Architecting as Decision Making with Patterns and Primitives

Uwe Zdun
Information Systems Institute
Vienna University of Technology
Vienna, Austria
zdun@infosys.tuwien.ac.at

Paris Avgeriou
Department of Computer Science
University of Groningen
The Netherlands
paris@cs.rug.nl

Carsten Hentrich
Enterprise Content Management Solution
CSC Deutschland Solutions GmbH, Germany
chentrich@csc.com

Schahram Dustdar
Information Systems Institute
Vienna University of Technology
Vienna, Austria
dustdar@infosys.tuwien.ac.at

ABSTRACT

The application of patterns is used as a foundation for many central design decisions in software architecture, but because of the informal nature of patterns, these design decisions are usually not precisely documented in the models. In our earlier work, we had proposed pattern primitives as a solution to precisely model the patterns in the corresponding architectural views. Building upon that approach, this paper introduces a pattern-based architecting process that aims at inexpensively documenting design decisions in the architectural views alongside the natural flow of design. The decisions that are made explicit, concern the selection of patterns, their variants and the corresponding primitives, as well as the resolution of inconsistencies between the architectural views. The approach is demonstrated in the domain of process-driven SOA for two architectural views: Component-and-Connector and Process Flow.

1. INTRODUCTION

Architectural Knowledge (AK) tends to evaporate as software systems evolve, with grave consequences for software development projects [16]. The effective and systematic documentation and subsequent sharing of architecture knowledge is imperative for large and/or distributed projects [2]. However, in practice, architectural knowledge documentation is conceived as a resource-intensive process without tangible (short-term) gains for the documenters themselves and is quite often skipped or performed inadequately [2].

One partial solution to this problem is to document the design decisions when applying patterns in the architecture [10]. Architectural patterns capture reusable architectural knowledge in the form of well-proven solutions to recurring software design problems arising in particular contexts and domains [17]. Patterns concern some of the most important architectural decisions, are easy to use, and provide a rich set of information about rationale, consequences, and related decisions; knowledge that must be documented. Documenting AK based on patterns fits within the "natural" course of architecting, without introducing extra efforts for the software architects and developers. Patterns are thus an inexpensive way of documenting some of the most significant AK entities.

Documenting the AK associated to the application of patterns is therefore of paramount importance, but there are two problems. Firstly, architectural patterns cannot be precisely specified or modeled, e.g., by using a parameterizable, template-style description, because they are described in an informal way [21]. This stems from the notion of a pattern per se: it does not describe a single solution but a solution space, variable enough to be applied in diverse situations. In our previous work [21] we have proposed a solution for the systematic modeling of patterns by introducing an intermediate abstraction, the *pattern primitives*: fundamental, precisely-specified modeling elements in representing a pattern. Pattern primitives enable us to document the usage of patterns inside the corresponding architectural views and, at the same time, document some of the most important architectural decisions with minimal effort.

Our original set of pattern primitives presented in [21] was targeted at modeling architectural patterns, using basic architectural abstractions like components, connectors, ports, and interfaces. In [21] we presented 9 architectural primitives, which can be used to model several of the most common architectural patterns (such as those found in [4]), in the *Component-and-Connector* architectural view. Later we extended this approach to the domain of process-driven, service-oriented architectures (SOA) [23]. A process-driven SOA provides a process engine (or workflow engine) at its top-level layer [24], and the services realize individual activities in the process. In [23] we introduced 13 pattern primitives, which can be used to model a pattern language for process-driven SOA, described in [12]. We use the primitives in the *Process Flow* architecture view, which mainly models the flow within a process-driven SOA.

The second problem in documenting pattern-based design decisions concerns the inconsistencies that occur in different architectural views. In current architecture documentation practice, the application of a pattern is usually modeled in a single architectural view [6]. Therefore different patterns and their consequent deci-

SHARK'08, May 13, 2008, Leipzig, Germany.
Copyright 2008 ACM 978-1-60558-038-8/08/05 ...$5.00.

sions are modeled independently in different views. This may potentially cause inconsistencies between the views, as the different patterns sometimes entail conflicting structural or behavioral semantics. The problem of inconsistencies across architectural views is of course not specific to applying patterns but a general problem in software architecture documentation [15], but so far this problem has not been addressed in the context of pattern-based architecting.

In this paper we extend our existing approach by proposing an *architecting process based on patterns and their modeling through primitives*, that supports the explicit documentation of design decisions. The process of architecting can be seen as a set of architecture design decisions [3], and our approach focuses on making explicit the following decisions: selecting patterns and their variants to solve problems, and modeling them through primitives. If, in the course of this architecting process, inconsistencies between views occur, we propose to remedy this problem by combining the pattern primitives in the different views in order to bridge the semantic gaps between the corresponding models.

The advantage of this approach is that documenting the aforementioned design decisions is performed at the same time as designing without requiring extra effort; the results of making these decisions are explicit in the architecture models. We demonstrate the applicability of our approach in the application domain of *process-driven SOA*. We have selected this domain as we already have two sets of pattern primitives that are suitable for two architectural views in this domain: the Process Flow and Component-and-Connector views.

This paper is structured as follows: First, the process-driven SOA domain will be presented in Section 2 through a domain-specific pattern language. Next, we describe in Section 3 the primitives for the Process Flow and Component-and-Connector views, as the background of this work. Section 4 presents the proposed architecting process and demonstrates it through a case study in the process-driven SOA domain. To aid the architects and developers, we have implemented a model-driven tool chain to validate the primitive models automatically, and to generate code from the models. Section 5 briefly reports on this tool chain. In the final two sections, we compare this approach to related work and share some final conclusions.

2. PATTERN LANGUAGE FOR PROCESS-DRIVEN SOA

Software patterns and pattern languages provide systematic reuse strategies for design knowledge [17]. A pattern encodes proven practices for particular, recurring design problems. A pattern language is a collection of patterns that solve the prevalent problems in a particular domain and context, and specifically focuses on the pattern relationships in this domain and context.

In this section, we give an overview of the subset of our pattern language for process-oriented integration of services that is needed for this paper (for details please refer to [12]). In the pattern language, the pattern[1] MACRO-MICROFLOW sets the scene and lays out the conceptual basis to the overall architecture. The pattern divides the flow models into so-called *macroflows*, which describe the long-running business processes, and *microflows*, which describe the short-running technical processes. The PROCESS-BASED INTEGRATION ARCHITECTURE pattern describes how to design an architecture based on a number of layers to compose services in a flexible and scalable manner, following the MACRO-MICROFLOW conceptual pattern.

[1]Note that we print the names of the patterns in capital letters to differentiate them from components, layers, or other concepts.

Figure 1 shows an exemplary (large-scale) configuration of a PROCESS-BASED INTEGRATION ARCHITECTURE with five layers. Let us illustrate the pattern language using this example. Multiple MACROFLOW ENGINES execute the macroflows, i.e., the MACROFLOW ENGINES allow for flexible orchestration of business processes. A typical example of a MACROFLOW ENGINE is a BPEL process engine. PROCESS INTEGRATION ADAPTERS integrate the specific interface and technology of the process engine into a system. The macroflow integration layer is used to connect the macroflow engines to the PROCESS-BASED INTEGRATION ARCHITECTURE via service invocations of corresponding PROCESS INTEGRATION ADAPTERS.

A dispatching layer enables scalability by dispatching to a number of MICROFLOW ENGINES. In this layer, a RULE-BASED DISPATCHER dynamically decides based on (business) rules, where and when a (macroflow) activity is executed. A MICROFLOW ENGINE allows for orchestrating service invocation through short-running technical flows. An example of microflow technology are message brokers.

In the example, services are executed that contain business application adapters that connect to backends. These business application adapters, just like the PROCESS INTEGRATION ADAPTERS, follow the CONFIGURABLE ADAPTER pattern. A CONFIGURABLE ADAPTER connects to another system in a way that allows to easily maintain the connections, considering that interfaces may change over time. They are typically managed in CONFIGURABLE ADAPTER REPOSITORIES.

There are several architectural views that can address stakeholders concerns specific to the domain of process-driven SOAs. Some commonly used views are:

- Business process flow, a sub-view of the Process Flow view (showing business processes in macroflows),
- Message flow, a sub-view of the Process Flow view (showing technical processes in microflows),
- Business objects (showing business objects/artifacts),
- Business resources,
- Components and connectors (showing run-time entities and their relations),

3. PATTERN PRIMITIVES

In this section, we briefly present our earlier work on pattern primitives for modeling patterns in the Process Flow and the Component-and-Connector views, hence making pattern-based design decisions an explicit part of the architectural documentation. Pattern primitives are modeling elements with precisely-described semantics, that are primitive in the sense that they represent basic units of abstraction in the domain of the pattern. We model pattern primitives using UML 2.0 extensions because the UML has become the "lingua franca" of software design and is vastly supported by tools. We specify an extension of a UML 2.0 metaclass for each elicited primitive, using the standard UML extension mechanisms: stereotypes, tag definitions, and constraints. We use the Object Constraint Language (OCL) to specify the constraints and provide precise semantics to the primitives.

3.1 Process Flow Primitives

In our earlier work [23], we presented 13 pattern primitives that we have mined from the pattern language presented in Section 2. These pattern primitives mainly deal with the Process Flow

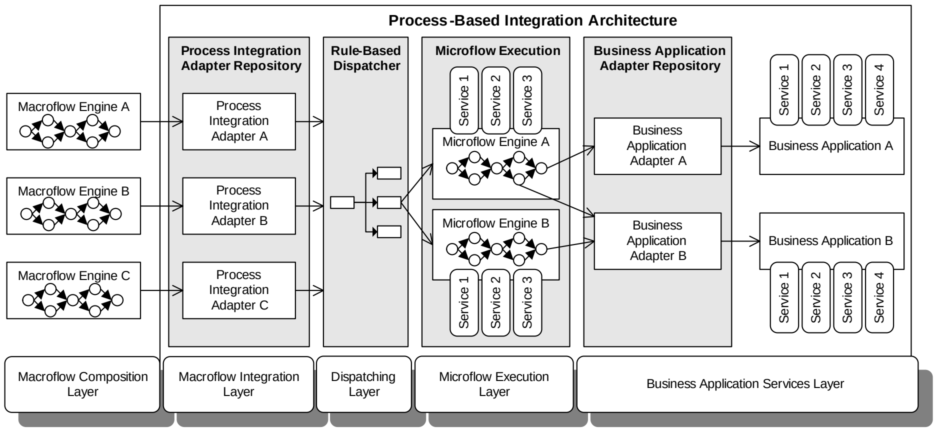

Process-Based Integration Architecture
Process Integration Adapter Repository
Rule-Based Dispatcher
Microflow Execution
Business Application Adapter Repository
Macroflow Engine A
Macroflow Engine B
Macroflow Engine C
Process Integration Adapter A
Process Integration Adapter B
Process Integration Adapter C
Microflow Engine A
Microflow Engine B
Service 1
Service 2
Service 3
Service 4
Business Application Adapter A
Business Application Adapter B
Business Application A
Business Application B
Macroflow Composition Layer
Macroflow Integration Layer
Dispatching Layer
Microflow Execution Layer
Business Application Services Layer

Figure 1: Example Configuration of a Process-based Integration Architecture

view, and hence the primitives extend UML's activity diagram meta-model, which describes how to depict flow abstractions in UML. There are flow models for macroflows, micoflows, and their activity steps. Even though these flow models have highly different semantic properties, they share the same basic flow abstractions – and glue the other models used in a process-driven SOA.

To illustrate our concepts, we will concentrate on one example primitive, the Macro-Microflow Refinement Primitive, and then present the other primitives as thumbnails. The Macro-Microflow Refinement Primitive models the refinement of Macroflows into Microflows. To model this primitive, we introduce UML2 stereotypes to distinguish the different kinds of refinement activities in the UML2 models. In particular, we introduce a `Macroflow` stereotype and a `Microflow` stereotype. These stereotypes have a common superclass extending UML's `Activity` metaclass. On this superclass a tag value `refinedNodes` is introduced to denote the refinement relationships between Activities[2].

We can then model the Macro-Microflow Refinement primitive by constraining the stereotypes. In particular, if the `refinedNodes` tag value of a `Microflow` is not empty, the `Microflow` is a refinement of another `Microflow` or a `Macroflow`. If `refinedNodes` of a `Macroflow` is not empty, it must be refined by another `Macroflow`. This can be precisely modeled using OCL constraints (see [23]) for details).

Each primitive describes a precise, parameterizable building block that can be used in the solution of the patterns. Our model validator can check all these constraints automatically, and indicate whether any constraints of the primitives have been violated.

Let us illustrate the use of the primitive in a specific example of modeling the application of the MACRO-MICROFLOW pattern. Specifically we want to model how the pattern structures a process model into macroflows and microflows. In the MACRO-MICROFLOW pattern at least one Macroflow Model and one Microflow Model with a refinement relationship between them must be present in a model. There are different specific kinds of refinement possible, specified by the primitive constraints, such as: macroflows can be refined by other macroflows, microflows can be refined by other microflows, macroflows can be refined by microflows, etc. The Macro-Microflow Refinement primitive allows us to model a number of pattern variants of the MACRO-MICROFLOW pattern. For instance, the MACRO-MICROFLOW structure may follow a refinement through a number of macroflow models (high-level to low-level) plus a low-level microflow model depicting the technical message flows. Figure 2 shows one example, but many other variants of the patterns can be modeled using the same primitive (see [23] for details).

[2]There are other kinds of flows than macroflows and microflows derived from this class (see [23]), but as they are not used in this paper, we omit them here to simplify the discussion.

The detailed specification of this and the other 12 primitives can be found in [23]. The thumbnails of these primitives are presented here as an overview:

- *Process Flow Refinement*: A macroflow or microflow is refined using another process flow.
- *Process Flow Steps*: A macroflow or microflow is refined by a number of sequential steps.
- *Macroflow Model*: A macroflow can be refined by other macroflows or macroflow steps.
- *Microflow Model*: A microflow can be refined by other microflows or microflow steps.
- *Automated Macroflow Steps*: Macroflow Steps are designed to run automatically by restricting them to three kinds: process function invocation, process control data access, and process resource access.
- *Automated Microflow Steps*: Microflow Steps are designed to run automatically by restricting them to two kinds: process function invocations and process control data access.
- *Synchronous Service Invocation*: A service is invoked synchronously.
- *Asynchronous Service Invocation* A service is invoked asynchronously.
- *Fire and Forget Invocation*: A service is invoked asynchronously with fire and forget semantics, i.e. no result or acknowledgment is written for this service.
- *One Reply Asynchronous Invocation*: A service is invoked asynchronously, and exactly one result is coming back.
- *Multiple Reply Asynchronous Invocation*: A service is invoked asynchronously, and multiple results are coming back.
- *Process Control Data Driven Invocation*: A service is invoked using only data from the process control data.

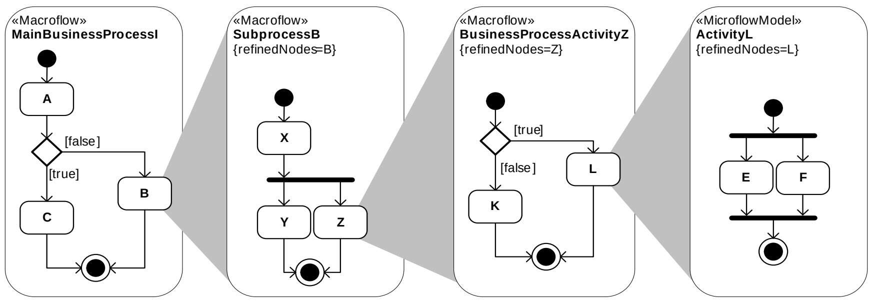


Figure 2: Macro-Microflow modeling example

3.2 Component and Connector Primitives

Similarly to primitives for process flows, we have proposed a set of primitives for modeling common architecture patterns in the Component-and-Connector architectural view [21]. These primitives extend basic architectural abstractions, such as components, connectors, ports, and interfaces. Hence, the composite structures and components meta-models of UML are extended here. We can use these primitives to model the structural aspects of the patterns in a process-driven SOA. Let us consider the PROCESS-BASED INTEGRATION ARCHITECTURE pattern, introduced earlier, and the Callback primitive as an example.

The Callback primitive [21] can be used to model asynchronous interaction: A callback denotes an invocation to a component B that is stored as an invocation reference in a component A. The callback invocation is executed later, upon a specified set of runtime events. Between two components A and B, a set of callbacks can be defined. To capture the semantics of callbacks properly in UML, we proposed five stereotypes to denote interfaces of events (`IEvent`) and callbacks (`ICallback`), ports of events (`EventPort`) and callbacks (`CallbackPort`), and a `Callback` stereotype that extends the `Connector` metaclass and specifies the semantics of a callback connector using the other 4 stereotype specifications. Again, we have precisely specified the constraints using OCL.

In the PROCESS-BASED INTEGRATION ARCHITECTURE pattern, different kinds of components are connected. Figure 1 shows an exemplary larger configuration, in which multiple macro-/microflow engines and a dispatcher are used. This pattern mandates the flexible assembly of the different components by following asynchronous messaging patterns from [13]. The use of the callback primitive is therefore the appropriate modeling abstraction to capture the asynchronous interaction within the PROCESS-BASED INTEGRATION ARCHITECTURE pattern. The event ports of each layer are listening to events from the higher-level layer, and when an event arrives, the components call into the lower-level layer. Once a result is received, it is propagated back into the higher-level layer using a Callback. Figure 3 shows an example UML2 model that applies the Callback primitive to the configuration from Figure 1 (components in higher-level layers are depicted on the left hand side of the callback connectors).

The other 8 primitives besides Callback introduced in [21] are:

- *Indirection*: One or more related "proxy" components receive a message on behalf of one or more "target" components and forward the message to these "targets".

- *Grouping*: A number of components belong semantically together, but the whole is made up only from the parts, and there is no notion of a component that explicitly represents the whole.

- *Layering*: Layered structures are ubiquitous in software architectures, where groups of components are ordered and invocations between the different groups need to respect certain rules.

- *Aggregation Cascade*: A part-whole hierarchy where the composite objects have (recursive) constraints of the form: "A composite A can only aggregate components of type B, B only C, etc".

- *Composition Casade*: Like Aggregation Cascade, but it further enforces that a component may not be part of more than one composite at any time.

- *Shield*: One or more components act as 'shields' for a set of components that form a subsystem, and no external client should be allowed to access members of the subsystem directly.

- *Typing*: Introduces the notions of a supertype connector and a type connector, which can be used to define custom typing models using associations.

- *Virtual Connector*: Explicitly models communication among components that have no direct relationship, but still communicate virtually using other components and connectors in between.

4. THE PROPOSED APPROACH

4.1 An Architecting Process

The general process of architecting as decision making, based on patterns and primitives, is shown in Figure 4. Design decisions are made in the course of architecting, when selecting patterns, pattern variants and primitives in four successive phases:

1. Phase 1: At first patterns from specific pattern languages are selected that are a good match to our problem space. Each pattern solves a design problem and might lead to a another problem that is solved by another pattern (in the same pattern language).

2. Phase 2: A pattern represents not a single solution but an entire solution space. Hence, the pattern's solution must be applied in the context of the design problem at hand. That is, a pattern variant must be selected to tailor the generic instructions described in the pattern description to the concrete design situation.

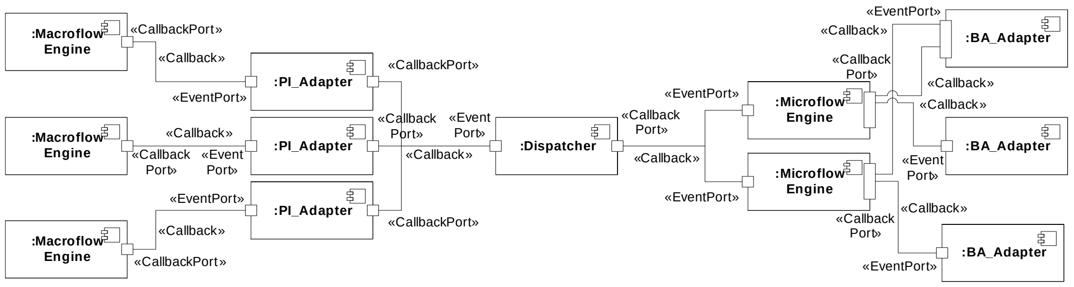


Figure 3: UML2 Model for the Example Configuration

3. Phase 3: Each pattern (or more precisely: each pattern variant) is modeled in an architectural view independently of the other patterns, by selecting the appropriate primitives. Usually, this should be possible with the existing primitives catalog. In seldom cases, mining and documenting new primitives might be required.

4. Phase 4: If there are inconsistencies between views, the approach varies depending on whether they can be resolved with the existing primitives. If this is the case, then additional primitives from the existing sets of primitives are selected to provide for the missing semantics. Also, additional constraints may need to be applied in some of the primitives, in order to align their semantics to those of the primitives used in the other views. Otherwise additional primitives need to be introduced to resolve the remaining inconsistencies. The new primitives are not limited to a single view but cut across different views: their constraints refer to model elements in at least two architectural views, in order to simultaneously satisfy the semantics of the different patterns.

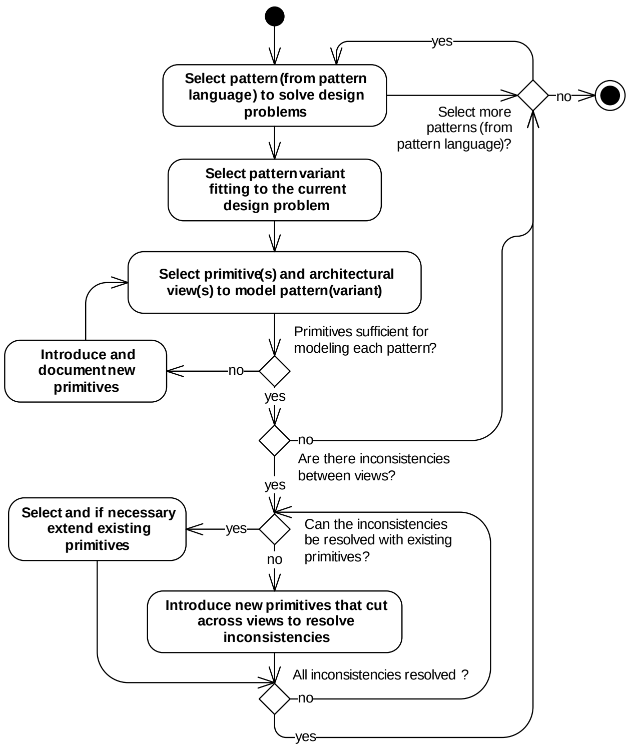
The result of this process is a number of architectural models in views that explicitly contain patterns through their primitives. All decisions about the selection of the patterns, their variants and the primitives are visible and formally defined in those models.

4.2 Case study

In the remainder of this section, we illustrate our architecting process using an extension of the example from the previous sections, in the process-driven SOA domain. Even though we apply the solution only for the process flow and the component & connector views, the general approach can be used for other architectural views and domains as well.

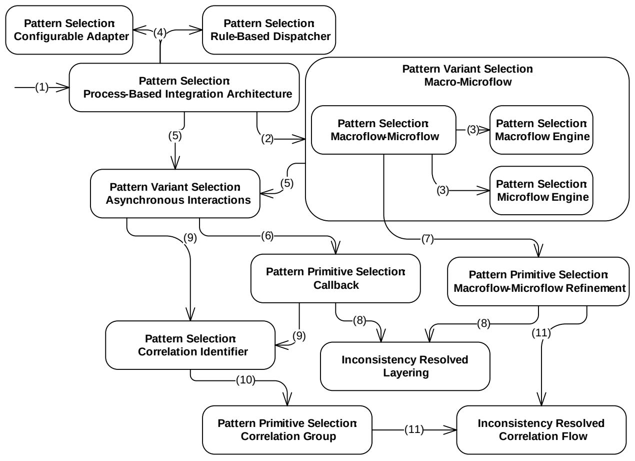
As illustrated in Figure 5, the first set of design decisions we make concerns the selection of a number of patterns. First, we select the PROCESS-BASED INTEGRATION ARCHITECTURE pattern (DD1), and specifically a variant of that pattern following the MACRO-MICROFLOW pattern (DD2). The latter is in turn realized using other patterns: explicit MACROFLOW ENGINES and MICROFLOW ENGINES (DD3), as in the example configuration in Figure 1. Furthermore we decide to use CONFIGURABLE ADAPTERS as well as a RULE-BASED DISPATCHER (DD4) as part of the PROCESS-BASED INTEGRATION ARCHITECTURE. To complete the realization of the PROCESS-BASED INTEGRATION ARCHITECTURE we also select its variant, where the interactions between the components are asynchronous (DD5).

Next we select primitives to model the selected patterns. In order to model the MACRO-MICROFLOW pattern in the process flow view, we select the Macro-Microflow Refinement primitive (DD7), as illustrated in Figure 2. Furthermore, to model the PROCESS-BASED



Select pattern (from pattern language) to solve design problems
yes
no
Select more patterns (from pattern language)?
Select pattern variant fitting to the current design problem
Select primitive(s) and architectural view(s) to model pattern(variant)
Primitives sufficient for modeling each pattern?
Introduce and document new primitives
no
yes
no
Are there inconsistencies between views?
yes
Select and if necessary extend existing primitives
yes
Can the inconsistencies be resolved with existing primitives?
no
Introduce new primitives that cut across views to resolve inconsistencies
All inconsistencies resolved ?
no
yes

Figure 4: Overview of the Architecting Process



Pattern Selection Configurable Adapter
(4)
Pattern Selection Rule-Based Dispatcher
(1)
Pattern Selection Process-Based Integration Architecture
Pattern Variant Selection Macro-Microflow
(5)
(2)
Pattern Selection Macroflow-Microflow
(3)
Pattern Selection Macroflow Engine
(3)
Pattern Selection Microflow Engine
Pattern Variant Selection Asynchronous Interactions
(5)
(9)
(6)
(7)
Pattern Primitive Selection Callback
Pattern Primitive Selection Macroflow-Microflow Refinement
Pattern Selection Correlation Identifier
(9)
(8)
(8)
(11)
Inconsistency Resolved Layering
(10)
Pattern Primitive Selection Correlation Group
(11)
Inconsistency Resolved Correlation Flow

Figure 5: Sequence of Design Decisions in the case study

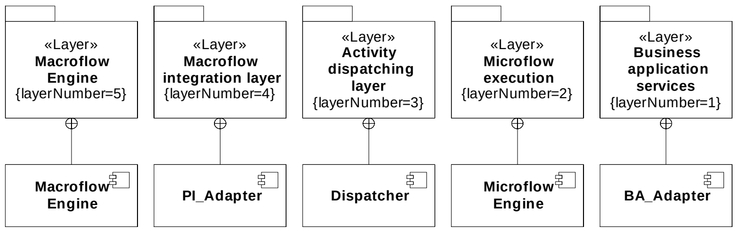


Figure 6: Extending the Example Configuration with Layering

INTEGRATION ARCHITECTURE with asynchronous interactions in the component & connector view, we select the Callback primitive (DD6), as illustrated in Figure 3. There are more primitives that need to be selected in order to model all the selected patterns, but we omit them here to keep the size and scope relatively small.

After using the primitives to model all the patterns, we examine the two views searching for potential inconsistencies. Indeed we discover one inconsistency between them: The use of MACRO-MICROFLOW inside the PROCESS-BASED INTEGRATION ARCHITECTURE introduces certain constraints, e.g. components that represent the microflow should not invoke macroflow functionality, macroflow adapters should not be used at the microflow level and vice versa, the dispatcher should only invoke short running microflows, etc. These constraints have not yet been modeled in the component & connector view.

We first try to resolve this inconsistency in the two views by looking for existing primitives to express the semantics not modeled so far in the component & connector view. Indeed, we can use an architectural primitive from [21] in the component & connector view to express the missing constraints: Layering. Layering describes groups of components and entails that group members from layer X may call into layer $X - 1$ and components outside the layers, but not into layer $X - 2$ and below. Figure 6 shows a UML2 model that extends the model from Figure 3 using the Layering primitive. That is, the `Layer` stereotype which is an extension of the Package metaclass is used here. This decision (DD8) works only in the component and connector view, but it does achieve to solve the inconsistency between the two views.

There are no more inconsistencies so far, so we go back to the start of our process, and check whether more patterns need to be selected in order to solve our design problems. Indeed, the components in the PROCESS-BASED INTEGRATION ARCHITECTURE communicate asynchronously via service invocations, but it has not yet been decided how to correlate the messages in the asynchronous interactions. This leads to selecting another pattern: To correlate the events and callbacks between components that interact using asynchronous interactions, CORRELATION IDENTIFIERS [13] (DD9) are passed between the components.

Next, we must find primitives to model CORRELATION IDENTIFIER in the component & connector view, showing which components need to use it. This is essential because in a PROCESS-BASED INTEGRATION ARCHITECTURE that uses multiple different components, as in the selected variant of the pattern, multiple CORRELATION IDENTIFIERS can be used[3]. There are no existing primitives that can help us to model this pattern, so we need to introduce a new primitive: the Correlation architectural primitive (DD10). The Correlation primitive introduces two stereotypes `CorrelationIdentifier`, extending the 'Class' metaclass, and `CorrelationGroup`, extending the 'Package' metaclass. Finally the following constraints are defined (in OCL): A correlation group package, used in the tag value `correlationGroup`, must be stereotyped as `CorrelationGroup`. Each correlation group must have a `CorrelationIdentifier` with a `correlationGroup` tag value that points to it.

Using this primitive we can model CORRELATION IDENTIFIER by putting together certain components into a correlation group and use a specific class (or component) as correlation identifier. For instance, the component model in Figure 7 expresses that the components MacroflowEngine, PI_Adapter, and Dispatcher form a correlation group using the correlation identifier type C_ID.

Modeling CORRELATION IDENTIFIERS solely in the components & connectors view of the PROCESS-BASED INTEGRATION ARCHITECTURE creates one more inconsistency with the Process Flow view. In particular, the macroflow and microflow models should pass a valid CORRELATION IDENTIFIER type to all asynchronous invocations. However, there are problems when trying to check this constraint in the process flow. Consider as an example the process flow model shown in the lower part of Figure 7, where `C_ID` is used in a process flow of the MACROFLOW ENGINE that is part of the correlation group, which is valid. Nevertheless, there is no way of checking whether this type of correlation identifier is used in activities of process engines that do not belong to the correlation group.

Existing primitives are not suficcient to resolve this inconsistency, hence we introduce a new primitive: Correlation Flow (DD11) for modeling the patterns MACROFLOW ENGINE and MICROFLOW ENGINE, which are part of the PROCESS-BASED INTEGRATION ARCHITECTURE. The goal is to be able to model which macroflow and microflow models are executed on which MACROFLOW ENGINE and MICROFLOW ENGINE respectively. The primitive defines three new stereotypes: `ProcessEngine`, extending the 'Component' metaclass, as well as `MacroflowEngine` and `MicroflowEngine`, two subclasses of `ProcessEngine`. Furthermore we define constraints (in OCL) that specify that a `MacroflowEngine` can only have `activities` that are stereotyped either as Macroflow or MacroflowSteps. A `MicroflowEngine` can only have `activities` that are stereotyped either as Microflow or MicroflowSteps. These constraints define precise relations between the component and flow models. Finally, we need to specify a concern regarding the correlation: If a particular process engine is part of a correlation group, then all `AsyncServiceInvocation` activity nodes of activities that belong to this engine must have an incoming Object Node that is of the `CorrelationIdentifier` type of the engine's correlation group.

5. TOOL SUPPORT

To validate our concepts, we have developed a model-driven tool

[3]Note that there are many other issues that need to be modeled in order to deal with correlations, but as an illustrative example of patterns that affect two views, we only want to focus on this aspect of correlation here.

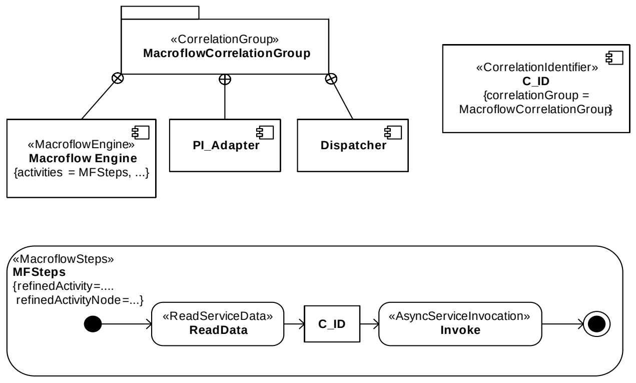


Figure 7: Correlation Identifier Group

chain (see [23, 22, 18] for details), which supports modeling and model validation for the concepts presented in this paper. In our tool chain, we mainly use UML2 models that are extended with UML2 profiles for modeling the pattern primitives as inputs. These UML2 models can either be developed with UML tools (with XMI export) or in domain-specific languages (DSL). If a UML tool is used, the XMI export is transformed into the textual DSL syntax. Internally all inputs are transformed into the same DSL syntax.

In the tool chain, a model validator gets all input models and validates the conformance of the application models to the meta-models. It also checks all OCL constraints, specified in the pattern primitive definitions. After the model is validated it is transformed into an EMF (Eclipse Modeling Framework) model, which is understood by the code generator. We then generate code in executable languages, such as Java and BPEL, using the code generator.

6. RELATED WORK

Zimmermann et al. present a generic modeling approach for SOA projects [25]. Similarly to our approach, they base their work on project experiences and distill proven practices from practical cases. They also integrate multiple architectural views for a SOA: object-oriented analysis and design, enterprise architecture, and business processes. Even though there are many similarities to our approach, there is the difference that the authors use rather informal models to illustrate their approach and do not provide a concept for explaining the conceptual building blocks of the architecture (like the patterns in our approach).

Some approaches use the Semantic Web to model the integration of (Web) services and process. OWL-S [7] includes a process model for Web services and it uses semantic annotations to enable dynamic composition. Cardoso and Sheth [5] use Semantic Web ontologies to facilitate the interoperability of heterogeneous Web services in the context of workflows. This is a different – though not contradictory – approach to model the integration of services and processes than the process flow paradigm used in our approach. Architectural abstractions are not integrated in these approaches.

Business process management tools, such as Adonis [1] or Aris [14], describe a holistic model of business process management, ranging from strategic decisions to the design of business processes. They also provide mappings (imports/exports) to the realization and execution of processes. They are integrated with standard model types and extensible with new model types. Such tools are related to our approach at a high level because they represent important prior art in the field of model integration. But they do not – such as our approach – concentrate on models based on proven practices. They also do not specifically focus on the field of process-driven SOAs; they are more focused on the business processes. However, an extensible tool suite like Adonis can be used for providing input models for our approach.

In Chapter 13 of the book Software Factories [9] it is briefly discussed how typical MDSD concepts can be used to support SOA modeling, but only with a focus on Web Services technology. Essentially, the process description, e.g. in BPEL, is seen as a platform for implementing abstractions in a product line, and the services are seen as product line assets for systematic reuse. It is advised that patterns are used as proven practices, but there is no guidance how to map them to formal modeling constructs, like the pattern primitives in our approach.

There are a number of UML profiles for various SOA aspects. Wada et al. [20] propose an UML profile to model non-functional aspects of SOAs and present an MDSD tool for generating skeleton code from these models. Heckel et al. [11] propose a UML profile for dynamic service discovery in a SOA by providing stereotypes that specify the relationships among service implementations, service interfaces, and requirements. Gardner et al. [8] define a UML profile to specify service orchestration with the goal to map the specification to BPEL code. Vokac and Glattetre [19] use – as in our examples – UML profiles to define DSLs. The proposed UML profile supports data integration issues. In contrast to our approach, these approaches focus on a single type of model, not on model integration or on extensibility with other model types. The modeling constructs are not systematically derived from proven practices. Hence, the approaches are very specific for the application area they focus on.

7. CONCLUSION

Modeling architectural patterns is an inexpensive and non-intrusive way to model major design decisions in the architecture. This paper presented an architecting process that extends our pattern primitives approach for modeling patterns, by focusing on the straightforward documentation of the corresponding design decisions. The decisions that concern the selection of patterns and their variants, as well as the selection of the primitives to model them are explicitly specified in the architecture models. Furthermore inconsistencies between multiple architectural views can be resolved, ensuring the consistent documentation of the pattern-based design decisions. We demonstrated the approach for two architectural views, Component-and-Connector and Process Flow, in the context of the process-driven SOA domain, but our approach is not limited to these architectural views or this domain. To validate our approach, we have implemented a model-driven software development tool chain, in which the primitive constraints get automatically validated by a model validator and the consistency to code is ensured.

As a pure extension to our previous work on primitives, this approach just requires architects or developers to create their models in the individual views containing the primitives. Only if the primitive that is needed has not yet been mined, additional effort for documenting the primitive is necessary. However this extra effort is a rather small investment, compared to the benefits of documenting the pattern-based design decisions, during the normal flow of architecting. The only substantial work that needs to be done in advance is the creation of the primitives repositories for other pattern languages, similarly to the two repositories presented in this paper. But this effort is required only once, and as the primitives are based on patterns – which represent a limited set of proven practices, it is expected that the specification of new primitives is rather seldom compared to the use of primitives in models. As future work, we plan to investigate what happens during the evolution of the architecture, when the design decisions have to be changed.

8. REFERENCES

[1] BOC Europe. Adonis. http://www.boc-eu.com/, 2006.

[2] P. Avgeriou, P. Kruchten, P. Lago, P. Grisham, and D. Perry. Sharig and reusing architectural knowledge–architecture, rationale, and design intent. In *ICSE COMPANION '07: Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 109–110, Washington, DC, USA, 2007. IEEE Computer Society.

[3] J. Bosch. Software architecture: the next step. In *First European Workshop on Software Architecture (EWSA)*. Springer, 2004.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented Software Architecture - A System of Patterns*. J. Wiley and Sons Ltd., 1996.

[5] J. Cardoso and A. Sheth. Semantic e-workflow composition. *J. Intell. Inf. Syst.*, 21(3):191–225, 2003.

[6] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2002.

[7] DAML Services. OWL-S 1.1 Release. http://www.daml.org/services/owl-s/1.1/, 2004.

[8] T. Gardner. Uml modeling of automated business processes with a mapping to bpel4ws. In *ECOOP Workshop on Object Orientation and Web Services*, Darmstadt, Germany, July 2003.

[9] J. Greenfield and K. Short. *Software Factories: Assembling Applications with Patterns, Frameworks, Models & Tools*. J. Wiley and Sons Ltd., 2004.

[10] N. B. Harrison, P. Avgeriou, and U. Zdun. Using patterns to capture architectural decisions. *IEEE SOFTWARE*, July - August 2007.

[11] R. Heckel, M. Lohmann, and S. Thoene. Towards a uml profile for service-oriented architectures. In *Workshop on Model Driven Architecture: Foundations and Applications (MDAFA) 2003, CTIT Technical Report TR-CTIT-03-27, University of Twente*, Enschede, The Netherlands, June 2003.

[12] C. Hentrich and U. Zdun. Patterns for process-oriented integration in service-oriented architectures. In *Proceedings of 11th European Conference on Pattern Languages of Programs (EuroPlop 2006)*, Irsee, Germany, July 2006.

[13] G. Hohpe and B. Woolf. *Enterprise Integration Patterns*. Addison-Wesley, 2003.

[14] IDS Scheer. Aris Platform. http://www.idsscheer.de/germany/products/53956, 2006.

[15] IEEE. Recommended Practice for Architectural Description of Software Intensive Systems. Technical Report IEEE-std-1471-2000, IEEE, 2000.

[16] A. G. J. Jansen, J. van der Ven, P. Avgeriou, and D. K. Hammer. Tool support for architectural decisions. In *6th IEEE/IFIP Working Conference on Software Architecture (WICSA)*, Mumbai, India, January 2007.

[17] D. Schmidt and F. Buschmann. Patterns, frameworks, and middleware: Their synergistic relationships. In *25th International Conference on Software Engineering*, pages 694–704, May 2003.

[18] H. Tran, U. Zdun, and S. Dustdar. View-based and model-driven approach for reducing the development complexity in process-driven soa. In *Proceedings of International Conference on Business Processes and Services Computing*, Leipzig, Germany, Sep 2007.

[19] M. Vokac and J. M. Glattetre. Using a domain-specific language and custom tools to model a multi-tier service-oriented application - experiences and challenges. In *Proc. of Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005*, pages 492–506, Montego Bay, Jamaica, October 2005.

[20] H. Wada, J. Suzuki, and K. Oba. Modeling non-functional aspects in service oriented architecture. In *Proc. of the 2006 IEEE International Conference on Service Computing*, Chicago, IL, September 2006.

[21] U. Zdun and P. Avgeriou. Modeling architectural patterns using architectural primitives. In *Proceedings of the 20th ACM Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA 2005)*, pages 133–146, San Diego, CA, USA, October 2005. ACM Press.

[22] U. Zdun and S. Dustdar. Model-driven integration of process-driven soa models. *Accepted for publication in Invited to the International Journal of Business Process Integration and Management (IJBPIM)*, 2007.

[23] U. Zdun, C. Hentrich, and S. Dustdar. Modeling process-driven and service-oriented architectures using patterns and pattern primitives. *Accepted for publication in ACM Transactions on the Web (TWEB)*, 2007.

[24] U. Zdun, C. Hentrich, and W. van der Aalst. A survey of patterns for service-oriented architectures. *International Journal of Internet Protocol Technology*, 1(3):132–143, 2006.

[25] O. Zimmermann, P. Krogdahl, and C. Gee. Elements of Service-Oriented Analysis and Design: An interdisciplinary modeling approach for SOA projects. http://www-128.ibm.com/developerworks/webservices/library/ws-soad1/, Jun 2004.