e-Informatica Software Engineering Journal, Volume 4, Issue 1, 2010

Pattern-Based Software Architecture for Service-Oriented Software Systems

Claus Pahl*, Ronan Barrett*
*School of Computing, Dublin City University
Claus.Pahl@computing.dcu.ie, Ronan.Barrett@computing.dcu.ie

Abstract

Service-oriented architecture is a recent conceptual framework for service-oriented software platforms. Architectures are of great importance for the evolution of software systems. We present a modelling and transformation technique for service-centric distributed software systems. Architectural configurations, expressed through hierarchical architectural patterns, form the core of a specification and transformation technique. Patterns on different levels of abstraction form transformation invariants that structure and constrain the transformation process. We explore the role that patterns can play in architecture transformations in terms of functional properties, but also non-functional quality aspects.

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 design paradigm and conceptual framework for distributed service-centric software systems, supported by platforms such as the *Web Services Framework* (WSF) [2]. Services are reusable software components that are explicitly described, published and provided at fixed locations. Due to the ubiquity of the Web, the WSF platform and SOA paradigm play a major role for software systems.

In service-centric distributed environments such as the Web services platform that allows services to be invoked using Internet protocols, a notion of workflow processes is central to capture service composition and interaction between services. We present techniques to support, firstly, modelling of services and service-oriented processes and, secondly, property-preserving transformations of service-oriented architectures. In contrast to a variety of architecture approaches that focus primarily on static, structural properties, we concentrate on dynamic dependencies in the form of interaction processes between services. Our solution is an approach to the *architectural transformation* of services, supporting the evolution of service-oriented architectures. Three aspects characterise our approach:

- Architecture modelling using hierarchical patterns. A *three-layered architecture model* addresses different levels of abstraction. Each layer is supported by a *pattern-based modelling approach* for service processes. A service-oriented architectural *configuration notation* that combines patterns and process behaviour in architectures forms the backbone. Patterns enhance reuse in SOA.
- Property-preserving architectural transformation. Based on the configuration notation as the abstract description language for source and target architectures, a *transformation technique* is developed. Patterns are considered as characteristics of a service architecture that are, due to the implied

reliability and maintainability, worth being preserved in transformations.
- Distribution and quality-of-service. We investigate the role of *distribution* for modelling and look at functional and non-functional service properties. The integration of *quality* aspects into modelling is important for the services platform, where providers and users are usually from different organisations.

We address the lack of behaviour and quality aspects in service-oriented architectural transformations. Our patterns capture essential behavioural service dependencies in the form of interaction process patterns and link these to quality properties. We utilise patterns to capture these properties and allow these properties to be preserved in transformations by identifying patterns as invariants. Formality is required to obtain unambiguous models of process-based service architectures and to complement modelling by analysis and reasoning facilities. Architectural change and integration require a technique for process-oriented property-preserving transformations.

We introduce our architecture model and transformation technique in Section 2. Pattern-based architecture modelling and specification, supported by the architecture configuration notation, is addressed in Section 3. Architectural transformations are defined in Section 4. Finally, we discuss related work and end with conclusions.

2. Architecture Model and Specification

Based on background definitions of service and software architecture, we now define the principles of our architecture model and the core notation.

2.1. Service-oriented Architecture

The objective of *software architecture* is the separation of computation and communication. Architectures are about *components* (i.e. loci of computation) and *connectors* (i.e. loci of communication). Various *architecture description languages* (ADL) and modelling techniques have been proposed [17]. An architectural model captures common concepts in architectural description: components provide computation, interfaces provide access and connectors provide connections between components. In service architecture, the main emphasis is on the composition of services to workflow processes and on the overall configuration of services and service processes. For instance [10], use *scenarios* – descriptions of interactions of a user with a system – to operationalise requirements and map these to a system architecture. We extend the notion of interaction and also consider system-internal interactions and allow interaction processes to be composite.

We focus on service architectures, i.e. service-oriented software architectures, here. A *service* is usually defined as a coherent set of operations provided at a certain location [2]. A service provider makes an abstract interface description available, which can be used by potential service users to locate and invoke this service. The Web Service platform provides description languages (WSDL) and invocation protocols (SOAP) for this purpose. Services are often used 'as is' in single request-response interactions. More recently, research has focused on the *composition of services* to *processes* [2]. Orchestration is the prevalent form of service composition. Existing services can be reused to form business or workflow processes. The *principle of architectural composition* that we look at here is *process assembly*.

2.2. An Architectural Configuration Notation

At the core of our architecture modelling and transformation technique is a conceptual architecture model. The objective of this conceptual architecture model is to capture the core layering and structuring principles of service-oriented architectures. The conceptual **service architecture model** (SAM), tailored towards the needs of service- and process-oriented platforms,

shall address the different abstraction levels and perspectives in service-oriented 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 data exchange-oriented workflow processes in an application domain.

Based on the architecture model, we define a notation for architectural specification – the **service-oriented architectural configuration** notation (SAC) – that has features of an abstract architectural description language (ADL). Two elements define our transformation technique: a *description notation* to capture architectural properties and *rules and techniques* for transformation.

Various formal approaches to the representation of processes have been suggested in the past, e.g. [6] using Petri nets. *Process calculi* such as the π-calculus [15, 13] are suitable frameworks for architectural configurations of service- and process-centric systems, i.e. support of modelling and transformation, due to their abstraction from service implementation and their focus on interaction processes. 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. Our notation is defined in terms of the π-calculus [15], but we want to firstly provide a less mathematical syntax and, secondly, allow the addition of further combinators to express workflow and design patterns. A simulation notion captures property-preservation and permitted structure and behaviour variations during transformation.

Our notation consists of process activities, combinators and abstractions, which are summarised in Fig. 1. The basic element describing process activity is an **action**. Actions π are combined to **service process expressions**. Actions of a service are primitive processes divided into invocations and activations. **Invocations** **inv** x(y) by a client of a service via channel x connects to the remote service, passing y as a parameter. **Activations** receive **rcv** x(a) from a provider from other services and the dual reply **rep** x(b), with channel x and parameters a and b. Based on actions, process combinators are

Actions:

π	::=	**inv** $x(y)$	Invocation
		rcv $x(a)$	Activation – Receive
		rep $x(b)$	Activation – Reply

Processes – workflow combinators:

P	::=	π	Action
		$P_1; P2$	Sequential Composition
		par (P_1, P_2)	Parallel Composition
		repeat (P)	Iteration
		choice (P_1, P_2)	Exclusive Choice
		mchoice (P_1, P_2)	Multi-Choice

Processes – other constructs:

P	::=	**let** $x = \pi$ **in**	Variable
		0	Inaction

Abstraction:

$$A(a_1, \ldots, a_n) \quad = \quad P_A \quad \text{with } a_1, \ldots, a_n \text{ are free in } P_A$$

Figure 1. Syntactical Definition of the SAC Notation

basic forms of workflow patterns. **Sequences** are represented as P_1; P_2 – 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 $(i = 1, \ldots, n)$ from **choice** $P_1, \ldots, P_n$ is chosen, **Multi-choice mchoice** $P_1, \ldots, P_n$ allows any number of the processes P_i $(i = 1, \ldots, n)$ to be chosen and executed in parallel. **Iteration repeat** P executes process P an arbitrary number of times. **Parallel composition par** $(P_1, \ldots, P_n)$ executes processes P_i concurrently. $A(a_1, \ldots, a_n) = P_A$ is a **process abstraction**, where P is a process expression and the a_i are free variables in P. A variable is introduced using **let** $x = \pi$ **in** P. **Inaction** is denoted by 0.

The semantics is defined in terms of the π-calculus [15], by mapping constructs directly to π-calculus constructs. The 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. Combinators are defined through their π-calculus counterparts, except multichoice **mchoice** P_1, P_2, which is defined 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 architectural configuration notation SAC enables the modelling of pattern-based service architecture configurations.

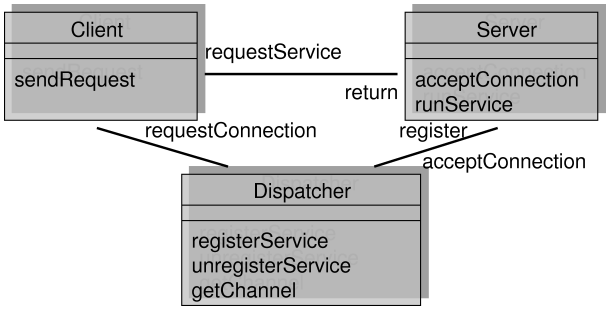
3.1. Patterns and Abstraction Levels

Architectural and *design patterns* are recurring solutions to software design problems [7]. Although originally proposed for object-oriented development, their applicability for service-based architectures has been demonstrated [18]. These patterns are about structure and interaction and provide reusable solutions to commonly encountered design problems. We use patterns at different levels of abstraction – reference architectures, architectural design patterns, and workflow patterns. We cover the three layers of the architecture model SAM. Workflow operators for service processes are directly integrated as operators. Architectural design patterns expressing service interaction patterns can be formulated as a number of concurrently executing processes. Reference architectures can be modelled at the level of abstractions.

Reference architectures, 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. Besides domain-specific architectures, which we will illustrate in the case study section, platform-specific reference architecture are important. Examples of classical Web-based architectures are *client-server* architectures or *three-tiered* architectures.

Design patterns are recognised as important building blocks in the development of software systems [7]. 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 iterator) as the former are linked to component frameworks. We see both forms 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 [18]. An example of an architectural design pattern in the Web services context is the *client-dispatcher-server* pattern [18]. The pattern architecture with its interactions is visualised in Fig. 2. The SAC notation adds behaviour specification to the static view of UML class diagrams. It is a textual description, similar to UML activity and interaction diagrams in purpose. We have used a UML class diagram to present the abstract service interface and the service connectivity. Pattern definitions such as *client-dispatcher-server* can




```
Client     = repeat (let requestServ = inv requestConnection()
                     in inv requestServ(resId))
Server     = inv registerServ(id);
             repeat (rcv acceptConnection(c); rcv requestServ(s);
                     rep requestServ(runService(s)))
Dispatcher = choice (
             choice (rcv registerServ(id), rcv unregisterServ(id)),
             repeat (rcv requestConnection();
               let c = getChannel()
               in  inv acceptConnection(c); rep requestConnection(c)))
```

Figure 2. *Pattern* – the Client-Dispatcher-Server Architectural Design Pattern

act as building blocks of complex architectures. Patterns are defined as process expressions and made available as process abstractions. These macro-style building blocks can also form a pattern repository.

Workflow patterns are small-scale process patterns [19] – often at the same level of abstraction as design patterns, but more focussed on data exchange. 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 sequencing workflow pattern. Workflow patterns are small compositions of activities. Workflow patterns for Web services architectures are described in [20].

To identify workflow patterns in an architecture specification is important since often not all patterns are supported by the implementation language.

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

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 [20].

3.2. Patterns and Quality

Patterns can influence a system's *quality characteristics* such as understandability or maintainability. For service-centric software systems specific properties arising from the often distributed and cross-organisational context are of central importance. The reliability of a system, the availability of services, and the individual service and overall system performance are often crucial.

- The qualtiy benefits of the client-dispatcher-server pattern are: composition is easy to *maintain*, as composition logic is contained at a single participant, the central dispatcher. *Low deployment overhead* as only the dispatcher manages the composition. Composition can consume participant services that are externally controlled. Web service technology enables the *reuse* of services.
- The main disadvantages are: a single point of failure at the dispatcher provides for *poor reliability/availability*. Communication bottlenecks at the dispatcher result in *restricted scalability*. Messages have considerable overhead for deserialisation and serialisation. A high number of often verbose messages

between dispatcher and clients/servers is sub-optimal and results in *poor performance*. All patterns have their advantages and disadvantages. Often, the qualities mutually affect each other negatively such as maintainability and performance. What is, however, important here is that the qualities associated to a given pattern are preserved during a transformation. The client-dispatcher-server pattern is typical for learning technology systems, for which maintainability and interoperability are central. Failure is not a highly critical problem and the number of users is predictable – which allows us to neglect two of the major disadvantages. Note, that these characteristics are associated to pattersn, but not part of our notation. For instance distribution is not part of our notation. We can use an annotation for the composition operators to indicate a distributed implementation if an extension is considered.

3.3. Case Study – Modelling Service Architectures

Our case study system is a learning environment called IDLE – the Interactive Database Learning Environment [14], which is based on object technology with a Web-based access interface. IDLE 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 an 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. Learners can develop database applications, supported by shared storage and workspace.

IDLE has been developed since 1996 in several stages. The consequence of this growth is a system without a designed architecture – an architecture that is even not explicitly captured and documented. However, the existing architecture is service-oriented (although not fully Web Service-based) and, consequently, is a suitable starting point for transformations. Evolving Internet technologies and frequently changing software developers are only two of the contributors to difficult maintenance. Besides achieving maintainability, interoperability and componentisation were reasons to choose a fully service-based architecture as the target.

Architecture modelling starts with the proposed three-layered approach. In the context of our case study domain, the IEEE-defined *Learning Technology System Architecture* (LTSA) provides a domain-specific service-oriented reference architecture [9], visualised in the UML-style class diagram in Fig. 3. Six central components such as Delivery or Coach are identified. These components provide services, e.g. the Delivery component provides a Multimedia delivery service to the LearnerEntity. These services are usually related to processing multimedia data. We use the LTSA reference architecture as a starting point for the re-engineering of IDLE.

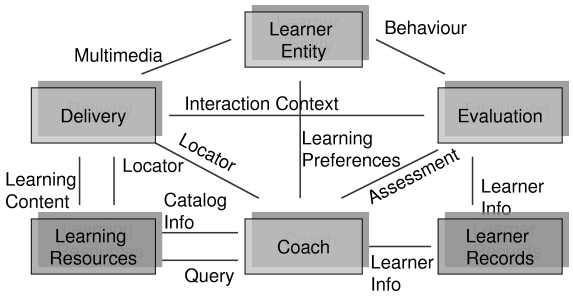


Figure 3. Overview of the LTSA *Reference Architecture*

In IDLE, a learner requests content from a resources server. The IDLE specification in SAC, Fig. 4, is based on the client-dispatcher-server design pattern, Fig. 2, with the learner (as client), a coach (as dispatcher), and the resources and delivery subsystem (as server). This specification captures a central behavioural property of IDLE, captured using the pattern. It is an extension of the pattern in terms of the IDLE application context that adds interaction with the Resources server to the Delivery component. Servers register their services with the dispatcher and clients request connection channels to servers in order to use the services. The learner is a client invoking services of the delivery (request a connection and an educational service). The coach is a broker and mediator that handles the service registration (from the

```
Learner  =  repeat (let requestEducServ = inv requestConnection()
                    in   inv requestEducServ(resId))
Delivery =  inv registerEducServ(id);
            repeat (rcv acceptConnection(c); rcv requestEducServ(s);
                    rep requestEducServ(run(s)); rcv locator(uri);
                    let learnResource = inv retrieveResource(uri)
                    in  rep multimedia(learnResource))
Coach    =  choice (
            choice (rcv registerEducServ(id), rcv unregisterEducServ(id)),
            repeat (rcv requestConnection();
              let c = getChannel()
              in  inv acceptConnection(c); rep requestConnection(c)))
```

Figure 4. *Specification* – Educational Service (EducServ) Registration and Provision in IDLE directly based on the Client-Dispatcher-Server Design Pattern

delivery) and forwards the delivery channel (provided by the delivery component) to the learner. Passing channel names over channels, as in the example, is typical for the notation's ability to model dynamic infrastructures. A learner uses the provided channel to access the delivery's educational service.

Another example of a design pattern is the *factory method pattern* – a creational pattern [7] that provides an interface for creating related objects without specifying their concrete classes. This pattern can be applied in IDLE for manipulating a variety of related persistent stores such as the learners records or adding/retrieving objects to/from a database such as a workspace feature.

Workflow patterns are the final architectural aspect. The multichoice operator denotes a process composition pattern. **mchoice**(Lecture, Tutorial, Lab) expresses that any selection of the IDLE services Lecture, Tutorial, and Lab can be used concurrently. We have realised the storage and workspace function, which could have been integrated into either learning resources or learner records, as a separate service. This IDLE feature can be specified as a complex service workflow process, see Fig. 5. The workspace service deals with incoming retrieval or storage requests.

Our service modelling notation needs a methodological context that covers modelling existing systems and transformations. Service-oriented architecture usually starts with the identification of services. Two cases can be distinguished:

- Some system components will exhibit service character – an SQL execution element, part of the IDLE lab resources and delivery subsystem, is an example.
- Some components could easily be wrapped up as services, if required. An example of this category is the IDLE storage and workspace feature.

Once all services have been identified, the connections and interactions between services have to be modelled. 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. The LTSA also determines the service-based modelling of IDLE due to the LTSA's SOA character. We have used a top-down approach to service identification as the first step of the transformation part.

The need to change, adapt and extend makes it clear that the original architecture cannot be fully preserved. An abstraction mechanism – in the form of patterns – answers the need to focus on essential, but not all architectural properties that should be preserved. Patterns not only identify common functional structures; they also have typical quality attributes associated with them. A central difficulty arises: how to identify suitable patterns. The collection of frequent patterns is often domain-specific, as our investigation indicates. Examples of frequently occurring design patterns in IDLE, other learning technology systems, and also the LTSA include the client-dispatcher-server

```
WorkSpace = choice (
                repeat (rcv retrieve(resId); inv provide(res)),
                repeat (rcv store(resId, res)))
```

Figure 5. *Specification* – Specification of the IDLE Storage and Workspace Service

pattern, but also the factory, proxy, observer, composite, and serialiser patterns [7]. Other, less frequent patterns include the iterator and the strategy pattern. These common patterns could result in a domain-specific formulation of patterns and a repository of domain-specific patterns, which would help software architects in identifying invariants of the transformation.

4. Transformation

Software architecture addresses more than the high-level system design. Software change resulting from maintenance requirements and integration problems is equally important. We focus on architecture transformations as a central software change technique. A number of reasons might require transformations:

- Interoperability can be a transformation objective.
- A reference architecture might need to be adopted.
- Changes in interface and interaction of services need to be addressed.

Architectures are often transformed if implementation restrictions have to be dealt with. An objective of architecture transformation is to implement changes, but also to preserve properties. Existing service connectivity and interaction is often worth being preserved, i.e. act as invariants of the transformation. Our patterns express processes at different levels of abstraction. Preserving patterns is desirable since patterns represent architectural configurations that are easy to understand and implement and describe structures that are often easy to maintain and reliable.

While the idea of preserving patterns at all architecture layers is therefore obvious, a verifiable transformation technique is needed. A generic constructive mapping rule is at the centre of our transformation technique. A notion of simulation captures the notions of equivalence and refinement of services and service processes.

A prerequisite for transformations is the explicit architecture specification of an 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 the three pattern-based architecture layers and the focus on patterns as transformation invariants. Each of the three architecture models might create its own requirements:

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

Property preservation is the goal of our architecture transformations. A simulation notion shall capture service process pattern preservation in the transformation technique. A *simu-*

lation definition, adopted from the π-calculus, satisfies the pattern preservation requirement for the processes that we envisage:

> Process Q **simulates** 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 definition expresses when process Q based on service expression n preserves, or simulates, the behaviour of process P based on service expression m. The services n and m can here be unrelated, as this definition is about observable behaviour only.

In order to automate transformation support based on this definition, a constructive theorem supporting simulation is needed. This theorem is the basis of a transformation rule which allows the verification of preservation and the automation of transformation. In [12], we have developed a constructive simulation test based on the construction of transition graphs for SAC process expressions.

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. Central to our transformation technique is a **transformation rule**, which associates patterns and simulation:

> Given an architecture specification S in SAC, 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 that A participates in, A' *simulates* P' with $P' = P[A/A']$, i.e. A' substitutes A and P is replaced by P' to cater for renaming of abstractions.

This produces pattern-preserving target architectures, if no further modification are made. We, however, argue that further modifications of the initial architecture in terms of additional or modified functionality are typical for transformations in evaluation and integration contexts. In this case, the invariant pattern preservation needs to be demonstrated. *Pattern-preserving transformation rules* can aid here. These are based on standard simulation relationships discussed in the process algebra literature [15], such as:

- $A; B$ simulates A: only transitions of B are added that do not affect A.
- **repeat**(A) simulates A: a single repetition corresponds to A.
- **choice**$(A; B)$ simulates A: the selection of A corresponds to A.
- **par**$(A; B)$ simulates A: A is always executed in the parallel composition.

From this constructive rule set, pattern-preserving transformations that even include structural and behavioural changes can be formulated.

The determination of an invariant, here the pattern P, is a common, but often non-trivial problem, which can be alleviated through domain-specific patterns.

4.2. Case Study – Pattern-Preserving Transformations

We demonstrate the adoption of the LTSA reference architecture 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. This interoperability objective, however, can have an impact on all levels of abstraction. Other learning technology standards, for instance, prescribe interfaces for learning technology objects, which would have to be reflected in service interfaces here.

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

$$\begin{aligned}\text{IDLE} = \textbf{par}(\ &\text{Learner}, \text{Delivery}, \text{StudentModel},\\ &\text{PedagogyModel}, \text{Workspace},\\ &\text{Evaluation}, \dots\ )\end{aligned}$$

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. Other existing IDLE components such as StudentModel and PedagogyModel have no direct counterpart in the LTSA, but can be abstracted by e.g. the Coach. Several different combinations of individual services can form patterns; these might actually overlap.

The first transformation step is to describe IDLE's architectural characteristics – ideally in LTSA terminology to simplify the transformation, see Fig. 3. The client-server-dispatcher pattern, see Fig. 2, is not identical to the structure that can be found in the IDLE system, see Fig. 4, since interactions with the resources server are added. 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 therefore *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 property is in our case easy to verify, since the IDLE specification in Fig. 4 describes only the service requests and connections that establish functionality defined in the pattern.

LTSA is a high-level system specification, to which we add functionality in IDLE in the form of new services not covered by LTSA. Architectural changes are necessary due to the workspace service integration into IDLE. The explicit storage and workspace service, see Fig. 5, requires the services LearnerEntity and Delivery to be modified in their interaction behaviour. 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.

Workflow patterns to be preserved can be identified due to their implementation as operators in the notation. The specification of the IDLE educational service system based on the client-dispatcher-server architectural design pattern in Fig. 4 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 reconstructed IDLE architecture is the transformation basis. The integration of specifications of the identified existing or created services forms the transformed architecture. The transformation task is to transform IDLE into LTSA-IDLE – an architectural variant of IDLE with LTSA-conform service interfaces and interaction processes. 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 its current structural, behavioural and quality properties based on existing documentation and system tests. The high-level architecture was given earlier and some detailed excerpts are presented in Figs. 4 and 5.
- **Invariant.** The invariant 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. The identification of patterns as invariants is a crucial and difficult step that depends on the expertise of the software architect – domain-specific patterns with common behaviour or qualities provide a starting point for invariant identification. The central pattern that we have identified and chosen to be an invariant captures the interactions between three of the central components of IDLE, i.e. learner, coach and delivery. It is one of the patterns that we found frequently in learning technology systems, and that we considered suitable to capture common interaction behaviour between central system components.
- **Target Construction.** The LTSA-based architecture specification of some IDLE services – which is the transformation result –

```
LearnerEntity = repeat (let requestEducServ = inv requestConnection()
                        in inv requestEducServ(resId);
                           let preferencesInfo = inv getPreferences();
                               learnResource = inv multimedia()
                           in inv setPreferences(alter(preferencesInfo)))
Delivery      =  inv registerEducServ(id);
                 repeat (rcv acceptConnection(c); rcv requestEducServ(s);
                         rep requestEducServ(run(s)); rcv locator(uri);
                         let learnResource = inv retrieveResource(uri)
                         in rep multimedia(learnResource))
Coach'        = choice (
                 choice (rcv registerEducServ(id), rcv unregisterEducServ(id)),
                 repeat (rcv requestConnection();
                   let c = getChannel()
                   in  inv acceptConnection(c); rep requestConnection(c);
                       repeat (
                         choice (
                           rcv getPreferences(); rep getPreferences(prefInfo),
                           rcv setPreferences(preferencesInfo),
                           rcv getLearnerInfo(id); rep getLearnerInfo(info),
                           let uri = inv locator(resource) in 0))))
LearningRes   =  rcv retrieveResource(uri); rep retrieveResource(retrieve(uri))
LearnerRec    =  rcv getLearnerInfo(id); rep getLearnerInfo(info(id))
```

Figure 6. *Transformation* – Resulting Adaptive Delivery in IDLE Architecture (selected components and services) based on the LTSA

can be found in Fig. 6. 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. This requires a reformulation of the IDLE process (parallel composition of composite processes, e.g. Delivery) as LTSA-IDLE by renaming abstractions and introducing Coach as a new element on the highest level.
- At the architectural design pattern level, the composition is changed at the subcomponent level. Coach is defined to reflect the merger of the two model components as a parallel composition of StudentModel and PedagogyModel.

– **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 terminology to LTSA. For instance, Learner becomes LearnerEntity. Renaming does not affect the simulation property. 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. In this case, Coach is only introduced as an abstraction for behaviour that already existed in the source system. Simulation is therefore also guaranteed. The new Coach' service handles the interaction with the learner and pedagogy model components. The original Coach specification from Fig. 4 has been extended to reflect this fact, which is presented in Fig. 6. 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 according 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[1].

[1] The formal proof is based on a constructive simulation test developed in [12], which is beyond the scope of this paper.

The specification in Fig. 6 describes a more complete range of interactions than the initial focus of Fig. 4 on educational service request and connection establishment. Fig. 6 adds 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. Adding functionality or significantly modifying the original architecture is common in evolution and integration situations. This is also the primary motivation for introducing invariants that are abstractions of the original architecture. The original architecture can due to these modifications in practice rarely be fully preserved – only well-chosen abstractions can be suitable invariants.

Verifying the preservation of the client-dispatcher-server invariant in the resulting architecture is a non-trival task. We can demonstrate for each affected service, i.e. LearnerEntity, Coach' and Delivery, that each simulates the original component:

- LearnerEntity simulates the Client through the first process elements (the repeat expression with the first two invocations) in the sequence of four subprocess expressions. In general, $\mathbf{repeat}(A; B)$ simulates $\mathbf{repeat}(A)$ because for each state transition in A there is a corresponding one in $A; B$.
- Delivery (which is unchanged) simulates the Server since, similar to the first case, only basic activities such as receive-reply interactions and invocations with the Resources component are added within the repeat loop.
- Coach' simulates the Dispatcher as the Dispatcher functionality becomes the outer process structure of the Coach to which preferences and learner initialisation and the location retrieval aspects are added. $A; B$ simulates A because transitions in A are also part of $A; B$.

Constructive rules are important in discharging the simulation proof obligation.

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*. 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. More involvement from the software architect is required if in the context of the transformation process, architectural features are also changed or extended. In this case, which is actually the standard situation in application integration and software migration, a fully automated approach is not feasible and the software architect needs to apply the provided constructive transformation rules to guarantee pattern preservation.

5. Related Work

Some ADLs are similar to SAC in terms of their focus on processes. Darwin [11] 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 flexibility and change demands such as both binding and unbinding on demand are required features. Wright [1] 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 [5], 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 transformation in the context of architecture evolution. 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. Architecture transformation also means controlled changes of architectural structures.

Patterns have recently been discussed in the context of Web service architectures [18, 19, 20, 4]. In [19, 20], collections of workflow patterns are compiled. We have based our catalog on these collections. The client-dispatcher-server pattern is also discussed in [18]. Other patterns that we have mentioned mainly originate from [7]. Grønmo et al. [16] consider the modelling and building of compositions from existing Web services using model-driven development. The authors consider two modelling aspects, service (interface and operations) and workflow models (control and data flow concerns). These efforts embed patterns into a methodological framework, similar to our objectives. Our consideration of distribution as a further dimension in service patterns, however, goes beyond those approaches.

A recent software architecture approach for service-based systems is model-driven development (MDD). MDD emphasises the importance of modelling and transformations. The latter are, in contrast to our framework, part of the modelling process between modelling levels of abstraction. Our framework addresses the transformation of architecture specifications, for instance to support software change and evolution. While MDD is vertically oriented, i.e. mapping from abstract domain models to more concrete platform models, we follow a more horizontal transformation approach on the level of architectures. We have focused on hierarchical pattern-based process modelling and architectural configuration – two aspects that can complement and extend MDD by providing higher levels of abstraction and architectural transformation. The formality of our approach satisfies the automation requirements of model-driven development and even adds reasoning support.

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 Web Services, an engineering methodology and techniques are still largely missing. We have presented a layered architecture model that captures behavioural aspects and associates quality of architectural structures at different levels of abstraction through patterns. A modelling notation allows interaction behaviour in architectures and architectural configurations to be captured and distribution and quality characteristics to be associated. Interaction behaviour and composite processes are essential aspects for the development and maintenance of distributed service-based systems.

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 hierarchically organised patterns, supported by the architecture model and the transformation technique, can play for service-oriented architecture. Patterns that capture interaction behaviour between services are ideally suited for the service context with its focus on processes. Process patterns provide an abstraction mechanism that captures relevant invariants for architectural transformation.

- Patterns as abstractions greatly improve the possibility to reuse and evolve architectural designs. As architectural abstractions, they capture important behaviour and quality invariants.
- Pattern-based modelling has implications for functional and quality characteristics of a service-centric software system. Pattern-based transformation focuses on functional properties, but also preserves the quality characteristics.

The novelty of our architecture transformation technique is to use patterns to capture behaviour and quality invariants in a layered architectural modelling approach to service-based architecture evolution and change.

We have applied the presented techniques in the ongoing design, maintenance and evolution of the IDLE system. It is an extensive system with a range of interactive, distributed features, characterised by complex a information archi-

tecture, that has been developed by more than 20 people and maintained for more than ten years – which indicates the scalability of the transformation technique. The technique was described in its principles and illustrated using the case study. Our tool implementation for distribution pattern architecture demonstrates the positive effect of pattern-based transformations on architectures in terms of quality. However, the pragmatics of modelling with formal notations need to be addressed further. While in the case study, architects were familiar with the notation, a closer integration with UML activity diagrams is envisaged to improve acceptance and usability.

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 – particularly obvious is migration and legacy integration projects. 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 here. Re-engineering and migration approaches for the architectural level can provide further solutions here.

References

[1] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(3):249, 1997.

[2] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web services: concepts, architectures and applications*. Springer Verlag, 2004.

[3] R. Barrett, L. M. Patcas, C. Pahl, and J. Murphy. Model driven distribution pattern design for dynamic web service compositions. In *Proceedings of the 6th international conference on Web engineering*, page 136, 2006.

[4] F. Buschmann, K. Henney, and D. C. Schmidt. *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. Wiley, May 2007.

[5] C. Canal, E. Pimentel, and J. M. Troya. Compatibility and inheritance in software architectures. *Science of Computer Programming*, 41(2):105–138, 2001.

[6] R. Dijkman and M. Dumas. Service-oriented design: A multi-viewpoint approach. *International journal of cooperative information systems*, 13(4):337–368, 2004.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-wesley Reading, MA, 1995.

[8] D. Garlan and B. Schmerl. Architecture-driven modelling and analysis. In T. Cant, editor, *Proceedings of the eleventh Australian workshop on Safety critical systems and software*, page 17, 2007.

[9] IEEE P1484.1/D8. Draft standard for learning technology – learning technology systems architecture LTSA, 2001.

[10] R. Kazman, S. J. Carrière, and S. G. Woods. Toward a discipline of scenario-based architectural engineering. *Annals of software engineering*, 9(1–4):5–33, 2000.

[11] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. *Software Engineering—ESEC'95*, pages 137–153.

[12] C. Pahl. An ontology for software component matching. *Fundamental Approaches to Software Engineering*, pages 6–21.

[13] C. Pahl. A Pi-calculus based framework for the composition and replacement of components. In *SAVCBS 2001 Proceedings*, page 97, 2001.

[14] C. Pahl, R. Barrett, and C. Kenny. Supporting active database learning and training through interactive multimedia. *ACM SIGCSE Bulletin*, 36(3):31, 2004.

[15] D. Sangiorgi and D. Walker. *The pi-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2003.

[16] D. Skogan, R. Grønmo, and I. Solheim. Web service composition in UML. In *Eighth IEEE International Enterprise Distributed Object Computing Conference, 2004. EDOC 2004. Proceedings*, pages 47–57, 2004.

[17] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. Software architecture: Foundations, theory, and practice. 2009.

[18] N. Y. Topaloglu and R. Capilla. Modeling the variability of web services from a pattern point of view. *Web Services*, pages 128–138.

[19] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, July 2003.

[20] M. Vasko and S. Dustdar. An analysis of web services workflow patterns in collaxa. *Web Services*, pages 1–14.