Layered Patterns in Modelling and Transformation of Service-based Software Architectures

Claus Pahl and Ronan Barrett

Dublin City University
School of Computing
Dublin 9, Ireland
[cpahl|rbarrett]@computing.dcu.ie

Abstract. Service-oriented architecture is a recent paradigm for architectural design. The software engineering aspects in this context, that have not been sufficiently addressed, are software evolution and software migration. Architectures are of great importance if large software systems change. Architectural transformations can guide and make this change controllable. In this paper, we present a modelling and transformation method for service-based software systems. Architectural configurations, expressed through architectural patterns, form the core of an underlying specification and transformation calculus. Patterns on different levels of abstraction form transformation invariants that structure and constrain the transformation process. We explore the role layered patterns can play in modelling and as invariants for transformation techniques.

Keywords: Service-oriented Architecture, Service Processes, Architecture Specification, Architecture Transformation, Web Services.

1 Introduction

The development of distributed software systems based on service architectures is rapidly gaining momentum. *Service-oriented architecture* (SOA) is emerging as a new paradigm for the architectural design of widely distributed software systems, supported by platforms such as the *Web Services Framework* (WSF) [1]. Due to the ubiquity of the Web, the WSF platform and SOA paradigm can be expected to play a major role in the future of software development.

Architectural design is about separating computation from communication. In service-based, distributed environments such as the WSF, a notion of processes is central to capture service composition and interaction between services. We present an architectural model and engineering techniques to support, firstly, modelling and specification of services and service-based processes and, secondly, property-preserving transformations of service-based architectures.

Our solution is an approach to the *architectural configuration* of services, based on formal modelling of service communication and interaction processes.

One of the distinguishing features of our approach is a *three-layered architecture model* addressing different architectural levels of abstraction. Each layer is supported through a *pattern-based modelling approach*. A service-based architectural configuration calculus that combines patterns and process behaviour in architectures forms the backbone of this approach. The exploration of the role of layered process-oriented patterns is the central objective here.

Formality is required in this framework to obtain precise and unambiguous specifications of process-based service architectures and to complement specification by analysis and reasoning facilities. In particular architectural change and evolution requires a technique for process-oriented property-preserving transformations. Various formal approaches to the representation of processes have been suggested in the past. *Process calculi* such as the π-calculus [2] are suitable frameworks for architectural configurations due to their abstraction from service

A number of different modelling approaches exist, using different formalisms, e.g. [5, 4] using Petri nets. We use the π-calculus as the basis, which helps us to define a notation for service-based architectural configuration. The π-calculus, a calculus for mobile processes, is particularly useful due to a similarity between mobility and evolution – both are about changes of a service in relation to its neighbourhood – which helps us to support architectural transformations.

We give some background and an introduction to our layered architecture model, called SAM, and our transformation calculus, called SACC, in Section 2. Pattern-based architecture modelling and specification, supported by the architecture configuration calculus SACC, is addressed in Section 3. Architectural transformations are defined in Section 4. Finally, we discuss related work in Section 5 and end with some conclusions in Section 6. A Web-based, service-oriented learning technology system serves as a case study throughout the paper.

2 Architecture Model and Specification Calculus

The objective of *software architecture* [3] is the separation of communication from computation. Architectures are about *components* (i.e. loci of computation) and *connectors* (i.e. loci of communication). This allows a developer to focus on structures and the dynamics between components separately from component implementation. Various *architecture description languages* (ADL) and modelling and development techniques have been proposed [6–8]. An architectural model captures common concepts found in a variety of architectural description languages: components provide computation, interfaces provide access to blackbox components, and connectors provide connections between components. In service-based architectures, the focus shifts towards the composition of services to processes and the overall *configuration of services and service processes*. Process and interaction behaviour is an essential part of modelling service architectures [3].

A *service* is usually defined as a coherent set of operations provided at a certain location [1]. A service provider makes an abstract service interface description available, which can be used by potential service users to locate and invoke this service. Services are often used 'as is' in single request-response in-

teractions. More recently, research has focussed on the *composition of services* to *processes* [1]. Existing services can be reused to form business or workflow processes. The *principle of architectural composition* that we look at here is *process assembly*. The discovery and invocation infrastructure – a registry or marketplace, where potential users can search for suitable services, and an invocation protocol – with the services and their clients form a service-oriented platform.

At the core of our architecture modelling and transformation technique is a conceptual architecture model. The objective of this architecture model is to capture the characteristics of service-based architectures. A layered conceptual *service architecture model* (SAM), that is tailored towards the needs of service- and process-oriented platforms, shall address the different levels of abstraction in service-based architectures:

- *Reference architectures* are high-level specifications representing common structures of architectures specific to a particular domain or platform.
- *Architectural design patterns* are medium-scale patterns – usually referred to as design patterns or architectural frameworks.
- *Workflow patterns* are process-oriented patterns that represent common business or workflow processes in an application domain.

Based on the architecture model SAM, we define a calculus for architectural specification and transformation – the *service-based architectural configuration calculus* (SACC) – that has features of an abstract architectural description language (ADL) at its core[1]. Its main aim is to support the architectural configuration of services. Two elements define our calculus:

- a *description notation* to capture architectural properties,
- *rules and techniques* for transformation.

The calculus is directly based on the π-calculus [2]. However, it adds a few combinators to express workflow and design patterns. A simulation notion from the π-calculus helps us to capture the idea of property-preservation and permitted structure and behaviour variations during transformation.

Our architectural process specification notation consists of basic process activities, activity combinators, and process abstractions. The basic element describing process activity is an *action*. Actions π are combined to *service process expressions*. Actions of a service can be divided into

- *invocations* **inv** x(y) of other services via channel x, which connects to the remote service, passing y as a parameter,
- *activations* receive $\mathbf{rcv}_x$(a) from other services and the dual reply $\mathbf{rep}_x$(b), with channel x and parameters a and b.

The *process combinators* are basic forms of *workflow patterns*:

- *Actions* π are primitive processes.

[1] This calculus does not qualify as an ADL since our focus is on processes and architectural configuration, neglecting interfaces and connector specifications.

– *Sequences* are represented as P_1;P_2, meaning that process P_1 is executed and the system transfers to P_2, where the next action is executed.
– *Exclusive Choice* means that one P_i $(i = 1, 2)$ from **choice** P_1,P_2 is chosen.
– *Multi-Choice* **mchoice** P_1,P_2 allows any number of the processes P_i $(i = 1, 2)$ to be chosen and executed in parallel.
– *Iteration* **repeat** P executes process P an arbitrary number of times.
– *Parallel composition* **par** (P_1,P_2) executes processes P_1 and P_2 concurrently.

Additionally, *restriction* **restr** m.P means that m is only visible in P. A(a_1, ..., a_n) = P_A is a *process abstraction*, where P is a process expression and the a_i are free variables in P. A local variable is introduced using **let** $x = \pi$ **in** P. Inaction is denoted by 0.

The *semantics* can be defined in terms of the π-calculus [2]. The language constructs can be directly mapped to π-calculus constructs. The basic actions are defined in terms of send $\overline{x}\langle y\rangle$ (for invocation **inv** and reply **rep**) and receive $x(y)$ (for receive **rcv**) of the π-calculus. The combinators are defined directly through their π-calculus counterparts, except the multichoice **mchoice** P_1,P_2, which is defined in terms of π-calculus-supported combinators as **choice** (A, B, **par** (A,B)) – essentially a parallel composition of all elements of the powerset of the **mchoice** argument list. The abstraction is the π-calculus abstraction.

3 Pattern-based Service Architecture Modelling

The *service-based architectural configuration calculus* SACC enables modelling and specification of pattern-based service architecture configurations. We will use an e-learning system called IDLE – the Interactive Database Learning Environment – to illustrate our approach [9]. IDLE is based on a Web software architecture that provides a range of educational services:

– It is a multimedia system that uses different mechanisms to provide access to learning content, e.g. Web server and a (synchronised) audio server.
– It is a composite, interactive system that integrates components of a database development environment (a design editor, a programming interface, and an analysis tool) into a teaching and learning context.
– It is a constructive environment in which learners can develop their database applications, supported by shared storage and workspace.

In this section, we introduce a pattern-based modelling method that is suitable for modelling architectural configuration and processes of service architectures at different levels of abstraction, using IDLE for illustration. Our hypothesis is that the presented service process calculus SACC provides a suitable specification technique for modelling service architectures for all pattern types.

3.1 Patterns and Abstraction Levels

Architecture and *design patterns* are recurring solutions to software design problems [10]. These patterns are about the design and interaction of objects, as

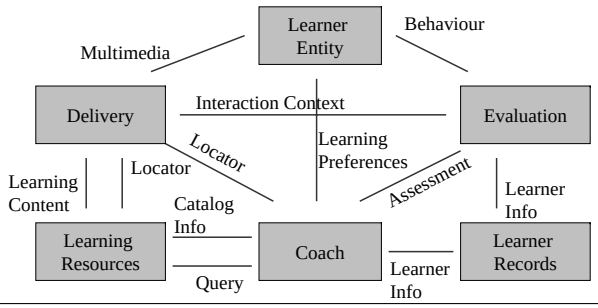

Fig. 1. Reference Architecture – Overview of the Reference Architecture LTSA.

well as providing a communication platform concerning reusable solutions to commonly encountered design problems. Patterns at different levels of abstraction – reference architectures, architectural design patterns, and workflow patterns – form an essential part of our service-specific architectural transformation approach. We cover the three layers of the architecture model SAM with our notation. Workflow operators for service processes are directly integrated as operators. An architectural design pattern expressing service interaction patterns can be formulated as an expression of a number of concurrently executing processes. Reference architectures can be modelled at the level of abstractions.

Reference Architectures. *Reference architectures*, if they exist for a platform or a domain, can play an essential role in the architectural definition of a software system. They often emerge in an abstracted and standardised form from successful architectural assemblies. Reference architectures define accepted structures that help us to built maintainable and interoperable systems.

In the context of educational software systems, our case study domain, the IEEE-defined *Learning Technology Standard Architecture* (LTSA) provides a service-oriented reference architecture [11], see the UML-style class diagram in Fig. 1. Six central components such as Delivery or Coach are identified. These components provide services to other components, e.g. the Delivery component provides a Multimedia delivery service to the LearnerEntity. These services are usually related to processing data in different types of media.

Besides domain-specific architectures, platform-specific reference architecture are important. Examples of classical Web-based architectures are *client-server* architectures or *three-tiered architectures*.

Design Patterns. *Design patterns* are recognised as important building blocks in the development of software systems [10]. Their purpose is the identification of common structural and behavioural patterns. A rich set of design patterns has been described, which can be used to structure a software design at an intermediate level of abstraction. Usually, *architectural patterns* (such as model-view-controller) are distinguished from *design patterns* (such as factory, composite, or

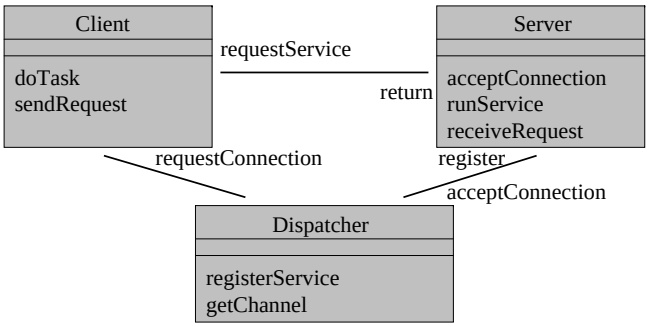

Fig. 2. Pattern – the Client-Dispatcher-Server Architectural Design Pattern.

```
Learner  = repeat ( inv requestEducServ = requestConnection();
                    inv res = requestEducServ(resId)  )
Delivery = inv registerEducServ(id);
           repeat ( rcv_acceptConnection(c); rcv_requestEducServ(s);
                    rep_requestEducServ(run(s))  )
Coach    = choice (
               choice ( rcv_registerEducServ(id); rcv_unregisterEducServ(id) )
               repeat ( rcv_requestConnection();
                    let c = getChannel()
                    in par ( inv acceptConnection(c); rep_requestConnection(c) ) ) )
```

Fig. 3. Specification – Educational Service (EducServ) Registration and Provision in IDLE based on the Client-Dispatcher-Server Design Pattern.

iterator). We see both forms of patterns as intermediate-level constraints on a system architecture, i.e. on services and on their interaction patterns.

Design patterns also play a role in the design of Web services architectures [12]. An example of an architectural design pattern is the *client-dispatcher-server pattern* [12]. The pattern architecture with its interactions is visualised in Fig. 2 in UML-style representation. The client requests a service in the pattern. The server is the provider of the service. The dispatcher is the mediator between client and server. Servers register their services with the dispatcher and clients request connection channels to servers in order to use the services.

Example ***1*** *In IDLE, a learner requests content from a resources server. The IDLE specification in SACC, Fig. 3, is based on the client-dispatcher-server pattern, Fig. 2, with the learner (as client), a coach (as dispatcher), and the resources and delivery subsystem (as server). The learner is a client invoking services of the delivery (request a connection and an educational service). The coach handles the service registration (from the delivery) and forwards the delivery channel (provides by the delivery component) to the learner. Passing channel*

WorkSpace = **choice** (
repeat ($\mathbf{rcv}_{\text{retrieve}}$(resId); **inv** provide(res)) ,
repeat ($\mathbf{rcv}_{\text{store}}$(resId, res)))

Fig. 4. Specification – Specification of the IDLE Storage and Workspace Service.

names over channels is a typical example of the π-calculus ability to model dynamic infrastructures. The learner then uses the provided channel to use the delivery component's educational service.

Abstracted pattern definitions such as *client-dispatcher-server* can act as building blocks in higher-level architectural specifications. Patterns are defined as process expressions and made available as process abstractions. These macro-style building blocks also form a pattern repository. A detailed discussion of pattern-based specification of IDLE can be found in [13].

Workflow patterns. *Workflow patterns* are small-scale process patterns [14]. Workflow patterns relate to connector types that are used in the composition of services – we provide them as built-in operators. An example of a workflow pattern is the Unix-style pipe, which is similar to a sequencing workflow pattern. Workflow patterns are small compositions of basic activities. Workflow patterns and their implementation in Web services architectures are described in [15].

Example 2 The multichoice operator is an example for process compositions [15]:

$$\mathbf{mchoice}(\text{Lecture}, \text{Tutorial}, \text{Lab})$$

expresses that any selection of the IDLE services Lecture*,* Tutorial*, and* Lab *can be used concurrently, e.g. a user can use lecture and lab services in parallel.*

To identify these workflow patterns in the architecture specification is important since often not all pattern are supported by the implementation languages. Then, workarounds based on architectural transformations have to be found.

$$\mathbf{choice}(A, B, C, \mathbf{par}(A, B), \mathbf{par}(A, C), \mathbf{par}(B, C), \mathbf{par}(A, B, C))$$

is an equivalent workaround to the multichoice workflow, needed if the implementation language does not support the multichoice pattern **mchoice**(A, B, C) – which is the case with some WS-BPEL implementations [15].

3.2 Modelling Service Architectures

Modelling service-oriented architectures starts with the identification of services. Two cases can be distinguished:

- Some of the components of a system will clearly exhibit service character – an SQL execution element, which is part of the IDLE lab resources and delivery subsystem, is an example.

– Some components might not be implemented as services, but could easily be wrapped up if required. An example of this category is a storage and workspace feature.

In our case study, the problem is re-engineering of a legacy system into a service-based system. The existing architecture – even though not adequately designed and documented – provides a starting point for service identification.

Example 3 We use the LTSA reference architecture as the starting point for the service-based modelling of IDLE due to the LTSA's SOA character. We, however, realise the storage and workspace function, which could have been integrated into either learning resources or learner records in terms of the LTSA, as a separate service. This IDLE feature can be specified as a service process, see Fig. 4. The workspace service either deals with incoming retrieval or storage requests.

Once all services have been identified, the connections and interactions between services have to be modelled. We propose a top-down method starting with reference architectures, followed by architecture and design patterns and finally workflow patterns. Subsystems and composite components of high-level architectures are refined down to the workflow level. For instance, top-level LTSA services can be internally composed of small-scale interacting services. The presented modelling technique allows us to adequately address the modelling aspects of a service-based educational software system. We have presented this technique within a method for top-down, pattern-based layered modelling.

4 Transformation

Software architecture addresses more than the high-level system design. Software change resulting from maintenance and evolution is equally important. We focus on architecture transformations – a central software change technique. Often, architectural transformations are a necessity. Interoperability can be a transformation objective. For instance, a new reference architecture might need to be adopted. Another objective can be to accommodate changes in the interface and interaction processes of individual services. Workflow pattern are often transformed if implementation restrictions have to be dealt with.

A central objective of architecture transformation is to implement the planned changes, but also to preserve existing properties. Here, the existing service processes shall be preserved, i.e. process expressions act as invariants of the transformation. These processes are expressed as patterns at different levels of abstraction. While the idea of preserving patterns at all layers is obvious, a verifiable transformation technique is needed. A notion of simulation shall capture the ideas of equivalence and refinement of services and service processes – an essential element of the modelling aspect.

A prerequisite for the transformation is the explicit architecture specification of the existing system. A complete specification is not necessary; accuracy and level of preservation of the transformation, however, depend on the degree of

detail and number of patterns identified. In IDLE, we have for instance analysed an inadequately documented system to extract structures and patterns.

4.1 Simulation and Transformation Rules

Our transformation technique is based on a notion of simulation and on simulation-based transformation rules. It has to address the needs of layered pattern-based models and the focus on patterns as transformation invariants.

- Reference architectures. Each service abstraction is mapped to a service abstraction in the new architecture. The transformation objective determines whether the service process definition will have to be changed. The transformation is subject to invariants, i.e. pattern preservation.
- Architectural design patterns. Often, interaction processes needs to be changed to accommodate new or modified service functionality. Ideally, newly emerging patterns the service participates in will simulate the original patterns.
- Workflow patterns. Workflow pattern transformations can often be handled automatically in architecture implementations.

Property preservation is a central goal of our architecture transformations. A simulation notion shall capture service process pattern preservation in the transformation technique. A *simulation* definition, adopted from the π-calculus, satisfies the pattern preservation requirement for the processes that we envisage:

> A process Q *simulates* a process P if there exists a binary relation $\mathcal{S}$ over the set of processes such that if whenever $P\mathcal{S}Q$ and $P \xrightarrow{m} P'$ then there exists Q' such that $Q \xrightarrow{n} Q'$ and $P'\mathcal{S}Q'$ for service processes n and m.

This simulation definition expresses when a process Q based on service expressions n preserves, or simulates, the behaviour of a process P based on service expressions m. The services n and m can be unrelated, as this definition is about observable behaviour.

In order to automate transformation support based on this simulation definition, a constructive theorem supporting this definition is required. This will be the basis of a transformation rule which allows the verification of preservation and the automation of the transformation. In [16], we have developed a constructive simulation test based on the construction of transition graphs for the process expressions of the SACC calculus.

Since usually not the entire specified behaviour should be preserved, we have introduced the notion of patterns to capture common behavioural aspects that need to be preserved. Patterns at different levels of abstraction identify reliable and maintainable interaction patterns between services. These are ideally preserved. Central in our transformation technique is, therefore, the following *transformation rule*, which associates patterns and simulation:

> Given an architecture specification S in SACC, create an architecture specification S' as follows. For each abstraction A in S (apply this rule

recursively from top to bottom), *map A to A'* where A' is another abstraction such that for any pattern P, which A participates in, A' *simulates* P' with $P' = P[A/A']$, i.e. A' substitutes A in P. P is replaced by P' to cater for renaming of abstractions.

The determination of an invariant, the pattern P, is a common, but often non-trivial problem. This problem can be alleviated through domain-specific patterns. We will address this methodological aspect below.

4.2 Applying Pattern-preserving Transformations

We will demonstrate the adoption of a new reference architecture, the LTSA, on the highest level of abstraction for the IDLE system. The transformation aim is interoperability of IDLE services and components with other LTSA-specified components and reuse. This interoperability objective, however, can have an impact on all levels of abstraction. For instance, the SCORM Run Time Environment standard prescribes interfaces for learning technology objects, which would have to be reflected in service interfaces here.

*Example **4** The starting point for the transformation is the architecture specification of an existing system – in our case IDLE in its current form. IDLE on the highest level of abstraction is a parallel composition of composite processes:*

$$\text{IDLE} = \textbf{par}\ (\ \text{Learner}, \text{Delivery}, \text{StudentModel}\\ \text{PedagogyModel}, \text{Workspace}, \text{Evaluation}, \ldots\ )$$

where each top-level service is an abstraction of a process expression based on other, more basic services. Some of these are already similar to LTSA components – we have indicated this fact by using the similar names – others such as StudentModel *and* PedagogyModel *have no direct counterpart in the LTSA. Several different combinations of individual services can form patterns; these might actually overlap. We will discuss an example later on.*

The first transformation step is to describe IDLE's architectural characteristics – ideally in terms of LTSA to simplify the transformation, see Fig. 1.

*Example **5** The client-server-dispatcher pattern, see Fig. 2, is not identical to the structure that can be found in the IDLE system, see Fig. 3. We have added the interaction with the resources server. The pattern itself as an identifiable pattern is nonetheless worth preserving and is, thus, one of the invariants. In our case, the* client-dispatcher-server *pattern:*

par (Client, Dispatcher, Server)

is simulated *by the composite IDLE process:*

par (LearnerEntity, Coach, Delivery)

resulting from the composition of learner, coach, and resources and delivery subsystems of the IDLE reformulation in LTSA terminology. This means that the pattern is a good abstraction of IDLE functionality that needs to be preserved.

LTSA is a high-level pattern. In IDLE, we add functionality. This architectural change arises from the workspace service integration into IDLE.

*Example **6** The explicit storage and workspace service, see Fig. 4, requires the services* LearnerEntity *and* Delivery *to be modified in their interaction patterns. Again, the pattern shall be the invariant of the transformation, but some refinements – constrained by the simulation definition – need to be made to accommodate the added service within the system.*

In order to identify workflow patterns that need to be preserved, these can easily be identified due to their implementation as operators in the notation.

*Example **7** The specification of the IDLE educational service system based on the client-dispatcher-server architectural design patterns in Fig. 3 based on Fig. 2 is defined in terms of workflow patterns. The* Learner *is based on a sequence of activities. The* Coach *is based on choice in the first part, and a concurrent split and merge in the second part. These are candidates for invariants.*

The transformation task is to transform IDLE into LTSA-IDLE – an architectural variant of IDLE with LTSA-conform interfaces and interaction processes.

*Example **8** In the transformation, we need to consider the source, the invariant, the target construction, and the preservation proof:*

- Source*: The starting point of the transformation is the original IDLE specification. Since in our case a full specification did not exist, we analysed the system and extracted central features. The high-level architecture is given in Example 4 and some detailed excerpts are presented in Figs. 3 and 4.*
- Invariant*: The invarant is determined by patterns on different levels of abstraction. The LTSA determines the high-level architecture. We focus here on the client-dispatcher-server pattern as the architectural pattern invariant as explained in Example 5.*
- Target Construction*: The LTSA-based architecture specification of some IDLE services – which is the transformation result – can be found in Fig. 5. It is constructed based on our transformation rule as follows. At the reference architecture level, IDLE is mapped to LTSA-IDLE where the merger of* StudentModel *and* PedagogyModel *simulates the* Coach*. At the architectural design pattern level, the parallel composition of the individual components is changed at the subcomponent level (*Coach*) to reflect the merger.*
- Simulation and Preservation*: The invariants – LTSA and client-dispatcher-server – are two patterns that have to be simulated by the new architecture:*
  - *We have adapted our original terminology to the LTSA terminology. For instance,* Learner *becomes* LearnerEntity*. The two components* StudentModel *and* PedagogyModel *are merged into* Coach*, i.e. the model components were abstracted by a single* Coach *interface, which results in the LTSA pattern being simulated.*

LearnerEntity	= **inv** preferencesInfo = getPreferences(); **inv** setPreferences(alter(preferencesInfo)); **inv** learnResource = multimedia()
Coach'	= **repeat** (**choice** ($\mathbf{rcv}_{\text{getPreferences}}()$; $\mathbf{rep}_{\text{getPreferences}}$(prefInfo), $\mathbf{rcv}_{\text{setPreferences}}$(preferencesInfo), $\mathbf{rcv}_{\text{getLearnerInfo}}$(id); $\mathbf{rep}_{\text{getLearnerInfo}}$(info), **inv** uri = locator(resource)))
Delivery'	= $\mathbf{rcv}_{\text{locator}}$(uri); **inv** learnResource = retrieveResource(uri); $\mathbf{rep}_{\text{multimedia}}$(learnResource)
LearningResources	= $\mathbf{rcv}_{\text{retrieveResource}}$(uri); $\mathbf{rep}_{\text{retrieveResource}}$(retrieve(uri))
LearnerRecords	= $\mathbf{rcv}_{\text{getLearnerInfo}}$(id); $\mathbf{rep}_{\text{getLearnerInfo}}$(info(id))

Fig. 5. Transformation – Resulting Adaptive Delivery in IDLE Architecture (selected components and services) based on the LTSA.

- *The new* Coach' *service handles the interaction with the learner and pedagogy model components internally. The original* Coach *specification from Fig. 3 has been adapted to reflect this fact. The structural and behavioural properties of the client-dispatcher-server pattern* P := **par**(Client, Dispatcher, Server) *are still intact, i.e. the pattern is preserved accroding to the transformation with pattern* P *and the original* Coach *adapted to* Coach'. *The three pattern components are still present and the externally visible interaction behaviour is the same*[2].

The specification in Fig. 5 describes the adaptive delivery of resources. After updating preferences by interacting with the coach, the learner entity requests and receives learning resources via a multimedia channel from the delivery service. The learning resources service retrieves the actual content for the delivery service, which in turn delivers it to the learner entity.

In our method, *design patterns* that can be identified in an existing system, such as the original IDLE, should be *invariants of the architectural transformation*. In [13], we have shown that design patterns that were identified for object-based systems also occur in service-oriented architectures. This method can be supported by transformation tools. The architect provides the source system model and identifies preservable patterns from the model patterns and, if necessary, renamings and non-standard transformations. The tool would then carry out the transformation by applying the transformation rule subsitutions to patterns and discharging the preservation proof obligations.

The combination of the most frequent patterns seem to be domain-specific, as our investigation indicates [13]. Examples of frequently occurring design patterns

[2] The formal proof is based on a constructive simulation test developed in [16], which is beyond the scope of this paper.

that we have identified in IDLE, other learning technology systems, and also the LTSA are the factory, proxy, observer, composite, and serialiser patterns. Other, less frequent patterns that we have found include the iterator and the strategy pattern. These common patterns could result in a domain-specific formulation of patterns such as LearnerEntity-Coach-Delivery (see Example 5) and a repository of domain-specific patterns, which would help software architects in the difficult task of identifying invariants of the transformation.

5 Related Work

Some ADLs are similar to our approach in terms of formality and their focus on processes. Darwin [17] is a π-calculus based ADL. Darwin focuses on component-oriented development approach, addressing behaviour and interfaces. Restrictions based on the declarative nature of Darwin make it rather unsuitable for the design of service-based architectures, where both binding and unbinding on demand are required features. Wright [18] is an ADL based on CSP as the process calculus. Wright supports compatibility and deadlock checks through formalised specifications, based on explicit connector types. This is an aspect that we have neglected here, but that could enable further analysis techniques, if we introduced typed channels. In [19], the formal foundations of a notion of behaviour conformance are explored, based on the π-calculus bisimilarity relation. We chose the π-calculus as our basis, since it caters for mobility, and, consequently, allows us to address architecture evolution and transformation [8]. Mobility allows us to deal with changes in the interaction infrastructure. The client-dispatcher-server pattern is an example where a new channel is dynamically formed. On the metalevel, architecture transformation also means controlled changing of architectural structures. The impact of observational semantics based on states denoting a family of bisimilar configurations has yet to be investigated in detail.

Patterns have recently been discussed in the context of Web service architectures [12, 15]. In [15], a collection of workflow patterns is compiled. We have based our operator calculus on this collection, aiming at a support for most of the patterns described. The client-dispatcher-server pattern that we have identified in our IDLE system is also discussed in [12]. Other patterns that we have mentioned mainly originate from [10].

A recent software architecture approach for service-based systems is Model-Driven Architecture (MDA) [20]. MDA emphasises the importance of modelling and transformations. The latter are, in contrast to our framework, part of the modelling process between modelling layers. Our framework addresses the transformation of multi-layered architecture specifications. While MDA is vertically oriented, i.e. mapping from abstract domain models to more concrete platform and implementation models, we follow a more horizontal transformation approach on the level of architecture models.

6 Conclusions

A new architectural design paradigm such as service-oriented architecture (SOA) requires adequate methodological support for design, maintenance, and evolution. While an underlying deployment platform exists in the form of the Web Services Framework (WSF), an engineering methodology and techniques are still largely missing. We have presented a layered architecture model (SAM) that captures architectural structures at different levels of abstraction through patterns. A calculus (SACC) allows the process behaviour in architectures and architectural configurations to be captured. Interaction behaviour and composite processes within the architecture have turned out to be an essential aspect for the development and maintenance of service-based systems.

The importance of modelling for SOA has been recognised – and has resulted in the development of Model-Driven Architecture (MDA) as an approach to support the design of service-based software systems. We have focussed on layered pattern-based process modelling and architectural configuration – two aspects that can complement the MDA approach. The formality of our approach satisfies the automation requirements of MDA and even adds reasoning aspects. We are currently working on an architectural configuration tool for Web services that supports workflow and architectural patterns in the specification and that automatically translates these platform-independent specifications into Web service-specific notations such as WS-BPEL. Our emphasis here was on the applicability of the method by demonstrating the usefulness for a service-based learning technology system. We have investigated the role that layered pattern modelling can play for service-oriented architecture. The purpose of the SAM model and the SACC calculus is to provide a support technique for this modelling.

We have applied the presented framework in the ongoing design, maintenance, and evolution of the IDLE environment. The transformation technique was only outlined in its principles – our objective was the motivation of the method. In general, some of the architectural engineering activities can be better supported. The pattern framework could be extended to include distribution patterns, which would complement the existing layered functional patterns. A critical aspect of the approach is the reliance on the quality of the architectural description of the original system and the adequacy of the identified patterns. Transformations depend on the detail of the input architecture and the patterns that define the transformation invariant. The extraction of a system's architecture and the correct identification of intended patterns for undocumented systems is a difficult aspect that, although essential for the success has been addressed only through the idea of domain-specific patterns. Re-engineering approaches for the architectural level can provide further solutions here.

References

1. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services – Concepts, Architectures and Applications.* Springer-Verlag, 2004.

2. D. Sangiorgi and D. Walker. *The π-calculus – A Theory of Mobile Processes*. Cambridge University Press, 2001.
3. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice (2nd Edition)*. SEI Series in Software Engineering. Addison-Wesley, 2003.
4. B.-H. Schlingloff and A. Martens and K. Schmidt. Modeling and Model Checking Web Services. *Electronic Notes in Theoretical Computer Science: Issue on Logic and Communication in Multi-Agent Systems*, 126:3-26. 2005.
5. R. Dijkman and M. Dumas. Service-oriented Design: A Multi-viewpoint Approach. *Intl. Journal of Cooperative Information Systems*, 13(4):337-368. 2004.
6. N. Medvidovic and R.N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. In *Proceedings European Conference on Software Engineering / International Symposium on Foundations of Software Engineering ESEC/FSE'97*, pages 60–76. Springer-Verlag, 1997.
7. C. E. Cuesta, M. del Pilar Romay, P. de la Fuente, and Manuel Barrio-Solorzano. Architectural Aspects of Architectural Aspects. In R. Morrison, B.C. Warboys, and F. Oquendo, editors, *2nd European Workshop on Software Architecture EWSA 2005*. Springer LNCS 3047, 2005.
8. F. Oquendo, B.C. Warboys, R. Morrison, R. Dindeleux, F. Gallo, H. Garavel, and C. Occhipinti. ArchWARE: Architecting Evolvable Software. In R. Morrison, B.C. Warboys, and F. Oquendo, editors, *2nd European Workshop on Software Architecture EWSA 2005*. Springer LNCS 3047, 2005.
9. C. Pahl, R. Barrett, and C. Kenny. Supporting Active Database Learning and Training through Interactive Multimedia. In *Proc. Intl. Conf. on Innovation and Technology in Computer Science Education ITiCSE'04*. ACM, 2004.
10. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Design*. Addison Wesley, 1995.
11. IEEE Learning Technology Standards Committee LTSC. *IEEE P1484.1/D8. Draft Standard for Learning Technology - Learning Technology Systems Architecture LTSA*. IEEE Computer Society, 2001.
12. N.Y. Topaloglu and R. Capilla. Modeling the Variability of Web Services from a Pattern Point of View. In L.J. Zhang and M. Jeckle, editors, *Proc. European Conf. on Web Services ECOWS'04*, pages 128–138. Springer-Verlag, LNCS 3250, 2004.
13. C. Pahl and R. Barrett. Towards a Re-engineering Method for Web Services Architectures. In *Proc. 3rd Nordic Conference on Web Services NCWS'04*. 2004.
14. W.M.P. van der Aalst, B. Kiepuszewski A.H.M. ter Hofstede, and A.P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14:5–51, 2003.
15. M. Vasko and S. Duskar. An Analysis of Web Services Flow Patterns in Collaxa. In L.J. Zhang and M. Jeckle, editors, *Proc. European Conf. on Web Services ECOWS'04*, pages 1–14. Springer LNCS 3250, 2004.
16. C. Pahl. An Ontology for Software Component Matching. In M. Pezzè, editor, *Proc. Fundamental Approaches to Software Engineering FASE'2003*, pages 6–21. Springer-Verlag, LNCS 2621, 2003.
17. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In W. Schäfer and P. Botella, editors, *Proc. 5th European Software Engineering Conf. (ESEC 95)*, Springer LNCS 989, pages 137–153. 1995.
18. R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.
19. C. Canal, E. Pimentel, and J.M. Troya. Compatibility and inheritance in software architectures. *Science of Computer Programming*, 41:105–138, 2001.
20. Object Management Group. *MDA Model-Driven Architecture Guide V1.0.1*. OMG, 2003.