* with $cc_l; k, l \in I$, when an order of completion execution in $cc_i \neq cc_j$ and the validity of the model structure or the result of analysis (e.g., performance prediction) is different of each of the chain definitions.

Because of *conflict* occurrence not all sequences of execution as defined above

are valid for a modeled system. In the motivating example, in section 3, we define the possible completion chains as follows: $C = \{c_{barriere}, c_{messaging}\}$, $V_{barriere} = \{v_{threads\_required\_4}, v_{threads\_required\_5}\}$, and $V_{messaging} = \{v_{connector\_1:1}, v_{connector\_1:N}\}$, where $T = \{(v_{threads_required_4}, c_{barriere}), (v_{threads_required_5}, c_{barriere}), (v_{connector_1:1}, c_{messaging}), (v_{connector_1:N}, c_{messaging})\}$and $CS = \{\emptyset, \{(v_{threads\_required\_5}, c_{barriere})\}, \{(v_{connector\_1:N}, c_{messaging})\}, \{(v_{threads\_required\_5}, c_{barriere}), (v_{connector\_1:N}, c_{messaging})\}\}$, the completion chains are defined as $cc_1 =< (v_{threads\_required\_5}, c_{barriere}), (v_{connector\_1:N}, c_{messaging}) >$ and $cc_2 =< (v_{connector\_1:N}, c_{messaging}), (v_{threads\_required\_5}, c_{barriere}) >$.

4.2 Levels of Completion Conflict Reduction

To define a new completion in the *completion library* we have to investigate the completion model and identify *conflicts* that have to be resolved. We reflect the need for identification, minimisation and reduction of *conflicts* by introducing three levels of *conflict* reduction:

(i) **Roles and Responsibilities Separation**: The first question is *"Who is able to provide all necessary information to use and configure the completion?"*. The selected role in the development process has to have all necessary input data to specify the completion's configuration during software design. Furthermore, he/she can profit from completion usage. Ideally, the assignment of completions to roles will lead to disjointed sets identification. Each role is only responsible for its related completions. This way, possible conflicts are limited to the completions in responsibility of one role. Additionally, separation of concerns based on the roles in the development process creates a hierarchy (identifying domains of concern) in the metamodel of used architecture description language. This is illustrated by a hierarchy of *packages* in the PCM metamodel on the Figure 9.

To focus our reasoning, we have to categorise completions based on the metamodel elements they could be assigned to. This way we reduce possible conflicts on a metamodel level. The proposed categorisation maps the roles in the CBSE development process [13] to groups of completions. The goal of this step is to identify sets of completions where conflicts are possible. This way, we define disjunct sets of completions C_i. For each two completions $c_1 \in C_1$ and $c_2 \in C_2; C_1 \neq C_2$ conflicts are not possible. Only in a case of completions from the same set, conflicts are possible. In a second case, we have to proceed with the next step to further identify affected elements.

(ii) **Conflicting Model Elements Identification**: If conflicts can occur, we have to answer the question *"Which model elements are affected?"*. For this purpose we have to know how the completions are modeled and at which places of the architecture they can be applied. We can identify affected elements as a difference between source and result model. Additionally, by this element identification we provide initial model transformation definition. Identified elements specify more exact locations where conflicts may occur.

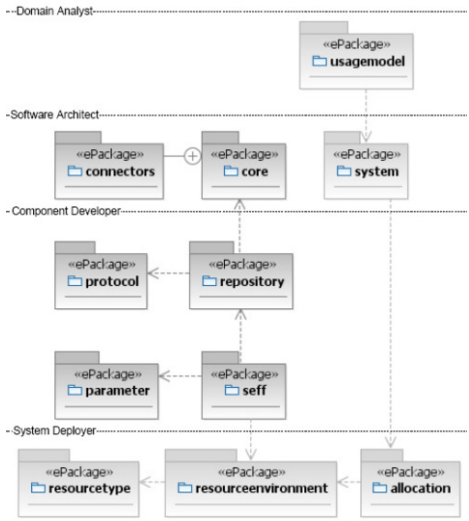

Fig. 9. A hierarchy in the PCM metamodel.

The evaluation of completion chain *cc* for *conflict-potential* is a function $\phi : T \rightarrow S$, where domain S is the set of possible conflicting instances of metamodel elements. For example when evaluating order of integration for locking and state manager (both of them should be applied to the same component) we identify on a model level possible conflict set that includes all elements needed in component and its behavior definition. This results in further separation of conflict domains and decreasing the number of completions that could introduce conflict on a model level. We define sets of potentially conflicting completions (*conflict space*): $ConflictSpace := \{t_i; t_i \in T\}$, where $t_i, t_j \in T, i \neq j | t_i$ potentially conflicts with t_j on a model element $e \in S$, where S is a set of conflicting elements orthogonal to the hierarchy from the previous level, $S = \{$component, connector, resource container$\}$

(iii) **Completion Dependencies Identification**: At the end we need to answer the question *"What are the dependencies to other completions from the same conflict space?"*. Based on previous steps we have to analyse related completions and their intersections (affected model elements). Furthermore, we can generalise dependencies between completions by definition of mutual exclusion or require relationships in a completion specification. Throughout the refinement process we can take advantage of component-based properties, such as components are black-box entities and refinements are applied in a hierarchy to the components (see Section 5.2.1). Based on a previous assumption we can take an advantage of component wrappers hierarchy (the *Interceptor* pattern or the *Layer* pattern in a case of connectors). Additionally, in the source model elements are annotated by completion configurations which results in a three actions: *refine*, *override* or *add* an new element to the model. Based on a completion action priority (specific per each modeled system) we can identify order of completion execution.

The approach introduced in this paper allows to reduce and avoid model completions conflicts on a model-level (*Conflicting Model Elements Identification*) or meta-model level (*Roles and Responsibilities Separation*). This way, we provide guidelines for implementation of model transformations and refinement transformation generators [10]. Introduced approach decreases the complexity of conflicts reduction and minimisation (avoiding non-determinism of conflicts similar as in graph grammars). The effort for manual conflict resolution is minimised on a small set of model elements and the number of cases when is this step needed is reduced.

5 Concurrent Software Systems Completions

Predicting the performance of software systems is especially challenging if software components communicate based on a complex interaction pattern. Such interaction is defined by concurrency, message-based communication, and synchronisation patterns. In the following we investigate these groups of patterns. In our approach, we simplify the design and the development of concurrent software architectures by completions for concurrency design patterns. We provide predefined parametrized performance completions based on a knowledge about concurrency design patterns and their implementation details. In general, design patterns provide enough information to allow accurate performance predictions. Patterns for concurrent and distributed systems address multiple aspects, such as synchronisation, communication, and Quality of Service (QoS). For example, the patterns monitor object [19], thread-safe interface [19], guarded call [6], and rendezvous [6] provide different means for synchronisation and communication. Patterns like Half-Sync/Half-Async, Leader Followers, Reactor, and Proactor as described Schmidt et. al. [19] are used in servers to efficiently dispatch and process concurrent requests. Furthermore, replication and load balancing are employed to enhance different QoS attributes in distributed systems.

Even though it might be known that a certain pattern affects the quality of a system [19,6], the extend of the effect in a certain scenario is unknown. Furthermore, a design pattern may affect several quality attributes. For example, replication increases the availability of a service, but decreases its performance. If multiple patterns are combined to enhance QoS, synchronise components, or ensure data consistency, their overall effect cannot be assessed manually. Therefore, we use model-driven performance and reliability prediction techniques to evaluate the influence of concurrency patterns on the QoS of a software architecture. In the following, we apply our approach for completion conflict reduction to concurrency design patterns.

5.1 Roles and Responsibilities Separation

The categorisation of design patterns based on a development roles and their responsibilities separation builds the basis for reduction and avoidance of conflicts. Additionally, software developers can select suitable patterns for certain problem

	Event-based communication	Synchronisation	Concurrency	Message-oriented communication
Component Developer		Scoped Locking Strategized Locking Thread-safe Interface Double-checked Locking Optimalisation Rendezvous/Barrier	Thread-specific Storage Monitor Object	Messaging Endpoints
Software Architect	Asynchronous Completion Token		Replication	Message Channel Message Routing Message Endpoints
System Deployer	Reactor Proactor Acceptor-Connector		Active Object Half-Sync/Half-Async Leaders Followers Thread Pool	Message Bus

Table 1
Mapping Design Patterns to Development Roles

domain without detailed knowledge about their structure. Developing concurrent software system is most challenging and complex. Design patterns decrease the complexity of concurrent programs and provide solutions to known concurrency problems. We categorised concurrency design patterns according to the development roles, that most likely will use them (see Table 1).

The category *Component Developer* includes patterns used for a definition of basic thread-safe components. These patterns solve the issues related to parallel usage of the component provided service, for example, data inconsistency.

The category *Software Architect* consists of patterns for specification of component interactions, such as coordination and optimisation of communication between components.

The category *System Deployer* subsumes patterns that are used to build middleware platforms for concurrent software systems. For example, the concurrent processing of requests by an application server can be realised by a *Leader/Follower* pattern.

5.2 Conflicting Model Elements Identification

Based on this analysis of completions for concurrency design patterns, we identified three groups of elements that could be affected by completion integration. These elements define possible locations of completion conflict. These conflicts have to be minimised on the model level considering affected model elements. The model element is affected by a completion if i.) is holding completion annotation (*initially affected*) ii.) is *refined*, *overridden* or *added* by completion (*secondary affected*)

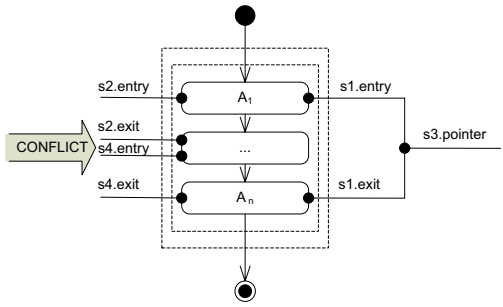

Fig. 10. Component Completions Scopes Definition.

5.2.1 Completions annotating components

The first group of the elements is defined by patterns that affect model elements describing component behaviour. These patterns refine behaviour by integrating new actions (e.g. external call, acquire or release) into the component's control flow. All design patterns for synchronisation and thread-safety belong to this group, e.g., *Locks*, *Monitors*, *StateControllers* or the *Barrier* pattern. We will further discuss *Locks* as a suitable example for this group related concepts.

In order to avoid conflicts for this group of patterns, we need to determine the order of *Lock* acquisitions and releases. In general, *Locks* can be acquired and released at arbitrary points in the program. However, this can result in potential deadlocks. To avoid deadlocks, we need to ensure that resources are acquired and released in a specific (always the same) order. For this purpose, we introduce *scopes* for critical sections. Let *Actions* be the set of all actions used in a system to specify component behaviour and *Scope* the set of all scopes defined on these actions. Then we have that $\forall s_i \in Scope\ \exists A_{i,1} \dots A_{i,n}$ so that the entry point of $s_i; s_i.entry$, points to the first action $A_{i,1}$ in a sequence of actions and the exit point of $s_i; s_i.exit$, points to the last action in the sequence of actions $A_{i,n}$. Furthermore, it must hold that whenever a path (trace) of a component's behaviour includes $A_{i,1}$ it must also include $A_{i,n}$ with a condition that $A_{i,1}$ occurs before $A_{i,n}$ [9]. We define the set of actions that belong to a scope, $actions(s_i)$, as the set of all actions that lie on a path from $A_{i,1}$ to $A_{i,n}$. Having this in mind we can define a conflict as follows: $s_1, s_2 \in Scopes(s_1 \neq s_2)$ are in conflict, if $actions(s_1) \cap actions(s_2) \neq \emptyset$ and $(actions(s_1) \not\subseteq action(s_2)$ or $action(s_2) \not\subseteq actions(s_2)$(cf., Figure 10). This way the sequence of *Locks* completions is always implicit. Additionally scopes of critical sections define the execution location for a next completion (before the $s_i.entry$ or after the $s_i.exit$) on the same model of component behavior. The *Lock* completion could be configured, for example to specify strategy read or write or to define beginning of transaction. By the configuration is the focus even more on specific elements.

5.2.2 Completions annotating resource containers

Dispatching and the management of threads are addressed by a set of patterns dealing with event-based communication and the infrastructure's support for concurrency. Completions for dispatching annotate resource containers to which com-

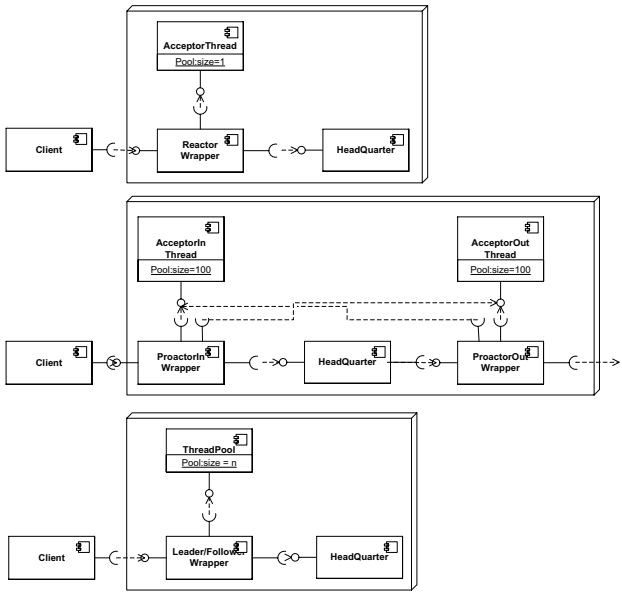

Fig. 11. Performance abstractions for Reactor, Proactor and Leaders Followers pattern.

ponents can be allocated. From the perspective of performance prediction, these patterns can be abstracted as variations of the *ThreadPool* pattern. The implementation of *ThreadPool* has a prominent impact on the performance due to its ability to limit the level of concurrency in the system [4]. We designed performance abstractions based on *ThreadPool* for the following patterns: (cf., Figure 11) for patterns i.) *Reactor*: The *Reactor* pattern realises synchronous communication. This pattern could be abstracted as *ThreadPool* with a size of one thread for a client; ii.) *Proactor*: The *Proactor* pattern realises asynchronous communication. The pattern separates the processing of incoming and outgoing requests. For each type there is a distinct pool of worker threads. Therefore, we can abstract the pattern as incoming and outgoing *ThreadPool* couple with a size equal the capacity of the system; and iii.) *Leaders Followers*: The *Leaders Followers* pattern is a special version of a *ThreadPool* where one particular thread takes the role of the leader and waits for the next request. All other threads are either queued (i.e., followers) or processing requests (i.e. workers). To model this pattern we can easily use one *ThreadPool* component with a size equal the capacity of the system. These patterns belong to the platform definition, therefore we allow only one of these completions per resource container. Consequently, no conflicts are possible.

5.2.3 Completions annotating connectors

Assembly connectors [21,1] are the last (but by no means least) type of model elements that can be refined by completions. For connectors, several performance completions can be applied on one connector instance so that their order has to be determined.

The first kind of completion provides details about the type of the connector, i.e, whether it is 1:1, 1:n, or n:1. Connectors of type 1:1 are typical message passing or

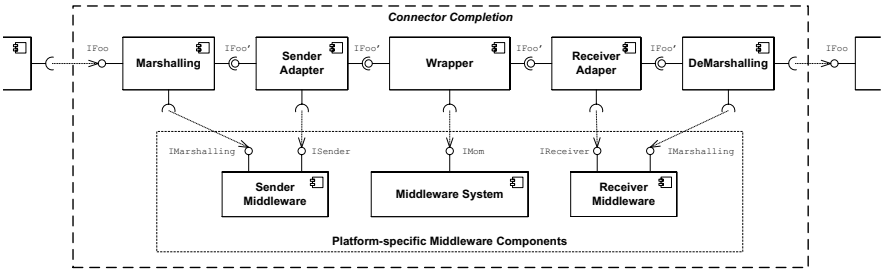

Fig. 12. Connector Middleware Completion.

RPC style connectors which connect a single client component instance to a single server component instance. In case of 1:n connectors, a single client component sends requests to a set of server components which is semantically the case for server replication scenarios or voting based server queries. Finally, n:1 connectors are the usual case of n clients instances talking to a thread-safe server instance.

Orthogonal to the type of the connector, connector performance completions also include details about the processing of the communication (synchronous or asynchronous) in the participating middleware layers as illustrated in Figure 12 [8]. Here we find services for message marshaling, message encryption, call authentication, message compression, etc. For these types of message processing steps, existing performance completions [1] insert a completion component for each processing step. However, the order of these services is important because of the differences in the data flow involved. For example, the size of the message to be sent over the network is different if the message's body is first encrypted and then compressed versus an initial compression followed by a subsequent encryption step. Hence, for the processing steps the order of application of a set of performance completions does matter and needs clarification.

Connector completions rely on components which reflect the performance related behaviour of the used middleware. As a consequence, these middleware components implement both, the resource demand caused by the middleware's processing but also the data transformations they perform on the message to be sent over the network. Note, that in some usecases the size of the message is not of major interest for the overall performance of the network link. In such cases, the data transformations become neglectable and consequently also the order of applying the corresponding performance completions does not matter any more.

As a result of the discussion of connector completions, we can conclude that we need at least two types of annotations. The first annotation class determines the connector kind and defines the exact implementation semantics of 1:1, 1:n, and n:1 connectors, e.g., whether voting or replication is used for a 1:n connector. The second class of annotations defines the pre- and post-processing details of the messages used by the connector for remote communication. Here, the annotation gives details about marshaling, encryption, compression, etc. A clear definition of the order in which such completions are added to the performance model is necessary to get accurate performance predictions from the refined performance model. The

following section gives details on how to identify completions dependencies and reach a deterministic completion application.

5.3 Completion Dependencies Identification

The approach introduced in this paper provides the basis for easy completion integration reducing potential conflicts. For the purpose of avoiding and minimising conflicts we map patterns to the roles and responsibilities involved in the software development process. Based on the meta-model elements affected, we determine those elements that are in potential conflict. After this analysis the complexity of the remaining conflicts on a model instance level is decreased, location of the conflict is identified and conflict resolution can focus on the small set of specific model elements (mostly one element). When is a new completion introduced it should be integrated in one of the categories and this way could be the relation to the other completions easily identified and conflicts minimised.

In this last step a small number of completions belongs still to the same group. As shown on a concurrency design patterns group this set of remaining completions is equal or smaller then two and involves mostly only one model element. Resolving remaining conflicts could be done manually, however we consider guidelines to resolve even this conflicts by prioritisation of completions defining *refine* action, followed by *override* or *add* actions. Additionally, in the case of connector completions the *Layers* [19] pattern give us guidelines to resolve first completions from the upper most layer till the lowest one. This way most of the remaining conflicts could be solved without a manual effort.

6 Related Work

The idea of using patterns as basic concept for predicting extra-functional properties has been discussed in the context of special components called adapter. Adapters are used to bridge any kind of interoperability problems when composing components. Initial work has been done by compiling a classification of adaptation patterns and defining a process to incorporate the patterns in a prediction process for extra-functional properties by Becker et al. [3]. Besides performance, there is also work looking at reliability prediction in the context of adaptation patterns by Reussner et al.[18].

Spitznagel et al. investigated the relationship of architectural connectors and common dependability techniques [20]. A special focus of their work was the composition of more than a single connector to combined connectors. However, their main interest has been guaranteable properties of systems like deadlock-freedom and not in the prediction of the extra-functional impact. In case of concurrency patterns modelling focuses mostly on functional properties or only make limited use of configuration options. Additionally, existing prediction approaches only provide basic modelling constructs for concurrency modelling leaving the creation of complex structures to software architects. Concurrent software systems are especially complex, hard to model and implement. Therefore, goal-oriented abstractions

are desirable for such systems. Several approaches exist addressing these issues partially. Lee proposed to use modelling constructs for concurrency patterns [14] to increase understandability of concurrency, communication, and synchronisation within a software architecture. Similarly, Spitznagel and Garlan [20] used connectors to extract communication aspects from components. However, both approaches focus on qualitative attributes, like deadlock-freedom, neglecting quantitative attributes, such as performance and reliability.

7 Conclusion

The approach introduced in this paper provides the basis for avoiding (or minimising) conflicts that may occur during the performance completion integration into architectural models. For this purpose, we map patterns to the development roles and affected model elements. The number of potential conflicts is decreased. Furthermore, we can focus the resolution on a small set of specific model elements. Additionally, we sketched configurable performance completions of concurrency patterns to enable developers to easily predict performance properties of different design alternatives. We have implemented the Chilies tool [15] to provide an initial prototypical implementation of the ideas presented in this paper. An extension for full conflict resolution is planned in the next step.

For the future, we plan to investigate the connector completion category more deeply. The support for automatic connector completions generation is a challenging issue, especially in case of more complex communication strategies (e.g., push-pull pipes and filters). Consequently, we have to investigate the sequences of the connector components, based on communication style driven connector configurations. In a area of code generation, sequences of completion code generation have impact on a resulting code, therefore we need to provide methods for connector components sequence generation and configuration. The another open issue is to investigate critical section scopes to analyse impact of the locking strategies on the performance.

References

[1] S. Becker. *Coupled Model Transformations for QoS Enabled Component-Based Software Design*. Dissertation, University of Oldenburg, Germany, January 2008.

[2] S. Becker, H. Koziolek, and R. Reussner. The Palladio Component Model for Model-Driven Performance Prediction: Extended version. *Journal of Systems and Software*, 2008. In Press, Accepted Manuscript.

[3] Steffen Becker, Antonio Brogi, Ian Gorton, Sven Overhage, Alexander Romanovsky, and Massimo Tivoli. Towards an Engineering Approach to Component Adaptation. In *Architecting Systems with Trustworthy Components*, volume 3938 of *Lecture Notes in Computer Science*, pages 193–215. Springer, 2006.

[4] Shiping Chen, Yan Liu, Ian Gorton, and Anna Liu. Performance Prediction of Component-based Applications. 2005.

[5] Krysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming*. 2000.

[6] B. P. Douglass. *Real-Time Design Patterns*. Object Technology Series. Addison-Wesley Professional, 2002.

[7] Thomas Goldschmidt and Guido Wachsmuth. Refinement transformation support for QVT Relational transformations. In *3rd Workshop on Model Driven Software Engineering (MDSE 2008)*, 2008.

[8] J. Happe, H. Friedrich, S. Becker, and R. H. Reussner. A Pattern-Based Performance Completion for Message-Oriented Middleware. In *Proceedings of the 7th International Workshop on Software and Performance (WOSP '08)*, pages 165–176. ACM, 2008.

[9] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Longman Publishing, 1979.

[10] Lucia Kapova and Thomas 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.

[11] Lucia Kapova, Barbora Zimmerova, Anne Martens, Jens Happe, and Ralf H. Reussner. State dependence in performance evaluation of component-based software systems. In *Proceedings of the 1st Joint WOSP/SIPEW International Conference on Performance Engineering (WOSP/SIPEW '10)*, New York, NY, USA, 2010. ACM.

[12] H. Koziolek, S. Becker, J. Happe, and R. Reussner. *Model-Driven Software Development: Integrating Quality Assurance*, chapter Evaluating Performance and Reliability of Software Architecture with the Palladio Component Model. IDEA Group Inc., December 2007. To Appear.

[13] Heiko Koziolek and Jens Happe. A QoS Driven Development Process Model for Component-Based Software Systems. In Ian Gorton, George T. Heineman, Ivica Crnkovic, Heinz W. Schmidt, Judith A. Stafford, Clemens A. Szyperski, and Kurt C. Wallnau, editors, *Component-Based Software Engineering, 9th International Symposium, CBSE 2006, Västerås, Sweden, June 29 - July 1, 2006, Proceedings*, volume 4063 of *Lecture Notes in Computer Science*, pages 336–343. Springer, 2006.

[14] E. A. Lee. The Problem with Threads. *IEEE Computer*, 39(5):33–42, May 2006.

[15] Lucia Kapova. CHILIES: Automated Model Completions, 2010.

[16] Object Management Group (OMG). Model Driven Architecture - Specifications, 2006.

[17] R. Reussner, S. Becker, J. Happe, H. Koziolek, K. Krogmann, and M. Kuperberg. The Palladio Component Model. Technical Report 2007-21, Universität Karlsruhe (TH), 2007.

[18] Ralf H. Reussner, Heinz W. Schmidt, and Iman Poernomo. Reliability Prediction for Component-Based Software Architectures. *Journal of Systems and Software – Special Issue of Software Architecture – Engineering Quality Attributes*, 66(3):241–252, 2003.

[19] Douglas C. Schmidt, Hans Rohnert, Michael Stal, and Dieter Schultz. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, Inc., 2000.

[20] B. Spitznagel and D. Garlan. A Compositional Formalization of Connector Wrappers. In IEEE, editor, *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*, pages 374–384, Los Alamitos, CA, May 2003. IEEE Computer Society.

[21] T. Verdickt, B. Dhoedt, F. Gielen, and P. Demeester. Automatic Inclusion of Middleware Performance Attributes into Architectural UML Software Models. *Transactions on Software Engineering*, 31(8):695–771, 2005.

[22] M. Woodside. Tutorial Introduction to Layered Modeling of Software Performance, May 2002. Last retrieved 2008-01-13.

[23] Murray Woodside, Greg Franks, and Dorina C. Petriu. The Future of Software Performance Engineering. In *Proceedings of ICSE 2007, Future of SE*, pages 171–187. IEEE Computer Society, Washington, DC, USA, 2007.

[24] Xiuping Wu and Murray Woodside. Performance Modeling from Software Components. *SIGSOFT Softw. Eng. Notes*, 29(1):290–301, 2004.