From Aspect-oriented Requirements Models to Aspect-oriented Business Process Design Models

Sébastien Mosser, Gunter Mussbacher, Mireille Blay-Fornarino, Daniel Amyot

▶ **To cite this version:**

Sébastien Mosser, Gunter Mussbacher, Mireille Blay-Fornarino, Daniel Amyot. From Aspect-oriented Requirements Models to Aspect-oriented Business Process Design Models. 10th international conference on Aspect Oriented Software Development (AOSD'11), Mar 2011, Porto de Galinhas, Brazil. pp.1-12. hal-00531027v1

HAL Id: hal-00531027

https://hal.science/hal-00531027v1

Submitted on 1 Nov 2010 (v1), last revised 12 Jan 2011 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

From Aspect-oriented Requirements Models to Aspect-oriented Business Process Design Models

An Iterative and Concern–Driven Approach for Software Engineering

Sébastien Mosser
University of Nice
CNRS – I3S Lab (UMR 6070)
Sophia Antipolis, France
mosser@polytech.unice.fr

Gunter Mussbacher
University of Ottawa
SITE
Ottawa, Canada
gunterm@site.uottawa.ca

Mireille Blay–Fornarino
University of Nice
CNRS – I3S Lab (UMR 6070)
Sophia Antipolis, France
blay@polytech.unice.fr

Daniel Aymot
University of Ottawa
SITE
Ottawa, Canada
daymot@site.uottawa.ca

ABSTRACT

Aspect-oriented approaches are available for various phases of software development such as requirements, design, and implementation. Yet, moving from one phase to the next with aspects remains a challenge seldom studied. In this paper, we present an iterative, concern-driven software engineering approach that is based on a tool-supported, semi-automatic transformation of scenario-based, aspect-oriented requirements models into aspect-oriented business process design models. This approach is realized by a mapping from Aspect-oriented Use Case Maps (AoUCM) to Adore business process models, allowing for the continued encapsulation of requirements-level concerns in design-level artifacts. Problems detected during the design phase can be rectified in the requirements models via several feedback loops that support iterative model development. We discuss the transformation process and illustrate, as proof-of-concept, our contribution on the PicWeb case study, a SOA–based implementation of business processes for pictures management.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements / Specification—*languages, methodologies, tools*; D.2.11 [**Software Architectures**]: Service–Oriented Architecture (SOA)—*workflow, multi–paradigm modeling*

General Terms

Design, Management, Experimentation, Languages.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AOSD '11 Porto de Galinhas, Pernambuco, Brazil
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$10.00.

Keywords

Service–Oriented Architecture (SOA), Transformation, Aspect-oriented Requirements, Aspect-oriented User Requirements Notation (AoURN), Adore, Aspect-oriented Design, Business Processes

1. INTRODUCTION

Over the last decade, many aspect-oriented software development (AOSD) techniques [3] have been proposed. Less attention has been paid to automatically linking together these approaches across life-cycle phases, even though AOSD claims improved productivity when crosscutting concerns are encapsulated across these phases. We investigate if the same concerns can be kept at the different levels of abstraction for requirements and design models in an AOSD process in the context of Service–Oriented Architectures (SOA).

During the last few years, SOA has been seen as a major philosophy and solution for the organization of IT systems to manage complexity and foster agility and innovation. Analysts [19] foresee a clear competitive advantage in adopting architecture-based solutions for IT products. SOA promotes a vision where functionality is grouped around business processes and packaged as interoperable services [10]. The development of business processes involves *(i)* the definition of business workflows while *(ii)* taking into account non-functional requirements such as conformity to strict legal constraints in terms of data privacy and security.

To address key non-functional requirements, the business process designer has to design workflows in their contextual environment, thus spending time on non-business oriented tasks. Several adaptations may occur on the same workflow due to the many different concerns such as cost, efficiency, and security. Furthermore, the end user has to ensure that the workflows conform to the initial and adapted requirements. Based on these observations, there is a need for helping users move from requirements to the design of accurate workflows while taking into account several possibly crosscutting concerns. Changes at requirements or design levels must be supported to apply adaptations in a safe way.

Clearly, tools are needed to ease the derivation of business

processes from requirements and ensure traceability between these two levels. Moreover, separation of concerns is needed at all levels of the software development process to tackle the complexity of defining business workflows. We propose an aspect-oriented approach encompassing general purpose requirements models and business process design models as a first step towards a larger, iterative, end-to-end concern–driven software engineering process that ensures the encapsulation of (crosscutting) concerns through all software development phases.

We implement this vision with a tool-supported, automatic transformation of aspect-oriented requirements models expressed with the Aspect-oriented User Requirements Notation (AoURN) [15] into aspect-oriented business process models expressed with ADORE [11]. Sect. 2 introduces AoURN, ADORE, and the PicWeb case study which will be used to demonstrate the feasibility of our proposed approach throughout this paper. Sects. 3 and 4 discuss the transformation process and various feedback loops that support our iterative, concern–driven approach. While Sect. 3 focuses on requirements models with little or no data-related information, Sect. 4 investigates the benefits of introducing more detailed data-related information in requirements models to strengthen the iterations between requirements and design models. We continue with a discussion of related work in Sect. 5 and our conclusions and future work in Sect. 6.

2. THE PICWEB CASE STUDY

PicWeb is a subpart of a larger legacy system called JSEDUITE [14]. JSEDUITE is a SOA–based information system designed to fit academic institution needs. It supports information broadcasting from academic partners (*e.g.,* transport network, school restaurant) to several devices (*e.g.,* user's smart-phone, PDA, desktop, public screen). This system is built upon a SOA and used as a validation platform by the FAROS project[1]. The implementation follows SOA methodological guidelines [22], positioning it as a typical usage of a SOA for experimental purposes. The system has been deployed and is daily used in two institutions. Further information about implementation and partners can be found on the project web site[2].

Inspired by Web 2.0 *folksonomies*, the JSEDUITE system relies on community–driven partners such as FLICKR (Yahoo!) or PICASA (Google) to store and then retrieve available pictures. PicWeb is a restricted version of JSEDUITE, only dealing with pictures management. It allows a set of pictures with a given tag and up to a given threshold number to be retrieved from existing partner services.

2.1 PicWeb AoURN Requirements Model

The Aspect-oriented User Requirements Notation (AoURN) [7, 15, 16] supports the elicitation, analysis, specification, and validation of requirements in an aspect-oriented modeling framework for early requirements with the help of goal and scenario models. AoURN's scenario models are defined with Aspect-oriented Use Case Maps (AoUCM) and consist of a path that begins at a *start point* (●, *e.g.,* get pictures in FIG. 1) and ends with an *end point* (▌, *e.g.,* displayed). A path may contain *responsibilities* (✕, *e.g.,* getPicturesWithTag), identifying the steps in a scenario, and notational symbols for alternative (⌐) and concurrent (⊣⊢) branches. Path elements may be assigned to a *component* (▭, *e.g.,* PicWeb). *Actor components* (웃, *e.g.,* User) are special kinds of components that are used to model entities that are interacting with the system. *Stubs* (◇, *e.g.,* truncate) are containers for sub-models called *plug-in maps*, thus allowing for hierarchical structuring of AoUCM models.

[1]http://www.lifl.fr/faros
[2]http://www.jseduite.org

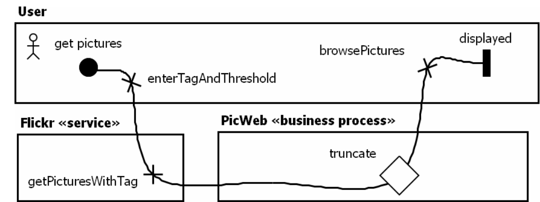


Figure 1: AoUCM: Root Map – Get Pictures

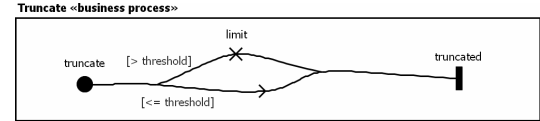


Figure 2: AoUCM: Plug-in Map – Truncate

For example in FIGS. 1 and 2, the Get Pictures scenario for the PicWeb application starts with the User entering a tag and threshold number. The application then uses Flickr to retrieve pictures with this tag and the Truncate business process limits the pictures to a maximum threshold number if the threshold is exceeded. The scenario concludes with the User browsing the pictures.

AoURN allows the definition of name/value pairs called *metadata* (*e.g.,* <<service>>) for any AoURN modeling element. In connection with OCL constraints which can be defined and verified for AoURN models by jUCMNav [21], the most comprehensive AoURN tool available to date, metadata thus provides an extension mechanism that permits the definition of profiles for AoURN. For more details about URN and AoURN, visit the URN VIRTUAL LIBRARY [1].

Crosscutting Concerns in AoURN.

AoURN introduces concerns as first-class modeling elements, regardless of whether they are crosscutting or not. Typical concerns in the context of AoURN are stakeholders' intentions, non-functional requirements, and use cases. AoURN groups together all relevant properties of a concern such as goals, behavior, and structure, as well as pointcut expressions needed to apply new goal and scenario elements or to modify existing elements in the AoURN model.

The AoUCM models in FIGS. 1 and 2 are standard UCM models for the Get Pictures and Truncate concerns which together belong to the PicWeb concern. For the PicWeb application, the Caching, Randomizer service, Payment service, and Picasa service (FIG. 3) concerns are applied to the PicWeb concern with the help of aspect-oriented techniques.

The aspectual properties for the Picasa concern are shown on an *aspect map* (top of FIG. 3). In parallel to the Flickr service, pictures are retrieved from the Picasa service and then merged with Flickr's pictures. On a separate map (bottom of FIG. 3), the *pointcut map* defines where the Picasa concern is to be applied, allowing the pointcut expression and the as-

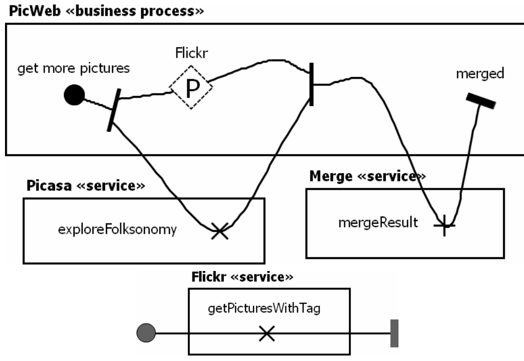


Figure 3: AoUCM: Aspect Map – Picasa Service

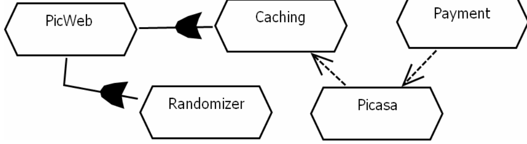


Figure 4: Concern Interaction Graph

pectual properties to be individually reused. Grey start/end points on the pointcut map are not part of the pointcut expression but rather denote its beginning/end. The pointcut of the Picasa concern therefore matches against the getPicturesWithTag responsibility of the Flickr component.

A *pointcut stub* (P) links the aspect map with the pointcut map. The causal relationship of the pointcut stub and the aspectual properties visually defines the composition rule for the aspect, indicating how the aspect is inserted in the base model (*e.g.,* before, after, optionally, in parallel or anything else that can be expressed with the UCM notation). In Fig. 3, the usage of the AND-fork and AND-join around the pointcut stub indicates parallel composition of the Flickr and Picasa services. As AoURN uses standard URN diagrams to describe pointcut expressions and composition rules, it is only limited by the expressive power of URN itself as opposed to a particular composition language. For more information about AoURN's matching and composition algorithms, the interested reader is referred to [15, 16].

Finally, aspects may depend on or conflict with each other. AoURN models dependencies and conflicts among concerns and their resolutions with the *Concern Interaction Graph* (CIG) [16]. Precedence rules can resolve many aspect interactions. The CIG is a specialized goal model that defines such precedence rules which then govern the order in which concerns are applied to an AoURN model.

The CIG for the PicWeb application in Fig. 4 states that the Randomizer and Caching concerns depend (—■—) on the PicWeb concern. This is because the pointcut expressions of the Randomizer and Caching concerns match against elements in the PicWeb concern. Furthermore, the Picasa, Caching, and Payment concerns conflict (⋯⋯→) with each other because of a shared join point. The direction of the arrows between the conflicting concerns indicates the precedence rule that resolves the conflict, *i.e.*, Caching is applied before Picasa which is applied before Payment to the AoURN model.

As AoURN goal models are out of scope for Adore, this paper focuses on AoUCM [17] scenario models and the CIG which contain all relevant information for the transformation of AoURN to Adore models.

Benefits of AoURN.

With the help of AoURN, each concern of the PicWeb application can be modeled from two points of view. AoURN goal models describe the reasons for including a concern in the application and the impact of the concern on high-level goals of all stakeholders. AoUCM, on the other hand, describes the scenarios of the concern as well as the actors and structural system entities that work together to realize the scenarios. While the goal models are not used for the transformation into Adore models, they provide an often neglected but important view for establishing traceability of system features to stakeholder intentions. AoUCM models abstracts from data and message details while describing workflow (*i.e.*, the causal relationships of system functionality) and the required abilities of structural system entities. This high level of abstraction is very appropriate for early requirements models where such details are not yet know.

Finally, goal analysis techniques allow for trade-off analysis, answering questions such as why a certain concern was chosen over another concern and which alternatives were considered. The analysis techniques for AoUCM models, on the other hand, ensure that scenarios can be regression–tested at a high level of abstraction based on the definition of pre- and post-conditions. This analysis builds on AoURN's simple, global data model which is used to formalize conditions for choice points in the AoUCM model. For example, the conditions of the two branches of the OR-fork in Fig. 2 require the definition of two Integer variables nrPhotos and threshold. The top branch's condition is nrPhotos > threshold, while the lower branch's condition is nrPhotos <= threshold.

2.2 PicWeb ADORE Business Processes

The Adore meta–model is defined as *"A meta–moDel suppOrting oRchestration Evolution"*. Using Adore, one can model complete business processes as an *orchestration* of services. Using the same formalism, an incomplete process can also be modeled, called a process *fragment*. Adore supports the integration of fragments into processes through the usage of several *composition algorithms* [11].

Adore's expressiveness is inspired by the Bpel[3]. The different types of activities that can be defined in Adore include *(i)* service invocation (denoted by `invoke`), *(ii)* variable assignment (`assign`), *(iii)* fault reporting (`throw`), *(iv)* message reception (`receive`), *(v)* response sending (`reply`), and *(vi)* the null activity, which is used for synchronization purposes (`nop`). In an Adore process model, each process starts with a `receive` activity and ends with `reply` or `throw` activities. Consequently, the Adore meta–model contains all the atomic activities defined in the Bpel normative document except the `wait` (stopwatch activity) and the `rethrow` (assimilated as a simple `throw`) activities.

As the Adore meta–model does not define composite activities, Bpel composite constructions are reified using different relations available in the meta–model. For example, a *sequence* of activities is defined by a `waitFor` relation and *if/then/else* flows are modeled using `guard` relations. Unlike Bpel, which uses composite activities to implement loops,

[3]The Bpel is defined as *"a model and a grammar for describing the behavior of a business process based on interactions between the process and its partners"* [20].

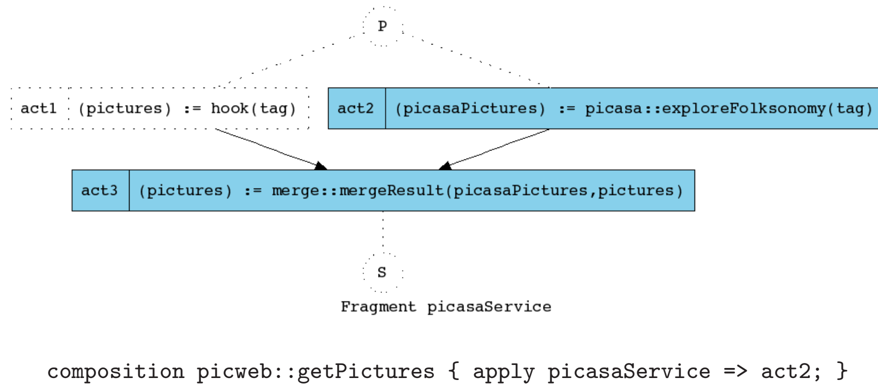


Figure 5: Adore: `picasaService` Fragment & Associated Composition Directive

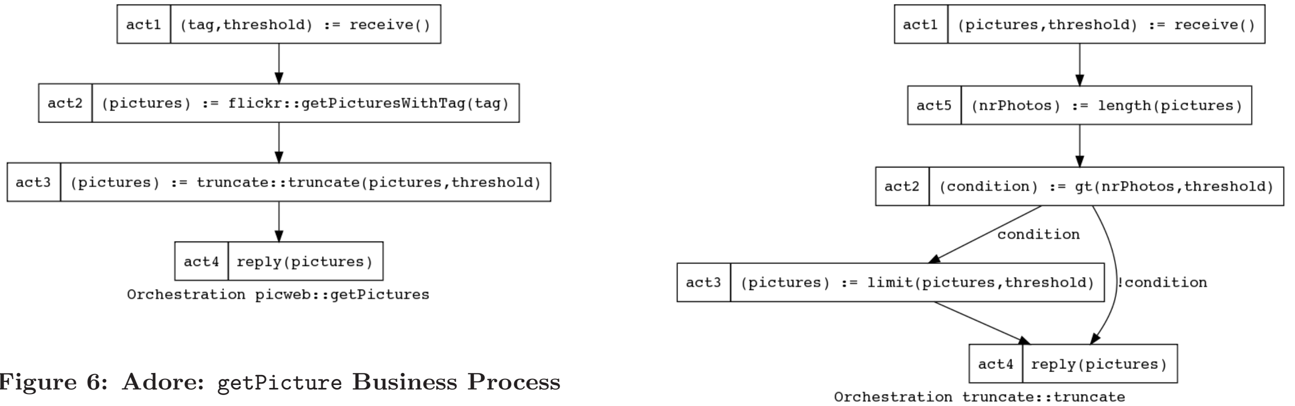


Figure 6: Adore: `getPicture` Business Process

Figure 7: Adore: `truncate` Business Process

ADORE uses *iteration policies*. As loop handling in ADORE is out of the scope of this paper, the interested reader can find a full description of it in our previously published work [12]. A more complete description of the ADORE modeling language can also be found on the project web site[4].

Relations to Aspect-oriented Programming (AOP).

According to the ERCIM working group on software evolution [5], *aspect–oriented* approaches rely at a syntactic level on four elementary notions: *(i) join points*, *(ii)*, *pointcuts (iii)*, *advice* and finally *(iv) aspects*. *Join points* represent the set of well-defined places in the program where additional behavior can be added. In the context of ADORE, we use activities for this notion. *Pointcuts* are usually defined as a set of join points. In ADORE, one can identify sets of activities as pointcuts using explicit declarations (*e.g.*, use $\{a_0, a_1\}$ activities as pointcuts) or computed declarations (*e.g.*, all activities calling the service *srv*). *Advice* describes the additional business logic to be added in the initial system. ADORE represents systems as a set of business processes. We reify *advices* in an endogenous way as business processes called *fragments*. Finally, *aspects* are defined as a set of pointcuts and advices. ADORE uses *composition directives* to bind fragments to sets of activities.

Modeling PicWeb with ADORE.

Using ADORE, the PicWeb system is realized using both orchestrations and fragments. On the one hand, main processes such as getPictures are designed as an orchestration of services (FIG. 6). In this process, the system receives a tag and a threshold in activity act_1. The tag is then passed to the flickr partner through a service invocation (act_2). The retrieved set of pictures is truncated according to the user–given threshold (act_3), and then replied to the user (act_4). The truncate process is depicted in FIG. 7. It illustrates the graphical syntax associated with *guard* relations. In this process, the activity which effectively truncates the set of pictures (act_3) is called only if the boolean condition is true (*i.e.*, the number of retrieved pictures nrPhotos is greater than the user given threshold).

On the other hand, process extensions such as picasaService are realized as a fragment (FIG. 5). In parallel to the *normal* behavior (hook activity, act_1), the PICASA service is invoked (act_2), and the retrieved pictures are merged with the legacy ones using a dedicated service (act_3). The integration of this fragment into the previously defined orchestration to compute the final behavior (o') is done through the application of a *weave* function (ω): $\omega(picasaService, getPictures) \leadsto o'$.

Benefits of ADORE.

ADORE's strengths rely on the two following principles: *(i)* conflict detection rules and *(ii)* shared join point han-

[4] `http://www.adore-design.org`

dling. The ADORE formalism is built upon first–order logic, and consistency rules (*e.g.*, no concurrent access, no dead path) are defined using the expressiveness of logical formulas. The underlying logical engine executes such predicates and identifies model inconsistencies through predicate satisfaction. ADORE's shared join point [18] handling philosophy does not rely on a–priori aspect ordering like existing AOP methods or frameworks. We defend a default merge function $\mu : Fragments^{\star} \rightarrow Fragment$ which merges fragments applied on a shared join point [13]. The designer is informed of such a decision, and can choose to keep the merged artifact or use fragment weaving to order aspects following usual AOP mechanisms. ADORE's orchestrations may then be transformed into standard BPEL processes[5].

3. TOWARDS AN ITERATIVE PROCESS

The goal of this paper is to link an aspect-oriented requirement approach (AoURN) with an aspect-oriented design approach (ADORE). Inspired by *agile* methodologies, we consider that the design phase impacts the requirements. As a consequence, we defend an iterative process were both requirements and design artifacts interact together to build the final system. We illustrate this vision in FIG. 8.

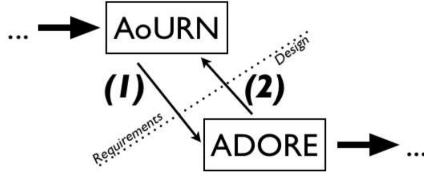


Figure 8: An Incremental ("Agile") Process

Such an iterative process assumes the sharing of information between the two layers. We denote information projected from the requirements model into the design model ((1) in FIG. 8) using the $[R \rightarrow D]$ notation. Information fed back from the design model into the requirements (2) is denoted using the $[D \rightarrow R]$ notation. As we defend an aspect-oriented approach at both levels, concerns discovered and modeled during the requirements phase can be traced more effectively to the design model regardless of whether the concerns are crosscutting or not.

3.1 Generating Designs from Requirements

The first step in the process is to generate design artifacts based on the structural and behavioral information available in the requirements model. An automated transformation of requirements models into design models reduces (or even eliminates) the amount of work done in the requirements phase that needs to be redone in the design phase while at the same time improving the consistency between the two layers.

⇒ $[R \rightarrow D]_1$ Design artifacts are generated from requirements (*e.g.*, services, concerns), improving the consistency between the two layers and allowing requirements engineers and designers to use their own formalisms.

[5]This transformation needs to introduce technical details such as data structure descriptions of service URLs.

Changes at the requirements level are validated at this level, before being propagated to the design level. Moreover, due to the automatic, correctness-preserving transformation, we ensure that the generated design model conforms to these requirements. Hence, software engineers can take advantage of the analysis capabilities of requirements notations. The feedback from the analyses serves to improve both the requirements and the generated design models.

⇒ $[R \rightarrow D]_2$ Analyses realized at the requirements level ensure properties in the design model (*e.g.*, business rules may be expressed in the requirements formalism, and requirements models may be checked against these rules to prevent *business–driven* inconsistencies to be projected into the design layer).

Implementation.

We implement such a generative approach for AoUCM and ADORE models. For a given requirements model r, we define a model transformation τ to produce the associated design model $d = \tau(r)$.

A consistent transformation of AoUCM scenario models into ADORE process models requires a mapping to be defined from the AoUCM meta–model to the ADORE meta–model. Both, AoUCM and ADORE, have rather straightforward correspondences between their meta–models and their concrete syntaxes. Therefore and for understandability, the mapping (TAB. 1) in this paper will focus on the concrete syntax views while giving only brief pointers to those readers interested in the meta–model of AoURN [7, 17] and ADORE (see website). A more complete description of the transformation process and the associated algorithm is available in the appendix.

AoUCM	Adore
Start Point (●) [StartPoint]	orchestration ⇝ *receive*, fragment ⇝ *predecessors*
End Point (▮) [EndPoint]	*orchestration* ⇝ *reply*, fragment ⇝ *successors*
Responsibility (✕) [Responsibility, RespRef]	<<business process>> ⇝ *assign*, <<service>> ⇝ *invoke*
OR-fork including its outgoing branches ( ) [OrFork, NodeConnection]	boolean *assign* activity, *guard* relations
OR-join including its incoming branches ( ) [OrJoin, NodeConnection]	exclusive *waitFor* relations entering an activity
AND-fork including its outgoing branches ( ) [AndFork, NodeConnection]	*waitFor* relations exiting an activity
AND-join including its incoming branches ( ) [AndJoin, NodeConnection]	non–exclusive *waitFor* relations entering an activity
Static Stub (◇) [Stub]	see the appendix
Pointcut Stub ( ) [Stub]	*hook* activity
Sequence (if not covered earlier) [NodeConnection]	*waitFor* relation between two activities

Table 1: Transformation of Path Elements

Constraints. It is to be expected that not all AoUCM concepts can be easily mapped to ADORE concepts and vice

versa, since the two notations operate at very different levels of abstraction. Furthermore, constraints imposed by one notation now have to be considered for both notations. *E.g.*, AoUCM models do not need to be well-nested whereas ADORE requires all models to be well-nested. This leads to a set of assumptions that constrain AoUCM models to an ADORE-specific profile for which a successful transformation into ADORE can be guaranteed.

PicWeb Case Study.

The transformation from the AoUCM PicWeb model to the ADORE PicWeb model is for the most part quite intuitive. Components tagged with <<business process>> are transformed into ADORE modules. The start and end points in FIG. 1 are converted into the receive and reply activities of the orchestration in FIG. 6, respectively. While the responsibilities in the Actor component User are ignored, the responsibility in the Flickr <<service>> component and the truncate stub are mapped into the service invocations act_2 and act_3, respectively.

The content of the truncate stub in FIG. 2 is transformed into its own orchestration. Start and end points are transformed as described in the previous paragraph. The responsibility of the <<business process>> component Truncate is mapped to a local assignment (*i.e.*, act_3) instead of a service invocation because the transformation process is currently building the orchestration for the Truncate component instead of a different component. The remaining elements to be transformed are the OR-fork with its two branches and the corresponding OR-join, resulting in the gt activity with its guard relations and the exclusive waitFor relation links entering the reply activity[6].

For the Picasa service concern in FIG. 3, the start and end points are converted into the predecessor node P and successor node S of the fragment in FIG. 5, respectively. The two branches of the AND-fork are transformed into the two waitFor relation links exiting the predecessor node P. The pointcut stub turns into the hook activity and the responsibilities of the <<service>> components again are transformed into service invocation activities, *i.e.*, act_2 and act_3. Finally, the AND-join is reflected by the non-exclusive waitFor relation links entering act_3.

Benefits.

Using this transformation, designers retrieve immediately process *skeletons*, automatically generated from the requirements ($[R \to D]_1$). According to $[R \to D]_2$, the generated models are free of business inconsistencies. Consequently, designers can focus on the design of their SOA–based system without losing time and effort on requirements artifacts.

3.2 Transformation of Composition Rules

Thanks to the $[R \to D]_1$ point, concerns expressed at the requirements level will be mapped into associated artifacts in the design model. Choices made at the requirements level (such as concern composition and ordering) are kept and automatically projected into the design model.

⇒ $[R \to D]_3$: Concern composition and ordering defined in the requirements model are automatically reused in the design model.

According to its different goals, the design layer must handle more precise artifacts, as it aims to design an executable system. Consistency rules defined at the design layer can hence more easily identify interactions in the generated artifacts, such as unanticipated shared join points. Such interactions can then be solved at the requirements level.

⇒ $[D \to R]_1$: The design layer identifies interactions between several concerns around shared join points, which may not be described in the requirements model.

Implementation.

The composition techniques for ADORE models and AoUCM models are substantially different. While AoUCM allows regular expressions for the identification of join points, ADORE requires explicit bindings to be defined that identify the join points as exact locations in the ADORE model. For this reason, the AoUCM pointcut expressions are not transformed but rather the list of matched join points as it results from the AoUCM matching and composition algorithms is mapped onto ADORE's composition directives.

In addition, CIG precedence rules must be considered to take care of concern conflicts identified at the requirements level. While parsing the CIG model, the transformation algorithm generates composition directives to weave fragments into each other, and then instantiates the existing order of the fragments ($[R \to D]_3$). Essentially, a fragment with lower precedence that conflicts with another fragment must be explicitly applied to the hook activitiy of the other fragment. If ADORE detects remaining shared join points, the engine will trigger its default merge algorithm and inform the user of such a merge ($[D \to R]_1$).

PicWeb Case Study.

All AoUCM pointcut maps are transformed indirectly into ADORE's composition directives. For example, the pointcut map (bottom map in FIG. 3) matches the getPicturesWithTag responsibility of the Get Pictures concern, which is mapped onto act_2 of the getPictures orchestration. Hence, the picasaService fragment is applied to act_2 of the getPictures orchestration as shown in FIG. 5.

The CIG (FIG. 4) specifies the conflicts of the concerns in the PicWeb application. For example, the Payment concern conflicts with the Picasa concern because both apply to the getPicturesWithTag responsibility, *i.e.*, the shared join point. Furthermore, Payment has lower precedence as defined in the CIG. Hence, the Payment fragment is applied to the Picasa fragment with the composition directive "composition picasaService { apply Payment => act1; }" where act_1 is Picasa's hook activity. The same reasoning applies to the Picasa fragment: the composition directive derived from the pointcut map in FIG. 3 and shown in FIG. 5 must be overwritten because of the CIG preference rules. Hence, the Picasa fragment is applied to the hook activity of the Caching fragment which, in turn, is finally applied to the getPicturesWithTag activity of the getPictures orchestration.

Benefits.

Based on the information described in the requirements, the transformation algorithm generates *all* the process skeletons associated with PICWEB concerns that are composed

[6]Note that the XOR semantics of the guard relation in ADORE matches nicely the XOR semantics of an AoUCM OR-fork and that the exclusive waitFor relation also is compatible with the semantics of an AoUCM OR-join.

with each other as desired. Conflicting concerns identified at the requirements level are properly ordered in the design models ($[R \rightarrow D]_3$) and additional interactions are resolved ($[D \rightarrow R]_1$).

4. DATA–DRIVEN FEEDBACK AND REQ.

In Sect. 3, we mainly focus on design *skeleton* generation and essentially describe $[R \rightarrow D]$ information. We focus in this section on $[D \rightarrow R]$ information, that is, information discovered at the design layer which may impact the requirements. In this section, we use requirements expressiveness relative to *data* to strengthen the feedback a requirements engineer can obtain from the design layer to address inconsistencies in the requirements model. These mechanisms rely on data–driven analysis explained in this section.

4.1 Incomplete Requirements

The results of the transformation described in Sect. 3 are ADORE orchestrations and fragments as shown in FIGS. 6 and 5 but without any parameters, inputs, and outputs. As a consequence, all data–driven analysis capabilities of the design layer cannot be applied.

When the requirements model handles data (such as conditions expressed over exclusive paths), it is possible to also project this information into the design model. Then, data–driven analysis techniques can be applied to identify omissions in the requirements from the data usage point of view.

⇒ $[D \rightarrow R]_2$ The design platform identifies *inconsistent* data–flow introduced by omissions in the requirements models (*e.g.*, the requirements may assume the usage of data which is never defined in the system). This information is fed back to the requirements layer for iterative refinement.

Implementation.

Some data-related information exists in AoUCM models. Global variables are used in AoUCM models to formalize conditions of choice points. Furthermore, at least inputs from and outputs for Actor components are known even early in the development process. This information can be captured in the AoUCM model with the help of metadata and then also transformed into the ADORE model. Hence, responsibilities of Actor components are tagged with metadata named ST_in for inputs and ST_out for outputs. The values of the metadata describe the data including type information. The tagging with data-related information only looks at each responsibility in isolation, specifying only what is required by a responsibility and what is produced by a responsibility without specifying how the actual data flows through the system. Hence, this is still very much in the spirit of requirements models. Furthermore, if responsibilities in Actor components can be tagged with data-related information, so can responsibilities of <<business process>> and <<service>> components, leading to even more information that can be transformed into ADORE models and verified by its conflict detection rules. *E.g.*, the getPicturesWithTag responsibility of the Flickr component in FIG. 1 is tagged with the <<tag:String>> input and the <<pictures:Picture[]>> output.

The transformation process for ST_in and ST_out metadata is straightforward. Inputs and outputs are simply added to the local assignments and service invocations corresponding to responsibilities. However, the resulting ADORE model is not consistent and requires further manipulations. A business process modeled with ADORE requires its internal variable v_x to be assigned before used. The ADORE engine can also detect unused variables (as a bad smell), *i.e.*, variables assigned during the execution of a process but never used after the assignment. We propose two *refactoring* techniques to be used as default fixes when encountering such cases:

- *Pull in*: Let v be a variable used as input for a given activity, but never assigned before. The simplest solution to take care of this data assignment is to assume that it must have been received from the outside world. As a consequence, v is automatically added as an output of the initial *receive* activity.
- *Push out*: Let v' be a variable used as output in an activity, but never used afterwards. We assume this data should be returned to the outside world, and add v' as an input of the final *reply* activity.
- Fragment specialization: In a *fragment*, external variables are defined through the *hook* activity. Hence, unassigned variables are *pulled in* as *hook* input.

As these default fixes change business process interfaces, another refactoring technique must be applied to deal with *interface–mismatch*: an activity which invokes a refactored process must use the enriched interface (new inputs or outputs parameters). The *enrich* refactoring rule adds these new parameters inside such activities, and then propagates their usage in the process.

Binding of Matched Metadata.

Some concerns may have to make use of data-related information provided by model elements outside the concern. The Caching concern, for example, models a generic <<key>> and generic <<data>> in its AoUCM scenario model. Both need to be mapped to concrete data of the concern to which Caching is applied. In the case of PicWeb, these are the <<pictures:Picture[]>> and <<tag:String>> tags of the getPicturesWithTag responsibility. The mapping is achieved with the metadata named ST_bind (*i.e.*, <<data=pictures>> and <<key=tag>> are associated with the pointcut stub). The transformation process then simply substitutes the generic data with the concrete data in the fragment that corresponds to the AoUCM concern.

PicWeb Case Study.

According to the requirements model defined in FIG. 2, the truncate business process limits the number of received pictures according to boolean conditions (nrPhotos > threshold and nrPhotos ≤ threshold). Semantically, the nrPhotos variable contains the cardinality of the picture set, but this is only specified implicitly in the AoUCM model. When transformed into ADORE artifacts, the nrPhotos variable is detected as *unassigned*, and then *pulled in* by the previously described refactoring technique. As a consequence, the truncate business process interface now defines that the process must receive a set of pictures, a threshold, **and** the cardinality of the picture set. This, however, is redundant, since the cardinality can be easily computed internally. This result may therefore be used (according to $[D \rightarrow R]_2$) by requirements engineers to update the requirements models.

Benefits.
The transformation algorithm automatically generates artifacts expressed in the language daily used by the designer. Therefore, designers can more readily identify inconsistencies (such as receiving a set *and* its cardinality) based on their experience. In this case, one may decide that this issue is too *technical* and must therefore be handled at the design level. Alternatively, a new responsibility may be defined in the requirements model to address the problem and maintain traceability between the models.

4.2 Design Choice Impacting Requirements

In Sect. 4.1, we considered *omissions* in the requirements model which lead to inefficient models at the design layer. This section discusses the existence of a design choice in the generated model, *i.e.*, a divergence between the generated artifacts and the actually expected design model.

⇒ $[D \to R]_3$ Design choices (such as service definition) may lead to the modification of the requirements models to remain consistent with the design model.

Implementation & PicWeb Case Study.
The implementation of this mechanism is based on the one described in the previous section, *i.e.*, the existence of data–driven concerns in the requirements models.

The Payment fragment (Fig. 9(a)) of the PicWeb system allows one to restrict the availability of a given service if the number of executed calls is greater than a daily limit[7]. A designer will identify three *points* in the Payment fragment where the $[D \to R]_3$ information is useful:

- act_2: This activity is named returnNoPhotos, but never interacts with the pictures variable. It, however, is necessary to assign the empty list to pictures while executing this activity because the pictures variable is a required result of this fragment.
- act_1: This activity uses two variables to check if the service should really be called: nrRequests and requestLimit. According to the refactoring rules, these variables are pulled–in into the hook (act_3), as they are not defined anywhere else.
  - From a design point of view, the requestLimit variable should be retrieved from a dedicated service defined in the SOA. As such a limit restriction is common in the SOA world[8], it should be handled in the requirements.
  - The nrRequest variable implicitly introduces the existence of a counter which needs to be incremented after each call and reset at the beginning of a usage period (considering the limit restriction as a requirements–driven concern, it is an omission in the requirements to not count the calls).

As a consequence, the three choices made at the design level by SOA experts impact the requirements model: responsibilities must be added to enrich the models with these new entities as highlighted in Fig. 9(b). The transformation is then re–played to obtain an updated Adore model.

[7] The Flickr service enforces such a restriction.
[8] E.g., cloud–driven services such as GoogleApp Engine allow a number of free calls before charging a pay–per–call fee.

Benefits.
Designers only focus on their field of expertise and can identify problems in the system (such as omissions) without having to build a design model from scratch. The work effort is reduced, and the interventions are more accurate and precise. The requirements engineer, on the other hand, benefits from an updated requirements model which can be verified against the stakeholders' goal models and re-evaluated by requirements analysis techniques to ensure that the changes do not conflict with the intended results.

4.3 Verification of User Interactions

In AoUCM models, detailed user interactions are captured by ST_in and ST_out metadata associated with responsibilities of Actor components[9]. *E.g.*, the responsibility enterTagAndThreshold in Fig. 1 is tagged with <<tag:String>> and <<threshold:Integer>> inputs, while the responsibility browsePictures is tagged with the <<pictures:Picture[]>> output. These inputs/outputs are compared with the final business process interfaces after the refactoring steps to ensure that information is provided as specified by the interface.

⇒ $[D \to R]_4$ Enriched business process interfaces are verified against the specification of detailed user interactions in the requirements model for further consistency between the two abstraction levels.

Benefits.
If there is a mismatch (*e.g.,* the User only enters a tag but not a threshold), then this gives an indication that there is a mistake in the requirements model or that additional activities need to be added to the design model to cover the required information.

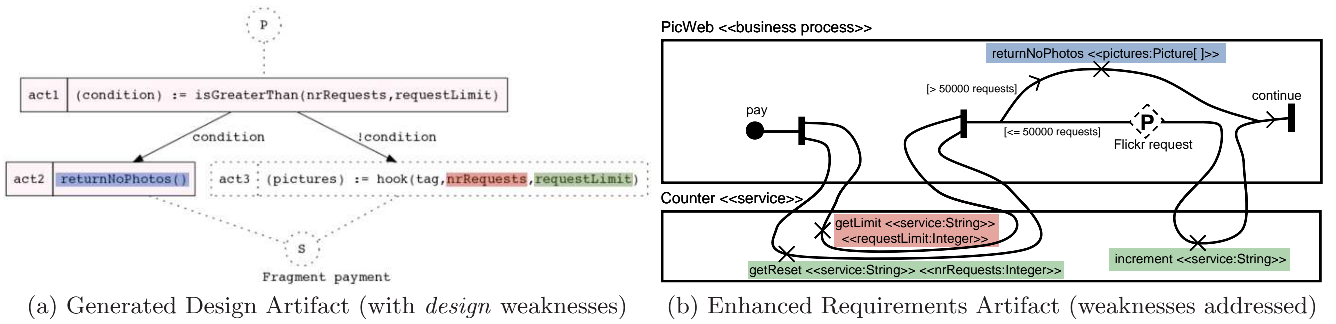
5. RELATED WORK

At the beginning of the development process, AoURN has been used in the context of business process modeling, monitoring, and improvement over the last few years. Pourshahid *et al.* [23] use AoURN goal models augmented with Key Performance Indicators (KPI) to assess the state of business processes and adapt them, if necessary, with the help of business process redesign patterns expressed as AoURN concerns. AoURN's goal and scenario models describe the behavioral, structural, and intentional dimensions of each pattern.

Jacobson and Ng [8] propose an aspect-oriented software development process based on use cases. Aspect techniques allow use cases to be encapsulated throughout the software development process. While the authors define the kind of requirements and design models required for modeling use cases in an aspect-oriented and traceable way, they are not focusing on automatic transformations between requirements models and design models.

In [9], the authors propose Ram, an aspect-oriented modeling approach that provides multi-view modeling, covering structural, state-based, and scenario-based models. Adore and AoURN focus on the behavioral point of view based on scenarios and workflows with only limited structural modeling. However, the consistency of AoURN requirements and Adore design models can be greatly improved when

[9] *E.g.*, the User in the PicWeb application.



(a) Generated Design Artifact (with *design* weaknesses) (b) Enhanced Requirements Artifact (weaknesses addressed)

Figure 9: Illustrating Design Choices – ($[D \rightarrow R]_3$) Information

they are combined with more detailed structural information as provided by Ram. A detailed structural view is also useful when extracting service interfaces from orchestration and fragment definitions and when applying composition algorithms such as Kompose [6] to obtain the service models (*e.g.*, with the help of Uml class diagrams). AoURN, on the other hand, offers intentional models that describe stakeholder goals as a complementary view of the *reasons* for system choices and decisions.

At the end of the development process, several approaches fill the gap between orchestrations and Aop (*e.g.*, [2, 4, 24]). These approaches rely on the Bpel language and impose dedicated Bpel execution engines to interpret the aspects. Adore preaches technological independence and exposes itself as a *meta–model* to support composition [13]. Instead of interpreting *aspectized* Bpel code, Adore focuses on the design of workflows by composition and weaving of fragments.

6. CONCLUSIONS & PERSPECTIVES

In this paper, we address the interactions of requirements and design phases during an iterative aspect-oriented software development (AOSD) process in the SOA context. We propose an automatic transformation to help requirements engineers and designers convert requirements artifacts into design models. We also identify several classes of information (*i.e.*, $[R \rightarrow D]_i$ and $[D \rightarrow R]_i$) that enhance a design model based on the requirements model and vice versa. This, together with the transformation, ensures the consistency of models, reduces the amount of work done in one model that needs to be repeated in the other, and makes analysis results from one notation available to the other. The proposed approach was successfully applied as a proof-of-concept case study to the PicWeb application using the AoURN and Adore technologies. Our research yields several lessons learned as discussed in the following paragraphs.

Mapping of Concerns. In our experiments, it was possible to map concerns defined in the requirements scenario models one-to-one onto concerns in the design models, simplifying traceability and indicating that an end-to-end AOSD process is feasible in the SOA context. The similarity of AoUCM and Adore models plays a large role in this concern mapping. Concerns related to AoURN's goal models, on the other hand, are not mapped directly onto concerns in Adore but rather indirectly through AoURN's intrinsic relationships between goal and scenario concerns. Keeping concerns across development phases was not nearly as challenging as identifying those semantic concepts in both notations for which transformation rules can be established.

Loss of Expressiveness. The current transformation rules restrict the notations to a common, conceptual subset, thus constraining the AoUCM notation to a *profile*. It remains to be seen whether this loss of expressiveness at the requirements level is a real hindrance in practice, *i.e.*, some concerns will be more difficult to model with a restricted AoUCM notation. Alternatively, further AoUCM elements could be supported by more advanced transformation rules. However, any additions to the Adore notation have to be carefully weighed against Adore's need to keep its meta–model as reduced as possible to be in a position to prove its composition mechanisms. More complex AoUCM workflow concepts such as dynamic stubs, synchronizing stubs, and blocking stubs [7] could potentially be expressed as patterns of existing concepts in Adore. Other concepts such as timers [7] could be mapped to Adore fragments and then composed with processes.

Lightweight Modeling of Detailed Data. The lightweight addition of detailed data in the requirements model allowed for a much more effective use of Adore's data–driven analysis capabilities. Furthermore, Adore's refactoring techniques "fill in" the overall data-flow of a system based on simple, local definitions of required inputs and delivered outputs that can even be determined during the requirements phase. More research is needed to ascertain how far tool-generated data-flow can go in the design of SOA–based systems. More advanced refactoring techniques could potentially resolve even much more complicated data-flow situations and relieve the designer from yet another burdensome task.

To Unify or Not To Unify. During this research, we had the choice to apply Adore's data–driven analysis capabilities directly to AoUCM models. Essentially, this would have amounted to unifying the AoUCM and Adore notations. Instead, we opted to define a transformation process to retain greater flexibility in applying each notation's unique capabilities, *i.e.*, AoUCM's connection to goal models and trade-off analysis as well as Adore's composition mechanisms and connection to Bpel implementations.

Even if the PicWeb example is rather small, it is part of an existing SOA–based system that is in use daily, which makes it pertinent in the context of SOA experimentation. The PicWeb model covers all notational elements supported by the transformation, and all major composition rules are used for its crosscutting concerns. In addition, we have ev-

idence from a much larger case study [11, 16] that AoURN and ADORE scale well to larger systems. In terms of the PicWeb models being specified by experts, we argue that this is actually necessary to establish the most appropriate transformation rules. However, more experiments are necessary to investigate how the proposed approach fares with less-experienced modelers.

Tool support for our approach is provided by the jUCMNav tool and ADORE. All models of the case study have been defined with the tools. The actual transformation has not yet been integrated with the jUCMNav tool but this is a rather manageable future task compared to the complexity of other transformations that already exist in the jUCMNav tool (*e.g.*, to Message Sequence Charts or the DOORS requirements management system). Facilities already exist in jUCMNav to check for well-nestedness and OCL constraints, allowing an AoURN profile for ADORE to be enforced. The refactoring techniques from Sect. 4.1 are part of the ADORE kernel[10].

Finally, we plan to combine the results of this work with the work of Pourshahid *et al.* [23] and on-going work on the generation of BPEL code from ADORE to establish an end-to-end modeling environment for SOA–based systems that enables runtime monitoring and runtime adaptation of business processes.

Acknowledgments. This research was supported by the Discovery Grants program of the Natural Sciences and Engineering Research Council of Canada as well as the Ontario Graduate Scholarship Program.

[10]See `http://code.google.com/p/adore/source/browse/trunk/prolog/algos/refactor.pl` for the implementation (expressed as PROLOG predicates over the ADORE logical model).

7. REFERENCES

[1] URN Virtual Library. http://www.usecasemaps.org/pub, 2010.

[2] A. Charfi and M. Mezini. Aspect-oriented web service composition with ao4bpel. In *ECOWS*, volume 3250 of *LNCS*, pages 168–182. Springer, 2004.

[3] R. Chitchyan, A. Rashid, P. Sawyer, A. Garcia, M. Pinto Alarcon, J. Bakker, B. Tekinerdogan, S. Clarke, and A. Jackson. Survey of Analysis and Design Approaches. Technical report, AOSD-Europe Report ULANC-9, 2005.

[4] C. Courbis and A. Finkelstein. Weaving aspects into web service orchestrations. In *ICWS*, pages 219–226. IEEE Computer Society, 2005.

[5] ERCIM. Terminology. Technical report, ERCIM, 2010.

[6] R. France, F. Fleurey, R. Reddy, B. Baudry, and S. Ghosh. Providing support for model composition in metamodels. In *EDOC'07 (Entreprise Distributed Object Computing Conference)*, Annapolis, MD, USA, 2007.

[7] International Telecommunication Union (ITU-T). Recommendation Z.151 (11/08): User Requirements Notation (URN) - Language Definition, approved November 2008.

[8] I. Jacobson and P.-W. Ng. *Aspect-Oriented Software Development with Use Cases*. Addison-Wesley Professional, 2005.

[9] J. Kienzle, W. Al Abed, and J. Klein. Aspect-oriented multi-view modeling. In *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2009. ACM.

[10] M. MacKenzie, K. Laskey, F. McCabe, P. Brown, and R. Metz. Reference Model for Service Oriented Architecture 1.0. Technical Report wd-soa-rm-cd1, OASIS, feb 2006.

[11] S. Mosser, M. Blay-Fornarino, and R. France. Workflow Design using Fragment Composition (Crisis Management System Design through ADORE). *Transactions on Aspect-Oriented Software Development (TAOSD) Special issue on Aspect Oriented Modeling*, pages 1–34, 2010.

[12] S. Mosser, M. Blay-Fornarino, and J. Montagnat. Orchestration Evolution Following Dataflow Concepts: Introducing Unanticipated Loops Inside a Legacy Workflow. In *International Conference on Internet and Web Applications and Services (ICIW) AR=28%*, Venice, Italy, May 2009. IEEE Computer Society.

[13] S. Mosser, M. Blay-Fornarino, and M. Riveill. Web Services Orchestration Evolution : A Merge Process For Behavioral Evolution. In *2nd European Conference on Software Architecture (ECSA'08) AR=14%*, Paphos, Cyprus, Sept. 2008. Springer LNCS.

[14] S. Mosser, F. Chauvel, M. Blay-Fornarino, and M. Riveill. Web Service Composition: Mashups Driven Orchestration Definition. In *International Conference on Itelligent Agents, Web Technologies and Internet Commerce(IAWTIC'08) AR=29% , long paper*, pages 284 – 289. IEEE Computer Society, Dec. 2008.

[15] G. Mussbacher and D. Amyot. Extending the User Requirements Notation with Aspect-oriented Concepts. In R. Reed, A. Bilgic, and R. Gotzhein, editors, *SDL 2009: Design for Motes and Mobiles*, volume 5719 of *Lect. Notes Comp. Sci.*, pages 115–132. Springer, 2009.

[16] G. Mussbacher, D. Amyot, J. Araújo, and A. Moreira. Requirements Modeling with the Aspect-oriented User Requirements Notation (AoURN): A Case Study. In S. Katz, M. Mezini, and J. Kienzle, editors, *Transactions on Aspect-Oriented Software Development VII*, volume 6210 of *Lect. Notes Comp. Sci.*, pages 23–68. Springer, 2010.

[17] G. Mussbacher, D. Amyot, and M. Weiss. Visualizing Early Aspects with Use Case Maps. In A. Rashid and M. Aksit, editors, *Transactions on Aspect-Oriented Software Development III*, volume 4620 of *Lect. Notes Comp. Sci.*, pages 105–143. Springer, 2007.

[18] I. Nagy, L. Bergmans, and M. Aksit. Composing Aspects at Shared Join Points. In *NODe/GSEM*, pages 19–38, 2005.

[19] Y. Natis. Applied SOA: Conquering IT Complexity Through Software Architecture, Gartner Group, may 2005.

[20] OASIS. Web Services Business Process Execution Language Version 2.0. Technical report, OASIS, 2007.

[21] U. of Ottawa. jUCMNav website: http://softwareengineering.ca/jucmnav, 2010.

[22] M. P. Papazoglou and W. J. V. D. Heuvel. Service Oriented Design and Development Methodology. *Int.*

J. Web Eng. Technol., 2(4):412–442, 2006.
[23] A. Pourshahid, G. Mussbacher, D. Amyot, and M. Weiss. Toward an Aspect-Oriented Framework for Business Process Improvement. *International Journal of Electronic Business (IJEB)*, 8(3):233 – 254, 2010.
[24] B. Verheecke, W. Vanderperren, and V. Jonckers. Unraveling crosscutting concerns in web services middleware. *IEEE Software*, 23(1):42–50, 2006.

Appendix: Transformation Process

To determine the scope of the tranformation of AoUCM models into ADORE models, a standard AoUCM model is tagged with metadata. A component may either be tagged with <<business process>> or <<service>> (*i.e.*, values of metadata named ST_type). <<business process>> components constitute modules that are to be built for the system. More than one of them may exist in the model and each distinct path through one of them corresponds to a specific orchestration or fragment in ADORE. <<service>> components represent already existing, external services (*e.g.,* Flickr). Again, several such components may exist and each path through such a component results in service invocation(s) of the external service in the ADORE model.

The transformation traverses the AoUCM model beginning at the start points of the concerns' root maps (regardless of whether the concern is crosscutting or not). Root maps are at the highest level in the map hierarchy - a root map may contain stubs which lead to further maps but a root map itself is not plugged into any stub. A concern's root map is hence at the highest level in the map hierarchy of the concern. In general, each root map of a non-crosscutting concern is transformed into an ADORE orchestration, while those of a crosscutting concern are transformed into ADORE fragments. During the traversal of the AoUCM model, each AoUCM path element is transformed into its ADORE equivalent and added to the orchestration/fragment. Note that only a subset of AoUCM path elements is supported by the transformation process as corresponding concepts are not always readily available in ADORE.

AoUCM Profile for ADORE.

The presented approach imposes several constraints on the AoUCM model. First, AoUCM models must be well-nested[11]. Consequently, a root map can only have one start point and one end point. Plug-in maps, however, may have multiple start/end points as long as all of them are used by the plug-in map's stubs.

Second, all path elements except stubs must be bound to a component to ensure that all behavior is assigned to a module in ADORE. Third, components tagged with <<service>> must contain only responsibilities as external service components are black-box in a SOA context. Fourth, the AoUCM model must not contain path elements that are not supported in TAB. 1. Fifth, a static stub must have its plug-in map defined. Note that OCL rules can be defined and automatically checked by the jUCMNav tool for all constraints in this paragraph.

[11] *I.e.*, OR-forks and OR-join only appear in pairs, an OR-fork must be followed by an OR-join, AND-forks and AND-joins only appear in pairs, and an AND-fork must be followed by an AND-join.

Transforming Path Elements.

The transformation from AoUCM models to ADORE models assumes that the AoUCM model conforms to the ADORE profile. The procedure transformMap is called for each root map in the AoUCM model, regardless of whether it belongs to a crosscutting concern or not. Note that each root map belongs to exactly one concern. While the AoUCM model is traversed to transform each path element into ADORE elements, transformMap is called recursively to deal with lower level maps (see bolded highlights in FIG. 10). For any root map *r*, transformMap(null, am, r, null, true) is called and the transformed ADORE model *am* is returned. The parameters of transformMap describe the following:

context (the first parameter) describes for which component an ADORE model is created. If it is null, the context will be determined by transformMap (lines 2-3). The context is determined by the component tagged with <<business process>>. If there is no such component on a root map, then a generic context is used for the ADORE model. If there are more than one such component, then the first one is used. This is an arbitrary choice to keep the transformation process less complicated. The context parameter may be not null when transformMap is called recursively as it may already have been determined.

am (the second parameter) defines the ADORE model to which the results of the transformation are added. A new ADORE fragment or orchestration will be created by the process (lines 4-6). If the map belongs to a crosscutting concern, then an ADORE fragment is created, otherwise an ADORE orchestration is created. The union of all ADORE fragments or orchestrations generated for each root map constitutes the final ADORE model for the whole AoUCM model. As plug-in maps may be shared (*i.e.*, plugged into many stubs) and transformMap is called recursively, an attempt may be made to transform an already transformed map again. In this case, the already existing ADORE orchestration or fragment is used and the recursion ends (lines 7-8).

The name of an ADORE orchestration consists of two parts (*i.e.*, <ServiceProvider>::<ProvidedService>). The first is determined by the context whereas the second is determined by the name of the **map** (the third parameter). The name of an ADORE fragment consists only of the provided service and is therefore determined by the map alone.

p (the fourth parameter) indicates the current path elements that are to be transformed into ADORE (lines 9-11). If it is null, then it is initialized to the map's start point (note that there can only be one start point for a map because AoUCM models need to be well-nested).

flag (the fifth parameter) indicates whether the whole map (flag = true) or only the portion of the map inside the context starting from p (flag = false) is to be transformed. transformMap passes this parameter into transformElement which passes it into getNextPathElements (lines 43-45) where the flag is finally considered.

transformElement performs the actual transformation of AoUCM path elements into ADORE model elements while traversing the AoUCM model. transformElement has to deal with four possible cases. Lines 25-28 cover the case with a component tagged with <<service>> (*i.e.*, service invocation). Lines 29-30 ignore non-tagged components and Actor components. Lines 31-42 cover the cases for elements except stubs. If the component of the current element is the same as the context, then the transformation rules from TAB. 1

```
1   AdoreModel transformMap(context:Component, am:AdoreModel, map:UCMmap, p:PathNode[], flag:Boolean) {
2       // determine context
3       if (context == null) { context = selectFirstComponentTaggedBusinessProcess(map); }
4       // establish Adore fragment or orchestration
5       if map.getConcern().isCrosscutting() { created = am.addFragment(map, context); }
6       else { created = am.addOrchestration(map, p, context); }
7       // end recursion if the fragment/orchestration already exists
8       if (!created) { return am; }
9       // transform path element - also traverses the map to the next path element(s)
10      if (p == null) { p = map.getStartPoint(); }
11      am = transformElement(context, am, p, flag) // starts recursion
12      return am;
13  }
14
15  AdoreModel transformElement(context:Component, am:AdoreModel, p:PathNode[], flag:Boolean) {
16      foreach element e in p {
17          // deal with a static stub – requires examination of its plug-in map
18          if e.isStub() {
19              if (!e.getPluginMap().contains(context) &&
                e.getPluginMap().containsOtherComponentTaggedBusinessProcess(context)) {
20                  am.addServiceInvocation(e);
21                  am = transformMap(null, am, e.pluginMap(), null, true);
22              }
23              else { am = transformMap(context, am, e.pluginMap(), null, true); }
24          }
25          // deal with responsibility inside an external service component (element must be a responsibility)
26          elseif (e.isInsideComponentTaggedService()) {
27              am.addServiceInvocation(e);
28          }
29          // ignore elements that are inside non-tagged components or actors
30          elseif (e.isInsideNontaggedComponent() || e.isInsideActor()) { }
31          // deal with elements inside the current context
32          elseif (e.isInsideContext()) {
33              am.addElementTransformedAsPerTable1();
34          }
35          // and finally, deal with elements that are not inside the current context
36          else {
37              am.addServiceInvocation(e);
40              am = transformMap(e.getComponent(), am, map, e, false);
41              p = p.skipToLastPathElementInsideOtherContext(e.getComponent());
42          }
43          // traverse the model to the next path elements following the current path element and transform them recursively
44          // this does not traverse beyond a join unless all branches have arrived at the join
45          am = transformElement(context, am, p.getNextPathElements(context, flag));
46      }
47      return am; // if p is empty, the recursion stops
48  }
```

Figure 10: Transforming AoUCM Models into Adore Models

apply (line 33). If it is not the same as the context, then a service invocation is added and transformMap is called recursively (line 40) to deal with the portion of the path inside the other component for which a new ADORE orchestration is created. Note how the flag is set to false and the element's component is set as the context. Line 41 ensures that the portion of the path inside the other component which is dealt with recursively is not traversed a second time.

Finally, lines 18-24 deal with static stubs, which require an examination of a stub's plug-in map. If the plug-in map does not contain a component that is the same as the context but another component tagged with <<business process>>, then a service invocation summarizing the plug-in map is added to the ADORE model and transformMap is called recursively (line 21) to create a new ADORE orchestration for the plug-in map. Otherwise, the plug-in map is an extension of the current context and the traversal continues with the same context with the path on the plug-in map as if it were on the parent map (line 23).

Lines 43-45 transform the path elements following the current path element in the model. If the flag is set to false, only those next path elements are considered that are in the context. If there are more than one next path element (*e.g.,* for OR-forks), an array of path elements is returned by getNextPathElements. If there are no next path elements, then an empty array is returned which stops the recursions. This may be the case if the end of the AoUCM model has been reached or a join has been reached for which not all branches have yet arrived.