Model-Driven and Pattern-Based Integration of Process-Driven SOA Models

Uwe Zdun and Schahram Dustdar
Distributed Systems Group
Information Systems Institute
Argentinierstrasse 8/184-1, A-1040 Wien
Austria
E-mail: {zdun|dustdar}@infosys.tuwien.ac.at

Abstract Service-oriented architectures (SOA) are increasingly used in the context of business processes. However, the modeling approaches for process-driven SOAs do not yet sufficiently integrate the various kinds of models relevant for a process-driven SOA – ranging from process models to software architectural models to software design models. We propose to integrate process-driven SOA models via a model-driven software development approach that is based on proven practices documented as software patterns. We introduce pattern primitives as an intermediate abstraction to precisely model the participants in the solutions that patterns convey. To enable model-driven development, we develop domain-specific modeling languages for each kind of process-driven SOA model – based on meta-models that are extended with the pattern primitives. The various process-driven SOA models are integrated in a model-driven tool chain via the meta-models. Our tool chain validates the process-driven SOA models with regard to the constraints given by the meta-models and primitives.

Keywords: SOA, Process-Driven SOA, Software Patterns, Services Modeling

Reference to this paper should be made as follows: ...

Biographical notes: Uwe Zdun is an assistant professor at the Vienna University of Technology. His research interests include software patterns, software architecture, SOA, distributed systems, language engineering, and object orientation. He received his doctoral degree in computer science from the University of Essen in 2002, and his his habilitation degree (venia docendi) from Vienna University of Economics and BA in 2006. He is coauthor of the books Remoting Patterns (John Wiley & Sons, 2004) and Software-Architektur (Elsevier/Spektrum, 2005).

Schahram Dustdar is a Full Professor for Internet Technologies at the Distributed Systems Group, Information Systems Institute, Vienna University of Technology (TU Wien) where he is director of the Vita Lab and Honorary Professor of Information Systems at the Department of Computing Science at the University of Groningen (RuG), The Netherlands. He received his M.Sc. (1990) and PhD. degrees (1992) in Business Informatics (Wirtschaftsinformatik) from the University of Linz, Austria. In April 2003 he received his habilitation degree (venia docendi).

1 INTRODUCTION

Many service-oriented architectures (SOA) provide a service composition layer that introduces a process engine as the top-level layer (cf. Zdun et al. (2006)). Services realize individual activities in the process. This kind of architecture is called *process-driven SOA*. The main goal of process-driven SOAs is to increase the productivity, efficiency, and flexibility of an organization. This is achieved by aligning the high-level business processes with the technical IT services. That is, the business goals get closer integrated with the IT architecture.

One of the most important characteristics of SOAs suggests heterogeneity of technologies and integration across vendor-specific technologies (cf. Vinoski (2003)). This, however, yields an important challenge for modeling process-driven SOAs: Many modeling domains need

to be considered, and the different kinds of models need to be integrated. Among many other modeling domains, we need to consider component architectures, message flows, transactions, security, workflows/business processes, programming language (snippets), business object designs, and organizational models. Application domains additionally introduce domain models, such as banking or insurance domain models. Also, implicit or explicit models for integrating existing legacy systems are needed, sometimes using different concepts than the rest of the SOA.

In other words, a central challenge for modeling process-driven SOAs is that we generally need to integrate different kinds of models and abstractions. This problem is challenging because so far there is no modeling approach for integrating all these kinds of models.

In this paper, we propose a concept for a model-driven tool chain that addresses these challenges through model-driven software development (MDSD) (cf. Stahl & Voelter (2006), Greenfield & Short (2004)). Our concept is based on the precise specification of the models in domain-specific languages (DSL) which are themselves precisely specified by meta-models and constraints on them. The code in the executable languages is generated from the models expressed in the DSLs. That is, the integration issues raised above are solved at the meta-model level.

Our tool chain and MDSD concepts break the integration issues down to the problem of finding adequate meta-models for representing all concerns to be modeled in the various modeling languages used in the modeling domains. In this paper, we propose to develop the meta-models according to proven practices that can be found in existing process-driven SOAs. Our assumption is that using proven practices as a foundation for meta-modeling leads to a close match between the modeling abstractions and the real world requirements.

In our approach, software patterns are used to describe the proven practices. Software patterns capture reusable design knowledge and expertize that provides proven solutions to recurring software design problems that arise in particular contexts and domains (cf. Schmidt & Buschmann (2003)). A software pattern, however, is described in an informal form and cannot easily be described formally, e.g., by using a parameterizable, template-style description. Hence, as such, patterns are not usable as elements of meta-models. We remedy this problem by introducing an intermediate abstraction, called pattern primitives. A pattern primitive is a fundamental, precisely specified modeling element in representing a pattern.

Our general approach to apply pattern primitives for process-driven SOAs is to define meta-models for all kinds of models that are needed, and specify the pattern primitives as extensions of these meta-models. The connection between the various kinds of models and the validation of the models – with regard to model integrability and consistency in and across the modeling domains – is the main task of our model-driven tool chain concepts. To demonstrate our approach, we use a precisely specified subset of UML2 and OCL to depict the various refinements of processes in process-driven SOA models. We use the precisely specified subset of UML2 and OCL only for demonstration purposes. Any other precisely specified meta-model can be used as well. "Precisely specified" means in this context that no informal elements are part of the meta-models or constraints (like the informal constraints used in some parts of the UML standard) and the semantics of the meta-model or constraint elements are specified in the generator – to define precisely how models need to be validated and how valid code can be generated.

In this paper, we first provide the background on MDSD and patterns/pattern primitives in Section 2. Next, in Section 3 we explain our model-driven tool chain to give an overview of our approach. In Section 4 we explain how we use meta-models to integrate process-driven SOA models. In Section 5 we explain the pattern primitives approach for process-driven integration of services using flow abstractions as the primary modeling domain. In Section 6 we demonstrate how architectural abstractions – as one example of another modeling domain – can be integrated with the flow models. We explain all primitive models with running examples from a pattern language for process-driven integration of services, which we have implemented in our MDSD tool chain to validate our approach.

2 BACKGROUND ON MDSD

2.1 Model-driven Software Development

Our approach to model-driven software development (MDSD) (cf. Stahl & Voelter (2006), Greenfield & Short (2004))[1] for process-driven SOAs is based on the notion of domain-specific languages (DSL) for modeling the various types of models. Domain-specific languages are "small" languages that are tailored to be particularly expressive in a certain problem domain. The DSL describes knowledge via a graphical or textual syntax (the DSL's concrete syntax in the terminology of Greenfield & Short (2004)), which is tied to domain-specific modeling elements through a precisely specified language meta-model (the DSL's abstract syntax in the terminology of Greenfield & Short (2004)). The meta-model can be instantiated in concrete application models. The semantics of the DSL are defined via unambiguous specifications of model-to-model or model-to-code transformations. There are different ways to specify transformations, such as transformation rules, imperative transformations, or template-based transformations.

The meta-models presented in this paper are based on the UML2 meta-model (and extensions of it): For example we use UML2 activity diagrams to model flow abstractions and UML2 class/component diagrams to model object-oriented design and architecture models. To make the UML2 meta-models usable in our approach, we first specify a precise sub-set of the UML meta-model (using model

[1] Please note that the OMG's MDA proposal is one specific MDSD approach that has some notable differences to our MDSD approach – especially in its focus on interoperability and platform independence.

elements, transformations, and constraints). We use the UML only for demonstration purposes (and because it was required in some of our projects); any other meta-models can be used equally. For instance, if a project goal is to generate BPEL code using the full BPEL specification, it is advisable to use a DSL that is closer to BPELs constructs to minimize potential transformation problems.

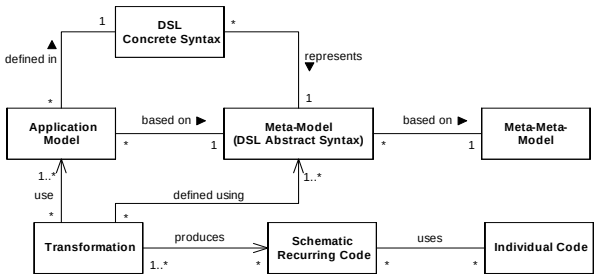

Figure 1: Relations of Artifacts in MDSD

Meta-models are defined in terms of a meta-meta-model. In UML (and the OMG's MDA proposal), for instance, this is MOF. Most MDSD tools support their own meta-meta-model, which basically represents a mapping from meta-model definitions to the implementation of the MDSD tool chain. Below, in the examples from our prototype, we use a simple meta-meta-model to define both the UML2 meta-model and pattern primitives extensions. It is not particularly important for our approach, which meta-meta-model is used, there just must be some way to specify the relationships and transformations between a meta-model and the target models/code of the transformations. The meta-meta-model is not visible to developers who build application models, but only to those who build meta-models.

The ultimate goal of the transformations in MDSD tools is to generate code in executable languages, such as programming languages or process execution languages. The MDSD tools are used to generate all those parts of the executable code which are schematic and recurring, and hence can be automated. Of course, some code must be hand-written either because it is individual code for a system or the semantics of the code are not fully covered by the DSLs (yet). The individual code and the generated code use each other and interact through well defined interfaces.

Figure 1 summarizes the relations of artifacts in MDSD.

2.2 Software Patterns and Pattern Primitives

Software patterns and pattern languages have gained wide acceptance in the field of software development, because they provide systematic reuse strategies for design knowledge (cf. Schmidt & Buschmann (2003)). A pattern encodes proven practice in form of a reusable design solution to a recurring design problem. A pattern language is a collection of patterns that solves the prevalent problems in a particular domain and context. Patterns informally describe many possible variants of one software solution that a human developer or designer can recognize as one and the same solution.

Even though these properties of the pattern approach are highly valuable in the software design process, they also make pattern instances hard to trace in the models and implementations. To overcome this problem, we introduced an approach to document precisely specified primitive abstractions that can be found in the patterns (cf. Zdun & Avgeriou (2005)). Documenting pattern primitives means to find precisely describable modeling elements that are primitive in the sense that they represent basic units of abstraction in the domain of the pattern. Our original pattern primitives concept presented in Zdun & Avgeriou (2005) is only targeted at modeling architectural patterns. In this realm, basic architectural abstractions like components, connectors, ports, and interfaces are used. An interesting challenge in describing the pattern primitives for the patterns of process-driven SOA is that this area requires various design and architecture concepts, as well as various design and implementation languages.

We specify an extension of a metaclass for each elicited primitive, using the UML's extension mechanisms: stereotypes, tag definitions, and constraints. We use the Object Constraint Language (OCL) to precisely specify the constraints of the primitives.

3 MDSD FOR PROCESS-DRIVEN SOA

3.1 Model-driven Tool Chain

Our approach requires a model-driven tool chain that loosely follows an architecture similar to our tool chain depicted in Figure 2. 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 directly in the textual DSL syntax. If a UML tool is used, the XMI export is transformed into the textual DSL syntax.

We use Frag (cf. Zdun (2006, 2005)) as the syntactic foundation of the textual DSLs and for defining the meta-models of the DSLs. Frag's main goal is to provide a tailorable language. Among other things, Frag supports the tailoring of its object system and the extension with new language elements. Hence, Frag provides a good basis for defining a UML2-based textual DSL because it is easy to define a meta-meta model on top of which we can define the UML meta-classes. Frag automatically provides us with a syntax for defining application models using the UML2 meta-classes. In addition, we have defined a constraint language which follows the OCL's constructs.

The model validator gets all input models and validates the conformance of the application models to the meta-models. It also checks all OCL constraints, in particular, the constraints given by the pattern primitive definitions.

After the model is validated it is transformed into an EMF model, which is understood by the code generator. We then generate code in executable languages using transformation specifications provided to the code generator.

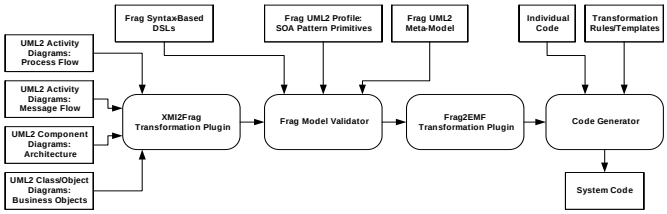

Figure 2: Tool Chain Overview

3.2 Model Integration Concepts: Meta-meta-model Based Integration

The model-driven concepts described in the previous section only concentrate on the individual modeling domains. For integration of the models, we propose further integration concepts that extend the general model-driven approach. Because they are independent of external tools, languages, or models, in our concept, the central point of integration are the meta-models that we need to define for the DSLs. Also, they are located at the central place of the model-driven architecture: at the point in the tool chain where all different models are assembled.

We propose to define the meta-models on top of one common meta-meta-model. The meta-meta-model can be very simple, or more elaborate like MOF. The most important criterion for the meta-meta-model is that the elements of the meta-meta-model allow the model validator to check models against the meta-models. In addition, it should be possible to define a constraint language using the meta-meta-model, with which models can be constrained at the meta-level and hence validated at the model level.

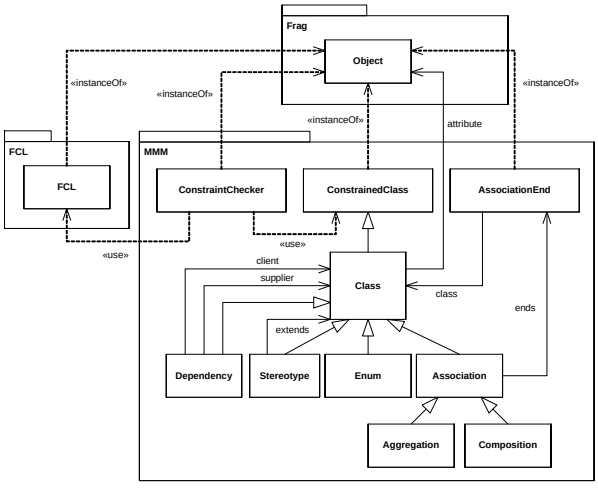

Figure 3: Meta-meta-model Excerpt

As an example, Figure 3 shows the relevant excerpt of the meta-meta-model that we use in Frag to define precisely specified sub-sets of the UML2 meta-model. This meta-meta-model is very simple and reuses Frag's language features wherever possible. It is derived from the most general class in the Frag object system: `Object`. The meta-meta-model classes are sub-classes of `ConstrainedClass` which allows us to add OCL-style constraints to classes. The convenience class `ConstraintChecker` looks up all `ConstrainedClass` instances via reflection and checks the constraints. Constraints are specified in a language similar to OCL (defined using the class `FCL`). The meta-models are defined using `Class`. We introduce also a number of relationships between classes: Dependencies, Associations, Compositions, and Aggregations. In addition, typed attributes can be specified. Please note that we do not define the generalization relationship, because multiple inheritance is suitably predefined by Frag and we can reuse this implementation. The `Stereotype` class defines the UML2 extends-relationship, which is used to extend meta-classes. `Enum` is a convenience class to define Enumeration types.

3.3 Model Integration Concepts: Proven Practices Based Integration

Besides the common meta-meta-model concept, we use proven practices descriptions as the second central model integration concept: As explained above, we use software patterns to describe proven practices of process-driven SOAs. Patterns have two characteristics which make them useful for model integration across modeling domains:

- Patterns describe recurring solutions in a particular problem domain in an informal and holistic manner. Hence, in contrast to most formal modeling notations, they do not abstract away from details that go beyond a specific modeling domain's abstractions, but instead explain the full solution. That is, if the solution has, for instance, implications for the workflow, the organization, and the software architecture, all these solution elements are described.

- As proven practice descriptions, patterns encode the recurring themes in the same kinds of models. Hence, they are also a good basis for defining a common meta-model for a modeling domain, because patterns typically describe the established, stable abstractions that are used across different modeling approaches and execution languages.

Because patterns are defined only informally, we use pattern primitives as an intermediary abstraction to represent the primitive concerns in the patterns precisely. At this point, it is very important that we use a common meta-meta-model and a common constraint language to define the meta-models that represent the abstract syntaxes of the DSLs: This way, the primitives can be connected via constraints, and also primitives that cut across different models can be defined. The model validator can check all structural properties and constraints in the complete model, even if modeling domains are crossed.

4 META-MODELS FOR SOA INTEGRATION

There are many modeling domains that play a role for a process-driven SOA. In our tool chain we have so far con-

centrated on a sub-set of these domains that deals with the integration of processes and services. In this context, the following types of languages/models are typically used: component architectures, message flow specifications, workflow or business process languages, programming languages and snippets written in programming languages, and business object design models.

For our tool chain, we model both, message flow specifications and workflow or business process languages, using extensions of UML2 activity diagrams[2].

Component architectures are modeled using UML2 component diagrams. Business object design models are modeled using UML2 class diagrams. In this paper, we will concentrate on examples that illustrate the integration of component architectures and flow abstractions, but the integration with business object design models can be done analogously. Programming language snippets are introduced as individual code (as explained in Section 3.1, cf. Figure 2).

As an example for a meta-model definition let us consider the central flow abstractions: The different models that are relevant for a process-driven SOA come together in various kinds of "flow" models. There are flow models for long-running business processes, activity steps in long-running processes, short-running technical processes, and activity steps in short-running technical processes. Even though these flow models have highly different semantic properties, they share the same basic flow abstraction concept, and at the same time they are a kind of glue for all the other models that are involved in a process-driven SOA (such as architecture and design models).

We can define meta-models by instantiating the meta-meta-model classes from Figure 3. In addition to the graphical DSL syntaxes that follow the UML's symbols (see the UML2 standard for details about the UML meta-models and graphical syntax), we also introduce a textual Frag syntax. To get a feel for these textual DSLs, here's a small excerpt of the Frag code specifying some classes and relationships. In the remainder of the paper, we omit the textual DSL syntaxes as they are pretty much defined in the same way for the different kinds of meta-models. That is, there is a 1:1 mapping between the textual syntaxes and UML's graphical syntaxes for the various model types.

```
MMM::Class create Activity
MMM::Class create ActivityNode
MMM::Composition create ActivityNodes -ends {
  {Activity -roleName activity -multiplicity 0..1
     -navigable 1 -aggregatingEnd 1}
  {ActivityNode -roleName node -multiplicity * -navigable 1}
}
MMM::Class create ActivityEdge
MMM::Composition create ActivityEdges -ends {
  {Activity -roleName activity -multiplicity 0..1
     -navigable 1 -aggregatingEnd 1}
  {ActivityEdge -roleName edge -multiplicity * -navigable 1}
}
...
```

[2]Please note that in both cases, long running business processes and short running technical processes, the UML2 activity diagrams must be extended to depict relevant additional information.

5 SOA Patterns and Pattern Primitives

We next discuss how to extend the meta-models with pattern primitive extensions. Before we can go into detail, we first give an overview of the pattern language from which we derive the pattern primitives.

5.1 Overview: Patterns for process-oriented integration of services

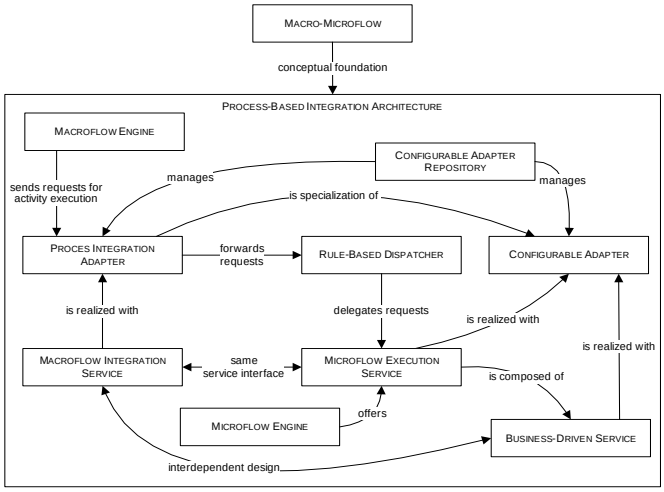

Figure 4: Overview: Pattern language for process-oriented integration of services

We have mined the pattern primitives from a pattern language for process-oriented integration of services (for details see Hentrich & Zdun (2006)). The patterns and pattern relationships are shown in Figure 4. In the pattern language, the pattern MACRO-MICROFLOW 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 a layered architecture, which is following the MACRO-MICROFLOW pattern.

The remaining patterns in the pattern language provide detailed guidelines for the design of a PROCESS-BASED INTEGRATION ARCHITECTURE. The automatic functions required by macroflow activities from external systems are designed and exposed as dedicated MACROFLOW INTEGRATION SERVICES. PROCESS INTEGRATION ADAPTERS connect the specific interface and technology of the process engine to an integrated system. A RULE-BASED DISPATCHER picks up the (macroflow) activity execution requests and dynamically decides based on (business) rules, where and when a (macroflow) activity is executed. A CONFIGURABLE ADAPTER connects to another system in a way that allows to easily maintain the connections, considering that interfaces may change over time. A CONFIGURABLE ADAPTER REPOSITORY manages CONFIGURABLE ADAPTERS as components, such that they can be modified

at runtime without affecting the systems sending requests to the adapters. A MICROFLOW EXECUTION SERVICE abstracts the technology specific API of the MICROFLOW ENGINE and encapsulates the functionality of the microflow as a service. A MACROFLOW ENGINE allows for configuring business processes by flexibly orchestrating execution of macroflow activities and the related business functions. A MICROFLOW ENGINE allows for configuring microflows by flexibly orchestrating execution of microflow activities and the related BUSINESS-DRIVEN SERVICES. To define BUSINESS-DRIVEN SERVICES, high-level business goals are mapped to to-be macroflow business process models that fulfill these goals and more fine grained business goals are mapped to activities within these processes.

Figure 5 shows an exemplary configuration of a PROCESS-BASED INTEGRATION ARCHITECTURE, in which multiple macroflow engines execute the macroflows. Process-integration adapters are used to integrate the macroflows with technical aspects. A dispatching layer enables scalability by dispatching onto a number of MICROFLOW ENGINES. Business application adapters connect to backends.

5.2 Pattern primitives for Process and Service Integration

In this section, we present the pattern primitives for flow abstractions that we have mined from the pattern language explained in the previous section. We will concentrate only on one detailed primitive example and summarize a few other primitives that are needed for the examples in the following sections in Table 1. The other primitives in the full catalog of primitives have been defined in the same way.

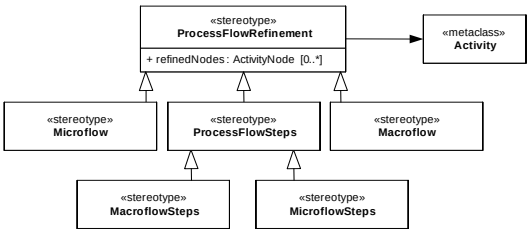

Figure 6: Stereotypes for Process Flow Refinements

Each primitive is precisely specified in the context of the UML2 meta-model using OCL constraints. To illustrate the precise specification of the primitives let us consider the Macro-Microflow Refinement Primitive. This primitive models the situation that Microflow Models are allowed to refine Macroflow Models. In addition to macroflows and microflows, we must consider the Macroflow Steps and Microflow Steps models, introduced by the Process Flow Steps primitive: A process activity node in a macroflow or microflow can optionally be refined by a number of sequential steps that detail the steps performed to realize the process activity.

To model this primitive, we first must introduce UML2 stereotypes to distinguish the different kinds of refined/refinement activities (see Figure 6) in the UML2 models. We can model the Macro-Microflow Refinement primitive by constraining the Microflow Activities. In particular, if a Microflow activity refines another activity, then this other activity must be itself stereotyped as Macroflow, MacroflowSteps, or Microflow. This can be precisely specified using the following OCL constraint:

```
context Microflow inv:
  self.refinedNodes->forAll(rn |
    Macroflow.allInstances()->exists(a |
      a.baseActivity = rn.activity) or
    MacroflowSteps.allInstances()->exists(a |
      a.baseActivity = rn.activity) or
    Microflow.allInstances()->exists(a |
      a.baseActivity = rn.activity))
```

We have also provided a textual Frag DSL for expressing such constraints. The OCL constraints in UML models can be automatically mapped to this constraint DSL. The task of such constraints is basically to limit the use of the primitives to the acceptable parameters – following the pattern descriptions in which the primitives are used – but no further. For each primitive we have hence described all such constraints (the others are omitted here for space reasons). Thus each primitive describes a precisely specified, parameterizable building block that can be used to specify the corresponding patterns.

5.3 Modeling patterns using the pattern primitives for process-driven service integration

The pattern primitives are not yet linked to the patterns. The patterns cannot be themselves precisely specified, but we can identify the pattern primitives that occur in individual patterns. For instance, some of the primitives are mandatory in a pattern, others are optional, still others are only used in specific variants, etc. This mapping of patterns to pattern primitives hence provides us with modeling constructs that can be differently combined for different pattern instances, but must conform to the pattern-to-primitive mapping. If a primitive is used in a pattern instance, all constraints of the primitive must be fulfilled. Hence, the primitives precisely specify the proven practices documented in the patterns. In the remainder of this section, we illustrate our approach using the example of modeling the MACRO-MICROFLOW pattern. The other patterns are modeled following the same approach.

The MACRO-MICROFLOW pattern strictly separates the macroflow from the microflow, and uses the microflow only for refinements of the macroflow activities. Both in macroflows and microflows we can observe refinements. The different kinds of refinement can be modeled using the Process Flow Refinement primitive. Process Flow Refinement is a generic primitive that can be used for modeling all kinds of process refinements.

Figure 7 shows an example for Macro-Microflow refinement. We describe all models in Figure 7 essentially in the same way. The model integration of the short-running message flow models and the long-running business models is done by extending the models with the respective

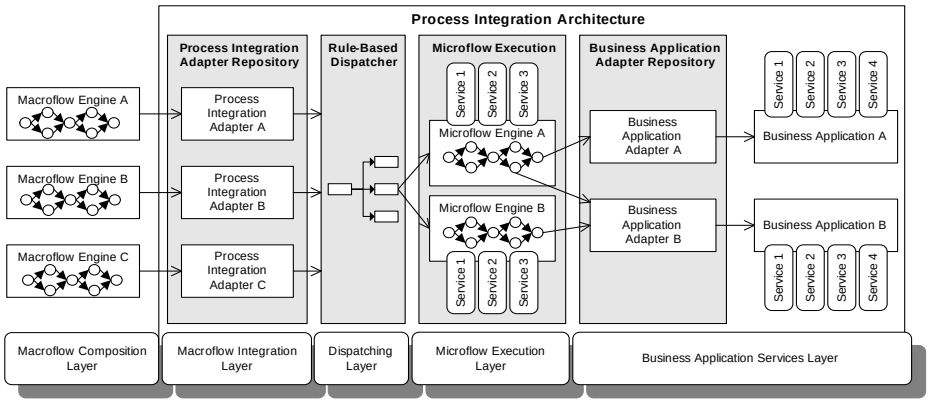

Figure 5: Example Configuration of a Process-based Integration Architecture

Primitive Name	Description	Modeling Solution
Process Flow Refinement	A macroflow or microflow is refined using another process flow.	The Activity metaclass is extended with the stereotype ProcessFlowRefinement, which also introduces tagged values for identifying the refinement.
Process Flow Steps	A macroflow or microflow is refined by a number of sequential steps.	A specialization of the ProcessFlowRefinement stereotype, called ProcessFlowSteps, is introduced and constrained to be a strictly sequential flow.
Macroflow Model	A macroflow can be refined by other macroflows or macroflow steps.	Macroflows are modeled by a ProcessFlowRefinement stereotype, called Macroflow, and macroflow steps are modeled as a specialization of ProcessFlowSteps, called MacroflowSteps.
Microflow Model	A microflow can be refined by other microflows or microflow steps.	The microflow model is modeled analogous to the Macroflow Model primitive: The Microflow and MicroflowSteps stereotypes are introduced.
Macro-Microflow Refinement	Microflow Models are allowed to refine Macroflow Models.	The Microflow Model primitive is extended: If refinedNodes of a Microflow is not empty, the Microflow is a refinement of a Microflow, a Macroflow, or MacroflowSteps.

Table 1: Overview: Process Flow Refinement Primitives

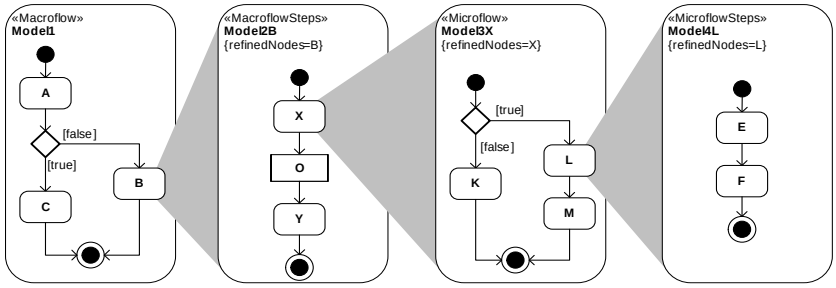

Figure 7: Macro-Microflow modeling example

stereotypes and tag values. After these stereotypes have been defined, the OCL constraints of the primitives enforce that those four models can only be composed in a way that is valid according to the Macro-Microflow Refinement primitive. Figure 7 hence shows a model conforming to the constraints. Again, the models can be mapped to a textual DSL expressing models defined on top of the DSLs for meta-models, primitives, and constraints introduced earlier.

Numerous other variants of the MACRO-MICROFLOW pattern are possible and can be modeled with the primitives introduced. Please note that the flexibility of model assembly through primitives is a very important characteristic of our approach, because it enables us to represent the inherent variability of software pattern solutions.

6 INTEGRATING ARCHITECTURAL MODELS

The MACRO-MICROFLOW pattern has implications for short-running message flow models and the long-running business models, and we were able to integrate the two model types (and even add macroflow/microflow steps as another kind of modeling abstraction). In our approach, these two kinds of models are modeled with the same model type: activity diagrams. The patterns, however, also have implications for other model types, such as the architectural components in the system or business object models. To model those abstractions, we additionally need to consider architectural abstractions and object-oriented design abstractions. It is typical for patterns that they have implications for different kinds of models; hence the conceptual integration approach, demonstrated in this section for one

pattern/primitive example, is also used to model the other patterns and primitives, as well as other modeling domain combinations.

In UML, business objects can be modeled using class diagrams. Architectural abstractions can be modeled via component diagrams. Component diagrams are a specialization of class diagrams. Therefore, our approach for integrating these models with flow models is very similar for both class diagrams and component diagrams. Hence, we demonstrate our approach only for one of those abstractions in depth: architectural components.

Our approach is to first find suitable pattern primitives for modeling the architectural abstraction corresponding to the patterns. Where further integration between modeling domains is needed, we model the flow model and architectural model with overlapping constraints and primitives that cut across model boundaries.

6.1 Modeling Architectural Abstractions with Pattern Primitives

Architectural modeling with pattern primitives follows the same approach as introduced for the flow abstractions. In the context of architectural abstraction, we have introduced similar primitives for architectural patterns (see Zdun & Avgeriou (2005)). Let us consider the Callback primitive as an example: In a process-based integration architecture, different kinds of components are connected. Figure 5 shows an exemplary larger configuration, in which multiple macro-/microflow engines and a dispatcher are used. The components are typically connected via asynchronous messaging. The modeling support for a process-based integration architecture should allow for flexibly assembling different kinds of architectural models.

The Callback primitive (cf. Zdun & Avgeriou (2005)) can be used to model the reactive behavior in this architecture: 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 a callback architecture properly in UML, we propose five stereotypes: IEvent, ICallback, EventPort, CallbackPort, and Callback. Again, we have precisely specified the constraints using OCL (see Zdun & Avgeriou (2005) for details), and defined the meta-model and constraints in Frag, so that we can use the constraints on components that represent the process-based integration architecture components.

All components in a PROCESS-BASED INTEGRATION ARCHITECTURE are interconnected following the callback style because they use asynchronous communication. The event ports of each layer are listening to events from the higher-level layer, and when an event arrives, they call into the lower-level layer. Once a result is received, it is propagated back into the higher-level layer using a Callback. Figure 8 shows an example UML2 model for a Callback configuration modeling the situation from Figure 5 in a more precise way. For the architectural primitives we have introduced textual DSLs, just like the ones for the flow abstraction primitives introduced before.

6.2 Integrating Architectural and Flow Abstraction Models

In addition to the architectural flexibility of the PROCESS-BASED INTEGRATION ARCHITECTURE pattern, we need to model the pattern's constraints. If the pattern implementation follows the MACRO-MICROFLOW pattern, analogous constraints to the macro-microflow refinement in the flow models must be introduced, such as: 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.

In this situation, we can use another architectural primitive from Zdun & Avgeriou (2005): Layering. Layering describes groups of components and further constrains them. Specifically, it 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. To model Layering in UML2, we introduce the `Layer` stereotype, which specializes the `Group` stereotype (which itself is an extension of the Package metaclass). We also impose the following constraints: a component can only be member of one layer and not multiple layers; components who are members of layer X may call their fellow components in layer X, as well as components in layer $X-1$ but not in other layers (e.g. $X-2$ and below). Also, we introduce the tag definition layerNumber for Layers which represents the number of the layer in the ordered structure of layers. Figure 9 shows an UML2 model that extends the model from Figure 8 using the Layering primitive.

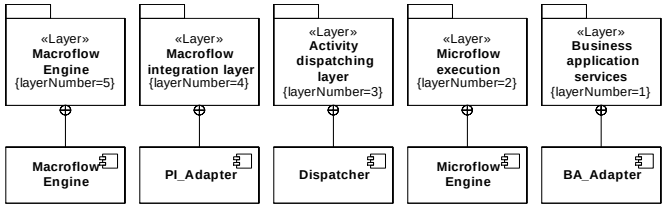

Figure 9: Extending the Example Configuration with Layering

In this example, we have modeled the integration of the macroflow-microflow refinements at the flow model level with the architectural components by introducing OCL constraints for the refinements between the flow models, and by adding similar constraints to the architectural model. That is, the primitive Macro-Microflow Refinement was used at the flow model level and Callback/Layering were used at the architectural level.

Sometimes this is not enough, and it is necessary to extend this strategy and add a direct relationship between the flow models and the architectural models. In such cases, primitives and OCL constraints that contain ele-

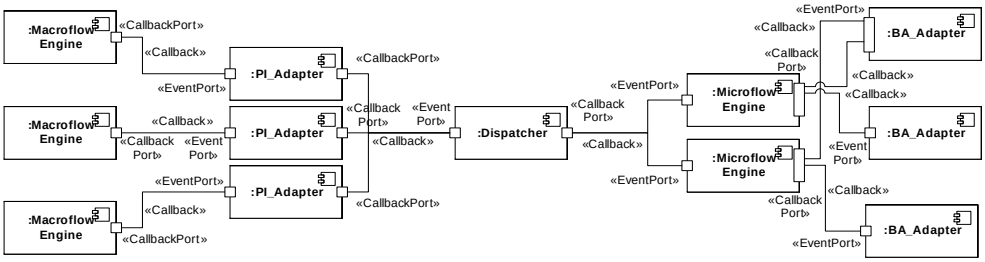

Figure 8: UML2 Model for the Example Configuration

ments from both model types need to be specified. That is, the OCL constraints cut across the model types and hence allow for validating that the models do not violate integration concerns between the model types.

7 RELATED WORK AND EVALUATION

Our approach supports models for business processes, message flows, OO design, and software architecture – and programming language code/snippets provided as individual code in the MDSD tool chain. Our approach is extensible with new model types, especially domain-specific models. We plan to extend our approach in additional relevant modeling domains, such as organizational models or human-interaction models. None of the related approaches offers sufficient support for all these model types. Most related work focuses only on one type of modeling domain.

There are only a few exceptions: Zimmermann et al. (2004) present a generic modeling approach for SOA projects. As in our approach, the approach by Zimmermann et al. is based on project experiences and distills proven practices. The approach also integrates multiple kinds of models for a SOA: object-oriented analysis and design, enterprise architecture, and business processes.

Business process management tools, such as Adonis (cf. BOC Europe (2006)) or Aris (cf. IDS Scheer (2006)), describe a holistic model of business process management, ranging from strategic decisions to the design of business processes. They are integrated with standard model types and extensible with new model types. Such tools represent important prior art in the field of model integration. But they 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 or be extended to model the DSLs.

There are many modeling approaches for business processes, such as Event-Driven Process Chains (EPC) (cf. Keller et al. (1992)) and the BPMN (cf. Object Management Group (2006)). Our approach has in common with these approaches that we use the flow abstraction as the central modeling abstraction. Unlike the BPMN, our approach is based on a precisely specified meta-model. Among others, Kindler (2006) has proposed formal semantics for EPCs. In contrast to modeling approaches for business processes, our approach allows to integrate other model types of a process-driven SOA. That is, we take a more "integrative" view than those more specific approaches.

Our approach extends the MDSD concept proposed for instance in Stahl & Voelter (2006), Greenfield & Short (2004) with the idea to use one common meta-meta-model for model integration, primitives as modeling constructs based on proven practices, and model validation tools for these concepts. In Greenfield & Short (2004), Chapter 13, 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. This view does not contradict our approach, but our approach goes beyond this vision. Through the common meta-meta-model we can integrate any kind of model types; hence, process descriptions are not only a platform, but a first-class model type. In this sense, our approach is also related to the OMG's MDA proposal, which is a specific MDSD approach focusing on interoperability and platform independence through the distinction of platform-independent models (PIM) and platform-specific models (PSM). Again, our general approach can be used as an extension of MDA, even though our concrete realization follows more closely the DSL-based MDSD approach (cf. Stahl & Voelter (2006)).

Our approach is not the only approach that is based on proven practices, but only our approach and the workflow patterns approach by van der Aalst et al. (2003) combine proven practices and precisely specified models. The workflow patterns are formalizable constructs (e.g., formalized in the Petri-net-based language YAWL). In YAWL, the workflow patterns are provided as language constructs; hence in the workflow patterns approach the flexibility of assembly of pattern primitives is not (yet) supported, because the variation points offered by the primitives are not offered by the workflow patterns. To support a similar approach as ours, it would be necessary to mine higher-level patterns in workflows that provide guidance on how to assemble the workflow patterns to larger structures.

Some other approaches define particular aspects of service or business process composition using formal specifications, such as the activity-based finite automata based approach by Gerede et al. (2004) or the interval tempo-

ral logic approach by Solanki et al. (2004). Desai et al. (2005) propose to abstract business processes using interaction protocol components which represent an abstract, modular, and publishable specification of an interaction among different partner roles in a business process. These approaches aim at model-based verification. Our approach is not designed for this goal. Of course, it is possible to define verifiable models through meta-models and extend our approach, but this has not yet been the focus of our work.

8 CONCLUSION

In this paper, we have introduced a concept for model-driven development of process-driven SOAs that is based on proven practices. We have especially focused on the aspect of model integration by introducing an approach that is based on a common meta-meta-model from which concrete meta-models for DSLs are derived. In the different DSLs and their respective meta-models, proven practices (described as software patterns) are precisely specified as modeling primitives, and their constraints can be validated for all instances of all different meta-models. We have shown in the examples how to integrate message flow models, business process models, and architectural models. The approach is, however, applicable for all other kinds of process-driven SOA models for which a precise meta-model is or can be specified. Our tools and DSLs can be flexibly used in model-driven development for precisely specifying process-driven SOAs, validating the models, and code generation for executable languages.

References

BOC Europe (2006), 'Adonis', http://www.boc-eu.com/.

Desai, N., Mallya, A. U., Chopra, A. K. & Singh, M. P. (2005), 'Interaction protocols as design abstractions for business processes', *IEEE Transactions on Software Engineering* **31**(12), 1015–1027.

Gerede, C. E., Hull, R., Ibarra, O. & Su, J. (2004), Automated composition of e-services: Lookaheads, *in* 'Proceedings of the International Conference on Service Oriented Computing (ICSOC 2004)', New York, NY, US, pp. 252–262.

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

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

IDS Scheer (2006), 'Aris Platform', http://www.idsscheer.de/germany/products/53956.

Keller, G., Nuettgens, M. & Scheer, A.-W. (1992), Prozessmodellierung auf der grundlage ereignisgesteuerter prozessketten (EPK), Technical Report Veroeffentlichungen des Instituts fuer Wirtschaftsinformatik (IWi), Heft 89, Universitaet des Saarlandes.

Kindler, E. (2006), 'On the semantics of EPCs: Resolving the vicious circle', *Data & Knowledge Engineering* **56**(1), 23–40. Elsevier.

Object Management Group (2006), 'Business Process Modeling Notation (BPMN)', http://www.bpmn.org/.

Schmidt, D. & Buschmann, F. (2003), Patterns, frameworks, and middleware: Their synergistic relationships, *in* '25th International Conference on Software Engineering', pp. 694–704.

Solanki, M., Cau, A. & Zedan, H. (2004), Augmenting semantic web service descriptions with compositional specification, *in* 'WWW '04: Proceedings of the 13th international conference on World Wide Web', pp. 544–552.

Stahl, T. & Voelter, M. (2006), *Model-Driven Software Development*, J. Wiley and Sons Ltd.

van der Aalst, W., ter Hofstede, A., Kiepuszewski, B. & Barros, A. (2003), 'Workflow patterns', *Distributed and Parallel Databases* **14**, 5–51.

Vinoski, S. (2003), 'Integration with Web services', *IEEE Internet Computing* .

Zdun, U. (2005), 'Frag', http://frag.sourceforge.net/.

Zdun, U. (2006), 'Tailorable language for behavioral composition and configuration of software components', *Computer Languages, Systems and Structures: An International Journal* **32**(1), 56–82.

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

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

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