Architectural Decision Identification in Architectural Patterns

Olaf Zimmermann
IBM Research – Zurich (when conducting this research)
ABB Corporate Research, Industrial Software Systems (at present)
Segelhofstrasse 1K, CH-5405 Baden-Dättwil, Switzerland
olaf.zimmermann@ch.abb.com

ABSTRACT
When modeling recurring architectural decisions for reuse, the boundaries of the knowledge asset under construction must be defined in a scoping step. This paper introduces and combines two supporting concepts for this step, pattern-centric decision identification rules and generic meta issues; one particular meta issue catalog is also presented. The resulting general-purpose decision identification method is validated by identifying 35 decisions that recur in enterprise application development and service-oriented architecture design.

Categories and Subject Descriptors
D.2.11 [**Software Engineering**]: Software Architecture

General Terms
Design, Human Factors, Standardization.

Keywords
Architectural Knowledge Management, Architectural Decisions, Architectural Patterns, Enterprise Applications, SOA.

1. INTRODUCTION TO SOAD
In this paper, we present the first of seven steps in the *SOA Decision Modeling Framework (SOAD)*[24]. Unlike most previous work in the architectural knowledge management community, SOAD does not solely focus on documenting decisions after-the-fact, but also on guiding design work by anticipating the decisions that will be required [25].

A central concept in SOAD is the notion of a *Reusable Architectural Decision Model (RADM)*, created and consumed in seven steps. Figure 1 on the next page introduces these seven SOAD steps along with the roles responsible for them, *knowledge engineer* and *software architect*, and the artifacts involved, *architectural patterns* described in the literature and project-level *analysis and design models*.

Knowledge asset creation. To define the boundaries of a RADM, a knowledge engineer performs the following step:

WICSA/ECSA 2012. August 20-24, Helsinki, Finland.

1. *Identify decisions* required in a domain (e.g., when applying a certain architectural style in a particular application genre). This step starts with a review of the patterns that are eligible in the domain (e.g., service consumer-provider contract, enterprise service bus, service composition, and service registry in SOA [25]). It returns a list of required and recurring decisions, to be included in the RADM.

To promote modularity and flexibility, SOAD distinguishes the identification of required and recurring decisions (step 1) from their detailed documentation (steps 2-4), also performed by the knowledge engineer:

2. *Model individual decisions*. In this step, the decisions in the list from step 1 are documented in such a way that the modeled knowledge can support the decision making on projects. The level of detail may vary by practitioner community. A metamodel, specifically designed for knowledge sharing, supports this step [26].
3. *Structure model* according to logical dependencies between decisions. The model structure developed in this step has the objective to make the RADM easy to navigate and to adapt to project needs [26].
4. *Add temporal decision order* by modeling temporal decision dependencies [26]. This order is leveraged later during decision making (step 6).

It is worth noting that steps 1 to 4 may be executed repeatedly and in an overlapping fashion to scope and populate a RADM iteratively and incrementally.

Knowledge asset consumption. Architectural Decision Models (ADMs) are created and used by software architects on projects that apply SOAD. The RADM as a reusable knowledge asset provides input to this work, which is organized in three steps:

5. *Tailor model*, creating an ADM from a RADM by taking project-specific requirements into account. An initial set of decisions required on the project is determined in this step. These may or may not appear in the tailored RADM; architectural decision knowledge can be added, updated, or deleted during the tailoring (as well as later steps).
6. *Make decisions*. In this step, architects review the architectural decision knowledge in the ADM created in step 5, match this information against the project requirements, make their decisions, and update the ADM. When locating the relevant parts of the model in a given project situation, they are assisted by the model structure

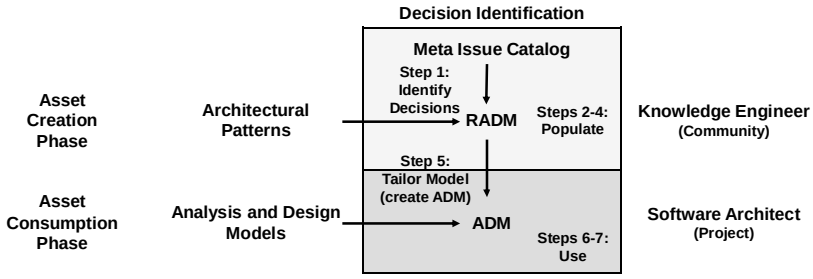

Figure 1. SOAD step 1 in context.

and the temporal order of the decisions developed in steps 3 and 4.

7. *Enforce decisions*. In this step, architects share the rationale for the decisions made in step 6 and captured in the ADM. They update other architectural artifacts accordingly. Via decision logs, they instruct the project team which chosen alternatives to implement. Furthermore, they provide fragments of development artifacts to demonstrate how to implement certain architectural concepts (as a form of coaching).

Like the earlier asset creation steps (steps 1 to 4), steps 5 to 7 also may be executed repeatedly and in an overlapping fashion. The execution rhythm depends on the software engineering methods and design practices in use (e.g., agile vs. iterative and incremental vs. big design upfront).

Steps 2 to 7 are described in our previous work [26]. In this paper, we introduce a generic *meta issue catalog* and seven *identification rules* for step 1. The meta issues in the catalog and a domain-specific set of architectural patterns serve as input to step 1 in SOAD (see Figure 1). As outlined above, the output of this step (i.e., of the execution of the identification rules) is an initial RADM enumerating the names of decisions required and recurring in the domain.

Combining meta issues, patterns, and identification rules yields a method that addresses the following *decision identification* (scoping) problem:

Which architectural decisions required (issues) recur?
Can such decisions be identified systematically in patterns?

2. RELATED WORK

State of the art. Pattern languages, genre- and style-specific extensions to software engineering methods, technical papers, and vendor documentation can be studied to identify recurring issues. In principle, these sources of information provide deep coverage of all issues. However, a vast amount of information must be studied; architectural decisions are often hidden behind various other material not targeting architects and therefore not being presented adequately [16].

The relations between architectural patterns and decisions are multi-faceted. Patterns per se do not aim at guiding the architect through the architecture design activities required once a certain pattern has been selected. The core metaphor of a pattern is *solution*, not *problem*, even if pattern templates usually contain an intent section or a problem statement [7]. Pattern authors often reverse engineer the problem statement from the solution they want to educate the readers about [11]. In the literature, we find work on documenting decisions with patterns [8], and how to combine pattern- and decision-centric design[30].

State of the practice. Decisions are often identified ad hoc based on personal experience, not via diligent literature studies, or systematic reuse of knowledge already gained. Independent of the technique in use, architects have to search for issues and *pull* the required knowledge from the literature and their experience today; methods and tools do not *push* this decision knowledge to them. As a consequence, much time is spent in the early project phases to identify relevant issues and alternatives; important issues are sometimes overlooked. This is particularly true for inexperienced architects. The assessment is subjective, drawing on input from practicing architects [26][27] and personal experience [28][29]. It is also supported by [3][20].

3. A DECISION IDENTIFICATION AND GUIDANCE MODEL SCOPING METHOD

To help knowledge engineers solving the decision identification problem, we now introduce a decision identification process. It comprises three activities:

1. For each eligible architectural pattern (e.g., patterns defining an architectural style), *review the pattern description*s and *enumerate the logical components and connectors* [1] referenced in the pattern.
2. *Apply identification rules*:
    1. Identify decision issues transcending a particular system context, e.g., business domain- and enterprise-wide ones [17][18].
    2. Identify pattern-specific issues.
    3. Identify technology-related issues.
    4. Identify decision issues dealing with products and open source assets.

    Two supporting techniques can be applied in this step: a) screen sources of architectural decision knowledge such as supplemental design artifacts (e.g., books about an architectural style such as SOA [13][14]) and b) instantiate generic meta issues to find relevant knowledge. We will describe these supporting techniques in Section 5.
3. *Add issues from activity 2 to RADM* if and only if:
    1. They are architecturally relevant (i.e., satisfy the definition of an architectural decision).
    2. They have a high potential to recur (i.e., they are not project-specific).
    3. They are not already present in the RADM.

The RADM creation activities continue until the model is rich enough to support design work on projects. No firm termination condition can be given for a technique targeting human knowledge engineers: According to our experience (e.g., see case study 3 in Chapter 9 of [24]) and assuming a codification strategy for architectural knowledge management, up to a dozen issues should be added for atomic patterns and about 20 to 30 for composite patterns. Editorial quality and technical accuracy have higher priority than quantity (“if in doubt, leave it out”) [27].

4. DECISION IDENTIFICATION RULES

Contemporary architecture design methods emphasize the need to refine and elaborate designs iteratively and incrementally. The importance of a global view is also stressed [9]. Following the same principles of stepwise refinement and separating such global view from that on individual design model elements, we introduce seven *Identification Rules (IRs)* to organize activity 2 in our decision identification method:

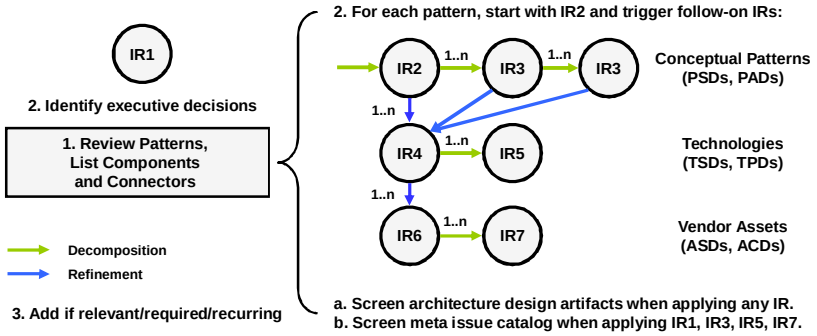

Figure 2. Identification rules in decision identification method.

IR1. *Identify pattern- and style-independent decision issues with project- or enterprise-wide scope.* We call decision issues identified with IR1 *executive decisions*, adopting a term from [16].

IR2. *For each pattern that is eligible* (output of activity 1 from the previous section), *add one issue to the RADM, deciding whether the pattern is used or not.* We call issues identified with this IR *Pattern Selection Decisions (PSDs)*. Eligible patterns can be found in patterns books, e.g., [2][5][6][12].

IR3. *Identify Pattern Adoption Decisions (PADs) in PSDs, already identified PADs, and the logical components and connectors comprising the patterns involved in these PSDs and PADs* (according to output of activity 1). Section 5 below introduces two supporting techniques for this IR.

IR4. *For each logical component and connector that is part of a pattern referenced in a PSD or PAD, add one issue concerning its implementation technology.* Such issues may present alternatives regarding integration middleware and application servers as well as application and network protocols. We call issues identified with this IR4 *Technology Selection Decisions (TSDs)*.

IR5. *Identify Technology Profiling Decisions (TPDs) in TSDs*, supported by the techniques presented in Section 5 below.

IR6. *For each technology appearing in a TSD, add one issue deciding which vendor asset is used to provide the technology.* Commercial, open source, and company-internal assets provide alternatives. We call issues identified with this IR6 *Asset Selection Decisions (ASDs)*.

IR7. *Identify Asset Configuration Decisions (ACDs) in ASDs*, supported by the techniques presented in Section 5 below.

Figure 2 illustrates the activities from Section 3 and the relations between the seven IRs. The IRs are grouped into *executive decisions* (IR1), *conceptual patterns* (IR2, IR3), *technologies* (IR4, IR5), and *vendor assets* (IR6, IR7). Figure 2 also introduces two types of relations between IRs: IRs in the same group have *decomposition relations*, while relations between IRs in different groups are called *refinement relations* [26].

IR1. IR1 deals with executive decisions about strategic technical directions [16] as well as requirements analysis [19]. It pertains to the scenario viewpoint in Kruchten's 4+1 model [15]. Examples of such strategic issues are platform directions (e.g., programming language, operating system, and hardware preferences) as well as strategic, cost-intensive decisions regarding network and server topologies (e.g., setup of geographically distributed data centers, standalone server versus high availability server cluster), but also decisions about the software engineering methods and tools to use (as far as these decisions concern the architect).

IR2, IR3. The need for PSDs is obvious if a pattern-centric design approach is followed. Patterns can be found in all architectural viewpoints; many existing patterns take a logical viewpoint [15]. PSDs identified with IR2 have a long lasting impact on project and solution health; many functional and non-functional decision drivers must be considered. Design concerns such as user channel diversity, process and resource integrity, integration challenges, and semantics dissonances [24] provide many of these decision drivers.

PADs then deal with selected patterns in a detailed way. Many pattern descriptions list variants; one or more variants have to be selected once a PSD has been made. For instance, the description of the "broker" pattern in [2] lists "direct communication" as a variant; hence, deciding for or against this variant is a PAD. A bullet list in the solution part of a pattern text may also indicate variability, requiring a PAD. Many pattern books supply navigable diagrams or decision trees to show how composite and atomic patterns in a pattern language relate to each other [5]. Pattern grammars are emerging as well [22]. These design options may also lead to the identification of one or more PADs.[1]

IR4, IR5. When refining a conceptual, platform-independent design based on patterns into an implementable, platform-specific one, decisions about implementation technologies must be made: TSDs identified with IR4 select certain technologies that implement the patterns selected in PSDs and adopted in PADs.

TPDs identified with IR5 follow TSDs. They specify implementation details, e.g., which version or subset of a technology standard to employ or which design alternatives permitted by a standard to pick. XML SCHEMA (XSD) CONSTRUCTS is an exemplary TPD issue recurring in SOA design: due to the large scope of the technology standard, the

[1] If two patterns have similar or identical intent, context, or forces sections, they can be combined into a single PSD. This is a modeling decision of the knowledge engineer.

Table 1. Identification rules, cardinalities, and artifacts to be screened.

Identification Rule	Cardinality (Section 4)	Artifacts to be Screened
IR1: Identify executive decisions	Apply *once* (specific for application genre)	Enterprise architecture documents, project proposals, system context diagrams, meta issues (Table 2)
IR2: Identify PSDs	Apply *once* per architectural pattern (e.g., in definition of an architectural style)	Architectural style definition, table of content, overview diagrams, and cheat sheets in pattern books
IR3: Identify PADs in PSDs and PADs	Apply *multiple times* per PSD/PAD and logical component and connector in pattern	Descriptions of architectural patterns (online, text books), pattern variants and grammars, meta issues
IR4: Identify TSDs in PSDs and PADs	Apply *once* per logical component and connector in pattern	Enterprise architecture documents, standards bodies (e.g., IEEE, ISO, W3C, OASIS)
IR5: Identify TPDs in TSDs	Apply *one or more times* per TSD	Technology standards and primers, tutorials, meta issues
IR6: Identify ASDs in TSDs	Apply *once* per technology appearing in a TSD	External parties (analyst reports), enterprise architecture documents
IR7: Identify ACDs in ASDs	Apply *one or more times per* ASD	Vendor documentation, previous projects, existing systems, meta issues

subset of the XSD language constructs used to model XML request and response messages must be decided.

Technology-level decisions are more concrete than those pertaining to pattern selection and adoption; measurable decision drivers regarding interoperability, performance (i.e., response times and throughput), and scalability apply.

IR6, IR7. ASDs and ACDs identified with IR6 and IR7 pertain to assets that provide and support the technologies selected in TSDs and profiled in TPDs. In SOA design, commercial products, open source, and company-internal assets supply the alternatives. Discrepancies between abstract concepts and implementation reality can be expressed as ACDs: Vendor products may implement a conceptual pattern in an unusual way, have limitations, or offer proprietary extensions.

5. SUPPORTING TECHNIQUES

This section specifies two techniques that enable the knowledge engineer to identify issues in the literature when applying the IRs from the previous section.

a) Screen supplemental design artifacts (all IRs). Table 1 repeats the IR cardinalities from Figure 2 and adds information about the artifacts in which architectural knowledge about the issues can be found, as well as additional follow-on issues. These artifacts may be part of the definition of an architectural style such as SOA. They may also originate from already completed projects which applied the patterns (see Appendix B of [24]).

b) Screen catalog of generic meta issues (IR1, IR3, IR5, IR7). IR2, IR4, and IR6 are straightforward to apply. However, architecture design does not stop when patterns, technologies, and vendor assets have been selected; pattern adoption, technology profiling, and vendor asset configuration issues exist as well [11][28][29]. According to our modeling experience, pattern texts, technology specifications, and vendor documentation often leave out detailed information about such issues; insight into platform-dependent architectural qualities such as performance and scalability remains tacit. For patterns, this is not the fault of the pattern author: by design, most patterns are "soft around the edges" [11] to make them broadly applicable and platform-independent. Hence, more knowledge is required to make IR1, IR3, IR5, and IR7 reproducible and scope a RADM in such a way that the issues are concrete and specific enough to be applicable during the design work on a project.

To provide such knowledge, we introduce the notion of generic *meta issues*: Meta issues are architectural decisions that recur within and across application genres, but are not specific to any architectural style, implementation technology, or vendor asset. Meta issues have to meet the qualification criteria for architectural decisions; for instance, they must pertain to the system as a whole or to its key components, and impact the quality attributes of the system [25]. However, they are more abstract and generic than RADM issues, e.g., they do not reference any particular component or connector in a pattern. Unlike patterns, they describe problems (design concerns) rather than solutions to them. Each issue references and instantiates one or more of the meta issues. To give an example: "system transactionality" is a meta issue because usage of the concept is common in many application genres. Fowler [6] instantiates the meta issue into an issue giving concrete advice for enterprise application architectures and concurrency management in application servers that support a Web-based presentation layer.

A *meta issue catalog* makes formerly tacit knowledge explicit. Table 2 presents an example of a meta issue catalog, harvested and compiled from architecting experience in the enterprise application genre since 1995 [28][29].

The meta issues in this catalog are relevant and recurring in enterprise application development and integration because they address common design concerns (i.e., user channel diversity, business process and resource integrity management, integration challenges, and semantic dissonances) [24]. Solutions to these challenges exist in pattern form; these patterns then become architecture alternatives resolving identified issues, e.g., [2][5][6][12].

The meta issue catalog merely serves as reference and input to SOAD step 1 (decision identification, performed by the knowledge engineer); it is neither self-explaining nor self-containing. To apply our technique, the knowledge engineer must be familiar with the subject matter and/or have project experience with the architectural concerns indicated by the meta issues. The referenced literature provides related background information.

Table 2. Generic meta issue catalog (independent of application domain).

IR and Artifact	Decision Topic	Meta Issues (Independent of Application Genre and Architectural Patterns)
IR1: Enterprise architecture documentation	IT strategy	Buy vs. build strategy, open source policy
	Governance	Methods (processes, notations), tools, reference architectures, coding guidelines, naming standards, asset ownership
IR1: System context	Project scope	External interfaces, incoming and outgoing calls (protocols, formats, identifiers), service level agreements, billing
IR1: Other viewpoints	Development process	Configuration management, test cases, build/test/production environment staging
IR3: Architecture overview diagram	Logical layers	Coupling and cohesion principles, functional decomposition (partitioning)
	Physical tiers	Locations, security zones, nodes, load balancing, failover, storage placement
	Data management	Data model reach (enterprise-wide?), synchronization/replication, backup strategy
IR3: Architecture overview diagram	Presentation layer	Rich vs. thin client, multi-channel design, client conversations, session management
	Domain layer (process control flow)	How to ensure process and resource integrity, business and system transactionality
	Domain layer (remote interfaces)	Remote contract design (interfaces, protocols, formats, timeout management)
	Domain layer (component-based development)	Interface contract language, parameter validation, Application Programming Interface (API) design, domain model
	Resource (data) access layer	Connection pooling, concurrency (auto commit?), information integration, caching
	Integration	Hub-and-spoke vs. direct, synchrony, message queuing, data formats, registration
IR3: Logical component	Security	Authentication, authorization, confidentiality, integrity, non-repudiation, tenancy
	Systems management	Fault, configuration, accounting, performance, and security management
IR3: Logical component	Lifecycle management	Lookup, creation, static vs. dynamic activation, instance pooling, housekeeping
	Logging	Log source and sink, protocol, format, level of detail (verbosity levels)
	Error handling	Error logging, reporting, propagation, display, analysis, recovery
IR5 and IR7: Components and connectors	Implementation technology (IR5)	Technology standard version and profile to use, deployment descriptor settings (QoS)
	Deployment (IR7)	Collocation, standalone vs. clustered
IR7: Physical node	Capacity planning	Hardware and software sizing, topologies
	Systems management	Monitoring concept, backup procedures, update management, disaster recovery

6. VALIDATION IN SOA DOMAIN

We now report how we applied our concepts to enterprise application development and Service-Oriented Architecture (SOA) design, using the IR structure from Figure 2 in Section 4.

Executive level decisions (IR1 applied). With IR1, we can identify a number of executive decisions [16]. Two of these executive decisions are: ARCHITECTURAL STYLE[2] with SOA MESSAGING as one of several alternatives, LAYERING, and LANGUAGE AND PLATFORM PREFERENCES with alternatives such as MICROSOFT .NET/C#, JEE/JAVA, and LAMPP. The TOOLING DIRECTIONS decision (e.g., OPEN SOURCE or SINGLE VENDOR) also recurs. Identified with IR1, these are one-of-a-kind issues. The identifying meta issues from Table 2 are "reference architectures" and "tools".

Two examples of decisions related to business requirements are ANALYSIS-PHASE BPM vs. USE CASE MODELS or USER STORIES as FUNCTIONAL REQUIREMENTS NOTATION and using BPMN or UML ACTIVITY DIAGRAMS as BPM NOTATION. They were identified with IR1 as well; the meta issue is "methods (processes, notations)".

Conceptual level decisions (IR2 and IR3 applied). The identification rules advised us to create one PSD per pattern (IR2) and multiple PADs per PSD (IR3). The issues in this conceptual level (see Figure 2) deal with the following topics:

- Selection and adoption of SOA patterns: service consumer-provider contract, enterprise service bus, service composition, and service registry [24][25].
- Design of abstract, non-technical part of service contract, corresponding to the WSDL 1.1 port type (interface in WSDL 2.0).
- Definition of security and service management concepts, e.g., transport- or message-layer security and business process monitoring concepts.
- Selection of transaction management patterns.

Service contract. A PAD related to the service consumer-provider contract pattern is to decide whether the IN MESSAGE GRANULARITY of the service operations should be coarse or fine in terms of the breadth and depth of the message parts (i.e., number of message parts, usage of scalar or complex data types). This decision is required for each service operation. A similar decision has to be made about the OUT MESSAGE GRANULARITY. Furthermore, a conscious decision for the OPERATION-TO-SERVICE GROUPING is also required. "API design" is the IR3 meta issue for both issues (see Table 2).

A related PAD is MESSAGE EXCHANGE PATTERN: A "service operation" appears in the SOA patterns in [24], and an IR3 meta issue called "synchrony" appears in Table 2. Combining these two knowledge sources identifies this issue: For each service operation invocation, it has to be decided how to invoke atomic services from the business activities in a process-oriented service composition layer. Synchronous REQUEST-REPLY calls and asynchronous ONE WAY messaging are two of the alternatives. INVOCATION TRANSACTIONALITY PATTERN is introduced in [24].

[2] We set issues and alternatives IN THIS FONT in this paper (SMALL CAPS).

Enterprise service bus. INTEGRATION PARADIGM is the PSD that originates from the enterprise service bus pattern [24]. The pattern text of the broker pattern in [2] supplies us with more knowledge about integration issues: "(1) define an object model. (2) decide which type of component interoperability the system should offer, binary or Interface Description Language (IDL). (3) specify the APIs the broker component provides for collaborating with clients and servers. (4) use proxy objects to hide implementation details from clients and servers. (5) design the broker component. (6) develop IDL compilers. Step (5) has nine sub steps: (5.1) on-the-wire protocol, (5.2) local broker, (5.3) direct communication variant, (5.4) (un)marshalling, (5.5) message buffers, (5.6) directory service, (5.7) name service, (5.8) dynamic method invocation, (5.9) the case in which something fails […]." All these steps qualify as PADs, following the INTEGRATION PARADIGM PSD according to IR3 (see Figure 2).

Table 3. Recurring SOA issues (instantiations of meta issues).

Identification Rule	SOA Pattern	Issue (Decision Required)
IR1: (Technical) Executive decisions, Requirements analysis decisions	n/a	ARCHITECTURAL STYLE LAYERING LANGUAGE AND PLATFORM PREFERENCES TOOLING DIRECTIONS FUNCTIONAL REQUIREMENTS NOTATION BPM NOTATION
IR2 and IR3: Pattern Selection Decisions (PSDs), Pattern Adoption Decisions (PADs)	Service consumer-provider contract	IN MESSAGE GRANULARITY OUT MESSAGE GRANULARITY OPERATION-TO-SERVICE GROUPING MESSAGE EXCHANGE PATTERN INVOCATION TRANSACTIONALITY PATTERN SERVICE PROVIDER TRANSACTIONALITY (ST)
	Enterprise service bus	INTEGRATION PARADIGM COMMUNICATIONS TRANSACTIONALITY (CT)
	Service composition	SERVICE COMPOSITION PARADIGM PROCESS LIFETIME SESSION MANAGEMENT RESOURCE PROTECTION STRATEGY PROCESS ACTIVITY TRANSACTIONALITY (PAT)
IR4 and IR5: Technology Selection Decisions (TSDs), Technology Profiling Decisions (TPDs)	Service consumer-provider contract	TRANSPORT PROTOCOL BINDING MESSAGE EXCHANGE FORMAT SOAP COMMUNICATION STYLE WEB SERVICES API JAVA SERVICE PROVIDER TYPE XML SCHEMA (XSD) CONSTRUCTS
	Enterprise service bus	INTEGRATION TECHNOLOGY TRANSPORT QOS
	Service composition	WORKFLOW LANGUAGE BPEL VERSION COMPENSATION TECHNOLOGY
IR6 and IR7: Vendor Asset Selection Decisions (ASDs), Vendor Asset Configuration Decisions (ACDs)	Service consumer-provider contract	SOAP ENGINE
	Enterprise service bus	ESB PRODUCT ESB TOPOLOGY (IBM DATAPOWER CONFIGURATION)
	Service composition	BPEL ENGINE INVOKE ACTIVITY TRANSACTIONALITY

Service composition. SERVICE COMPOSITION PARADIGM with alternatives such as WORKFLOW and OBJECT-ORIENTED PROGRAMMING is a recurring issue. Moreover, a PROCESS LIFETIME issue has to be decided for any executable process, with alternatives such as long running MACROFLOW and short running MICROFLOW. This is a conceptual abstraction of an engine-specific design issue not handled by the BPEL specification. The SESSION MANAGEMENT approach also has to be decided in this context.

"System transactionality" was one of the meta issues listed in Table 2. A RADM for SOA contains several issues dealing with this concern, created with IR3. For instance, it has to be agreed which RESOURCE PROTECTION STRATEGY should be taken, e.g., SYSTEM TRANSACTIONS or BUSINESS COMPENSATION (or a combination thereof) [6].

Technology level decisions (IR4 and IR5 applied). The identification rules instructed us to add one TSD per conceptual pattern in the RADM for SOA (IR4) and to add multiple TPDs per TSD (IR5). The issues deal with topics such as:

- Selection of technologies implementing the SOA patterns and profiling of standards defining these technologies.
- Design of the technical part of the service contract (WSDL binding), and decisions about WS-* standards such as SOAP, BPEL, and UDDI.
- Selection of protocols, algorithms, and data formats for security, e.g., authentication, authorization, and encryption with Transport Layer Security and/or WS-Security as well as service management, e.g., monitoring protocols/formats.
- Technology refinement of transaction management patterns, e.g., the decision to use WS-AtomicTransaction.

Service contract. For each service invocation, the following TSDs must be made: Which TRANSPORT PROTOCOL BINDING should be used to invoke atomic services from the processes in the service composition layer, e.g., HYPERTEXT TRANSFER PROTOCOL (HTTP) or JAVA MESSAGING SERVICE (JMS)? Which MESSAGE EXCHANGE FORMAT structures request and response messages in an interoperable manner, e.g., SOAP or JAVASCRIPT OBJECT NOTATION (JSON)?

SOAP COMMUNICATION STYLE with alternatives DOCUMENT/LITERAL or RPC/ENCODED is a related TPD, assuming that SOAP was decided for as MESSAGE EXCHANGE FORMAT. The WEB SERVICES API and JAVA SERVICE PROVIDER TYPE have to be decided per service consumer and service provider; JAX-RPC vs. JAX-WS and ENTERPRISE JAVABEAN (EJB) vs. PLAIN OLD JAVA OBJECT (POJO) are Java alternatives. This issue and its alternatives are identified with IR4 (see Table 1). Moreover, the subset of XML SCHEMA (XSD) CONSTRUCTS used to define

message parts in WSDL contracts and SOAP messages must be decided. These issues are identified with IR5, following the IR3-related meta issues about integration and component-based development; the meta issue is "API design" (Table 2).

Enterprise service bus. A TSD following the PSD about an INTEGRATION PARADIGM is to decide for an INTEGRATION TECHNOLOGY such as WS-* WEB SERVICES or RESTFUL INTEGRATION. It is identified with IR4. TRANSPORT QOS is a related TPD identified with IR5. See [24] for more information.

Service composition. A TSD that is required for each process is the choice of WORKFLOW LANGUAGE, e.g., BUSINESS PROCESS EXECUTION LANGUAGE (BPEL). Some TPDs follow the TSD to use BPEL: Which BPEL VERSION and which COMPENSATION TECHNOLOGY to use? The BPEL standards introduce these issues and their alternatives.

Vendor asset level decisions (IR6 and IR7 applied). ASDs are required for all alternatives of TSDs (IR6); ACDs follow ASDs (IR7). With support from the IR7 meta issues in Table 2, we can identify issues about the following topics:

- Issues pertaining to assets that implement the Web services standards, for instance, WSDL editors, SOAP engines, BPEL engines, and UDDI registries.[3]
- Design of the part of the service contract related to deployment, which corresponds to the service and port elements in WSDL 1.1.
- Configuration of the selected products to reflect the technology profiling choices made, including selection and customization of proprietary APIs.

Integration ASDs are the selection of a SOAP ENGINE, of an ESB PRODUCT, and of a BPEL ENGINE. For instance, the IBM DATAPOWER appliance is an XML processing hardware which implements several of the WS-Security specifications and can act as an ESB. ESB TOPOLOGY (IBM DATAPOWER CONFIGURATION) is a related ACD. The BPEL ENGINE decision has many vendor and open source alternatives, including, but not limited to IBM WEBSPHERE PROCESS SERVER, and ORACLE BPEL PROCESS MANAGER. SOAP ENGINE has alternatives such as APACHE AXIS2.

Table 3 summarizes the RADM for SOA issues we introduced in this section. In summary, applying the seven IRs and the meta issue catalog to SOA patterns such as service consumer-provider contract, enterprise service bus, service composition, and service registry, as well as general architectural patterns such as broker, yields an initial RADM for SOA. We identified 35 recurring issues in this paper (all set in SMALL CAPS FONT); our full RADM for SOA contains more than 500 such recurring issues [24][26]. The resulting RADM has been successfully validated and even used on commercial projects. We reported about the user feedback in our previous publications [24][27].

7. DISCUSSION AND CONCLUSIONS

This paper introduced the seven steps in SOA Decision Modeling (SOAD) and elaborated on SOAD step 1, which deals with the scoping of a reusable architectural decision model (which can serve as a design guidance model). To support this first step, the paper introduced and demonstrated a top-down process combining identification rules and a meta issue catalog to define the boundaries of a reusable architectural decision model.

As we could observe in one of the case studies in [24], the presented decision identification method increases the productivity of the knowledge engineer significantly.

Our decision identification approach is pattern-centric: architectural patterns serve as anchor points for the scoping of a reusable architectural decision model. Leveraging knowledge already captured in pattern form is a key advantage of SOAD; it saves the knowledge engineer a significant amount of documentation effort. The issue names in a reusable architectural decision model create a language for a problem domain, just like pattern names create one for a solution domain.

A key assumption of SOAD is that many of the architectural decisions required during design (also called issues in this paper) actually recur. The feedback obtained during the validating industry case studies indicates that this assumption is rather strong, but valid [24][27].

We do not claim the meta issue catalog to be complete;, it is possible to add, update, and delete meta issues in the catalog as needed (e.g., during tailoring). For instance, the following sources of input can be taken into account:

- Other architectural pattern languages such as those in [21][23]; the problem descriptions in intent, context, forces, and consequences sections of pattern texts are particularly knowledge-intensive and provide rich input to the knowledge engineer.
- Architectural tactics as defined in the software architecture literature [1].
- Design challenges explained in genre-specific literature, e.g., tutorials, handbooks, and industry reference models for business process management and enterprise application integration.
- Some of the SOA literature also presents style-agnostic knowledge [13][14].

All 35 SOA issues identified in Section 6 dealt with a logical viewpoint. However, many issues relate to a physical viewpoint. For example, several decisions are required to create a conceptual *operational model*, e.g., about clustering or a certain network topology. Follow on decisions are required to refine such operational model on the technology level, for instance selecting a certain data replication mechanism supporting backup or failover concepts specified on the conceptual level. Even more detailed decisions are required to create a vendor-specific operational model, e.g., concerning the proprietary system management scripts required to deploy the selected backup or failover technology, the installation of heartbeat and takeover protocols, and the configuration of servers and network equipment (e.g., firewalls). Further details regarding decisions about the physical viewpoint are out of scope of this paper.

The presented top-down identification process must be complemented with a bottom up knowledge harvesting method to ensure continuous content contributions from industry projects. This method must provide a repeatable process, criteria whether a decision qualifies for inclusion in a reusable architectural

[3] Many of these decisions may be made as executive decisions in practice, e.g., if strategic partnerships with certain vendors or a single vendor policy have been established. This is often the case for middleware such as application servers or databases, with justifications such as direct and indirect costs (e.g., licenses, training, and systems management).

decision model, and decision modeling guidance. Such process, criteria, and guidance are described in [24] and [27].

The issue catalog produced in step 1 of SOAD (described in this paper) does not give any advice how to document and use the issues; in this paper, we have only named them and touched upon alternatives and dependencies in anecdotal form. In our previous publications, we presented how to model, structure, order, and use issues once they have been identified [26][27]. It is not mandatory to perform all these steps; the issues identified in step 1 are already suited as review checklists, may serve as input to design workshops, and are able to supplement other architecture design methods such as attribute-driven design [1]. The additional usage scenarios and validation results described in [24] support this statement.

Our identification rules and meta issues leave many modeling choices to the knowledge engineer; this is deliberate. It is possible to combine or remove issues, e.g., if a pattern itself already resolves a meta issue, or the related knowledge cannot be made reusable (model tailoring).

We propose a human-centric approach to decision identification, rather than an algorithm than can be fully implemented in a tool. We considers this to be adequate given the current state of the art and the practice. For further automation, it would be required to capture expert knowledge in machine-readable form and apply data mining techniques. This appears to be overly ambitious, requiring strong assumptions regarding the formalization of input models and a highly stable application genre.

Despite its name, SOAD is *not* a SOA domain-specific solution, but provides a general-purpose decision modeling framework. The SOAD concepts were originally introduced in [24] in 2009; since then, we have applied and validated them successfully in non-SOA domains, including cloud computing and strategic outsourcing [27].

8. REFERENCES

[1] Bass, L., Clements, P., Kazman, R., Software Architecture in Practice, Second Edition. Addison Wesley, 2003.

[2] Buschmann F., Meunier R., Rohnert H., Sommerlad P., and Stal M., Pattern-Oriented Software Architecture – a System of Patterns. Wiley, 1996.

[3] de Boer R.C., Farenhorst, R., Lago P., van Vliet H., Clerc V., and Jansen A. Architectural Knowledge: Getting to the Core. Proceedings of QoSA 2007, Springer LNCS Volume 4880/2008. Pages 197-214.

[4] Duenas, J. C., Capilla R., The Decision View of Software Architecture. Proceedings of 2nd European Workshop on Software Architecture (EWSA), Springer LNCS Volume 3527/2005, Pages 222-230.

[5] Evans E., Domain-Driven Design. Tackling Complexity in the Heart of Software. Addison Wesley, 2003.

[6] Fowler M., Patterns of Enterprise Application Architecture. Addison Wesley, 2003.

[7] Fowler M., Writing Software Patterns. Available online:, http://www.martinfowler.com/articles/writingPatterns.html

[8] Harrison N., Avgeriou P., and Zdun U.. Using Patterns to Capture Architectural Decisions. IEEE Software, IEEE Computer Society 2007. Pages 38-45.

[9] Hofmeister C., Kruchten P., Nord, Obbink J. H., Ran A., America P., A General Model of Software Architecture Design Derived from Five Industrial Approaches. Journal of Systems and Software 80(1), Elsevier, 2007. Pages 106-126.

[10] Hofmeister C., Nord R., Soni D., Applied Software Architecture. Addison Wesley, 2000.

[11] Hohpe G., SOA Patterns: New Insights or Recycled Knowledge? Key note at Fifth International Workshop on SOA and Web Services Best Practices (at OOPSLA), Montreal, Canada, October 21, 2007.

[12] Hohpe G., Woolf, B., Enterprise Integration Patterns. Addison Wesley, 2004.

[13] Josuttis N., SOA in Practice – The Art of Distributed Systems Design. O'Reilly, 2007.

[14] Krafzig D., Banke K., Slama D., Enterprise SOA, Prentice Hall, 2005

[15] Kruchten P., The 4+1 View Model of Architecture, IEEE Software, Volume 12, Number 6, November 1995. Pages 42-50.

[16] Kruchten P., Lago P., van Vliet H., Building up and Reasoning about Architectural Knowledge. Proceedings of QoSA 2006, LNCS 4214, Springer 2006. Pages 43-58.

[17] Malan R., Bredemeyer D., Less is More with Minimalist Architecture. IT Pro, IEEE Computer Society, October 2002.

[18] Pulkkinen, M., Systemic Management of Architectural Decisions in Enterprise Architecture Planning. Four Dimensions and Three Abstraction Levels. Proceedings of the 39th Annual Hawaii International Conference on System Sciences, Volume 08. IEEE Computer Society, Washington, DC, 2006. Page 179.1.

[19] Sommerville I., Software Engineering, Fifth Edition. Addison Wes-ley, 1995.

[20] Tang, A., Ali Babar, M. A., Gorton, I., and Han, J. 2005. A Survey of the Use and Documentation of Architecture Design Rationale. Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture. IEEE Computer Society, 2005. Pages 89-98.

[21] Völter M., Kircher M., and Zdun U., Remoting Patterns – Foundations of Enterprise, Internet, and Realtime Distributed Object Mid-dleware. Wiley, 2004.

[22] Zdun U., Systematic Pattern Selection using Pattern Language Grammars and Design Space Analysis. Software: Practice & Experience, 2007.

[23] Zdun U., Hentrich C., van der Aalst, W., A Survey of Patterns for Service-oriented Architectures. International Journal of Internet Protocol Technology, 1(3), Inderscience Enterprises, 2006. Pages 132–143.

[24] Zimmermann O., An Architectural Decision Modeling Framework for Service-Oriented Architecture Design. Ph. D. thesis, Stuttgart University, 2009.

[25] Zimmermann O., Architectural Decisions as Reusable Design Assets. IEEE Software, vol. 28, no. 1, Jan./Feb. 2011. Pages 64-69.

[26] Zimmermann O., et al., Managing Architectural Decision Models with Dependency Relations, Integrity Constraints, and Production Rules, J. Systems and Software and Services, vol. 82, no. 8, 2009. Pages 1246–1267.

[27] Zimmermann O., Miksovic C., Küster J.M., Reference Architecture, Metamodel, and Modeling Principles for Architectural Knowledge Management in Information Technology Services, J. Systems and Software and Services, vol. 85, no. 9, 2012. Pages 2014–2033.

[28] Zimmermann O., Doubrovski V., Grundler J., Hogg K., Service-Oriented Architecture and Business Process Management in an Order Management Scenario: Rationale, Concepts, Lessons Learned. Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05). ACM, 2005. Pages 301-312.

[29] Zimmermann O., Milinski S., Craes S., Oellermann F., Second Generation Web Services-Oriented Architecture in Production in the Finance Industry, Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '04). ACM, 2004. Pages 283-289.

[30] Zimmermann O., Zdun U., Gschwind T., Leymann F., Combining Pattern Languages and Architectural Decision Models into a Comprehensive and Comprehensible Design Method. Proceedings of IEEE WICSA 2008, IEEE Computer Society, 2008. Pages 157-166.