MORSE: A Model-Aware Service Environment

Ta'id Holmes, Uwe Zdun, and Schahram Dustdar
Distributed Systems Group, Institute of Information Systems
Vienna University of Technology
Vienna, Austria
{tholmes, zdun, dustdar}@infosys.tuwien.ac.at

***Abstract*—In a number of scenarios, services generated using a model-driven development (MDD) approach could benefit from "reflective" access to the information in the models from which they have been generated. Examples are monitoring, auditing, reporting, and business intelligence scenarios. Some of the information contained in the models of a service can statically be generated into its source code. In a distributed and changing environment this approach is limited, however, due to the fact that models and their relations evolve after the generation and deployment of a service. For example, the current model of a service might be different than the deployed version of the service. Our approach to solve this issue is a Model-Aware Service Environment (MORSE). It consists of a model repository that manages MDD projects and artifacts, and model-aware services that interact with the repository for performing reflective queries on the models stored in the repository. Thus, MORSE supports the dynamic, reflective lookup of models in service-oriented systems.**

***Keywords*-model-aware; service; SOA; model; MDD; repository**

I. INTRODUCTION

Nowadays, there is a trend toward the use of precisely specified models, for example, for model-driven development (MDD) [1], [2]. Some reasons for using MDD are: Instances of the models can be validated for specified properties. Models can be defined and refined at different abstraction levels. This makes the models suitable to be used by various stakeholders, e.g., domain experts who use high-level, graphical models and technical experts who work with more low-level, textual models [3], [4]. Technical expertise can be captured in transformations, e.g., when platform-independent models are transformed to platform-specific models. This enhances portability and simplifies adaptations. Recurring code can be generated, easing the maintenance of a model-driven system.

Due to these benefits, many MDD approaches for service-oriented architectures (SOA) have been proposed (e.g., [5], [6], [7]). These approaches are model-driven in the sense that the SOA is specified using models and large parts or the whole source code of the SOA (including for example Web service code, WSDL files, policy code, business object implementations, and so on) is generated from those models.

While the model-driven SOA approach is highly useful in many cases, it has its limitations for scenarios that require information from the models at runtime because generation currently happens only at design time. Hence, all information that is needed at runtime from the models must be foreseen by the developers and must be statically generated into the source code. Many SOAs require "reflective" model information for monitoring, auditing, reporting, and business intelligence purposes. The requirements for this kind of reflection on model information can change quickly and are hard to foresee. However, regenerating and redeploying major parts of the SOA source code, because a certain model element's information is not exposed, is often not feasible in large distributed architectures. In addition, developing new reflection functions for each single model element is costly.

Finally, model information that is generated into or attached to a model-driven service is only up-to-date at its generation time. This is problematic when the service needs to reflect on information that was supplied after its generation and deployment. This is particularly true in a distributed and persistently evolving environment. Statically generated services need to keep up with changes such as additional model relations, e.g., new annotation models or a new version of the model of a service.

Our approach to solve these issues, i.e., facilitate services to dynamically reflect on models, is to create model-aware services (and components) for the SOA and support these with a model repository. We call such a SOA a *Model-Aware Service Environment* (MORSE).

During the MDD process, each model of the SOA is placed in the model repository. Each model and model element gets a Universal Unique Identifier (UUID) [8] assigned, with which the model or model element can be uniquely identified. The UUIDs are generated into the source code of the model-aware services (and components). Hence, they are *model-aware* in the sense that they can retrieve the models from which they have been generated from the repository at runtime using a service. In the same way, other components such as monitoring, auditing, reporting, and business intelligence components, or MDD tools such as a model-driven generator, can retrieve these models.

The repository service interface is a generic interface, and the UUIDs are generically added to generated code. Hence, no changes of the generated SOA are necessary in order to use a model element at runtime that has not been used before. Also the model-aware services can access evolved models and model relationships that occurred after generation time of the model-aware service.

In the following sections we introduce MORSE, present the design of the repository, and introduce some model-aware services. We will describe how model-aware services interact with the repository, how they can be created, and what functionality they may expose to other services.

This paper is structured as follows: In the next section we will give a broader motivation from the general MDD perspective and explain how MORSE addresses the general problems of traceability and collaboration in MDD systems. We will then present our approach by describing MORSE, the Model-Aware Service Environment, in Section III, focusing on the more specific issues of supporting model-aware services. Next, in Section IV we propose the model repository for facilitating model-aware services that we present in Section V. We will illustrate our work with a case study in Section VI. Section VII compares our approach to related work, and in Section VIII we conclude and refer to future work.

II. MOTIVATION FROM THE MDD PERSPECTIVE

In the broader view of model-driven systems in general, MORSE addresses two common problems in MDD systems: traceability and collaboration. In this section, we want to motivate both and explain briefly how the MORSE approach helps to address them. We describe these two common problems also to set the scope for this paper and delineate which parts of the general two problems are addressed by the model-aware services approach and which are not.

A. Traceability in Model-driven Systems

A particular problem of model-driven systems is traceability – asking the question: How do models and model elements of different abstraction layers and/or code correspond to each other? In particular, the traceability information for models that are transformed into other models or code can get lost in model-driven approaches. On the one hand, a transformation rule describes how source models are mapped to target models. On the other hand a traceability link at a target model would allow for identifying the source models. Traceability is essential for meaningful feedback from the runtime to stakeholders and for identifying and understanding the root cause, e.g., in case of a failure or exception. This is because, if we are able to trace the source model from which a target (model or code) has been created (via generation or transformation), it is possible to use the information in the source model to analyze or debug the target.

MORSE helps to address this problem of traceability as it manages MDD projects and artifacts and relates them to UUIDs that are generated into target systems. Moreover, MORSE facilitates such systems to exploit their traceability links via UUIDs by querying and reflecting on models, model elements, and model relationships.

B. Support for Collaboration in Model-driven Systems

Most current tool support for model-driven development only focuses on the design time and comes with limited collaboration features, if any. Model-aware services, however, rather assume a distributed environment, maybe even distributed development. In order to facilitate various services of a distributed environment, however, to concurrently work with MDD projects and artifacts we need to support the management of projects and artifacts, with (1) versioning capabilities while capturing and keeping track of model relationships and (2) services for the information retrieval and the management of these. For (3) facilitating collaboration scenarios, we also need to (4) deal with concurrency, e.g., provide locking mechanisms, raise the awareness of the work of others, offer compare and merge possibilities as well as support for resolving conflicts.

MORSE, as it is presented in this paper, addresses the first two of these issues, versioning capabilities and services for information retrieval and management, as these features are also needed for the monitoring, auditing, reporting, and business intelligence purposes addressed by the model-aware services approach proposed in this paper. For example, a monitoring component requires a service for retrieving model information from the MORSE repository, and it requires the models in the version of the model instance that it monitors. Please note that the collaboration features of MORSE could also be used for other scenarios, such as supporting the MDD development process for distributed MDD development.

III. MODEL-AWARE SERVICE ENVIRONMENT

For facilitating services to dynamically work with models in a SOA, we propose MORSE, the Model-Aware Service Environment. MORSE consists of a model repository and model-aware services that interact with the model repository using generic service-oriented interfaces. Figure 1 gives an overview of the MORSE. From the repository model-aware services can be generated that interact with the information retrieval interface. Also services with traceability information that emit events to model-aware services can be generated. In the following sections we will discuss these services in more detail.

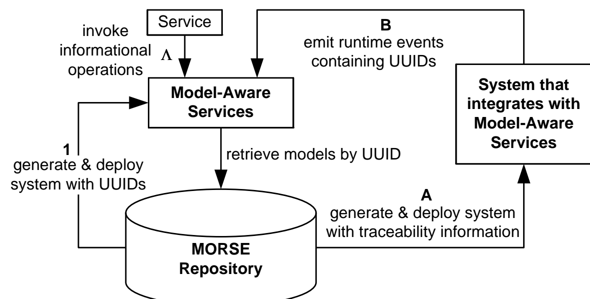


Figure 1. Overview of the Model-Aware Service Environment

Figure 2 gives an high-level overview of the model repository architecture. Different Web service interfaces allow for the administration and resource management of MDD projects and artifacts and offer information retrieval functionality (see end of the following section) to model-aware services. The MORSE builder service can create

these model-aware services. Also it can weave UUIDs[1] of MORSE objects into generated code. A deployment service is used for deploying resulting services and processes on runtime engines.

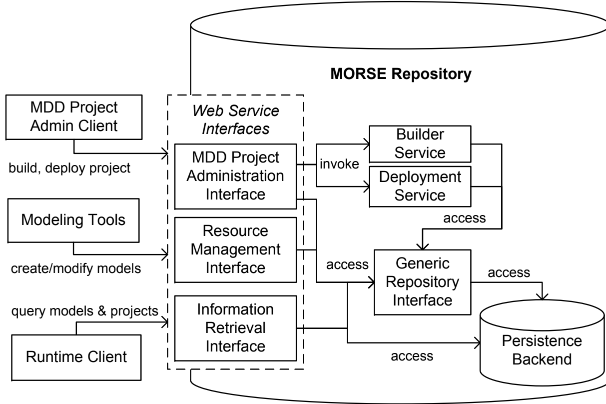


Figure 2. Architecture of the MORSE Repository

Because all MDD projects and artifacts are managed in a common model repository, model-aware services can query these for any information on themselves and other model-driven components. Although models and model relations may evolve over time, using UUIDs, it is always possible to retrieve a specific version of a MORSE object. Derived versions, e.g., new versions of the model that were created after deployment time, can easily be identified, permitting a model-aware service to e.g., retrieve and work with the latest version of a model.

MORSE can also be beneficial for MDD tools, e.g., in a distributed, collaborative development environment [9] while fostering service-orientation to support the MDD design-time tooling. In this case, not the model-aware services or components monitoring them would retrieve and change the models, but the MDD tools such as a model-driven generator. In this paper, we do not go into more details about this use scenario, as we want to concentrate on the case of using model-aware services at runtime. However, the MORSE tools themselves use the repository in this way. In Section V we will illustrate this use of MORSE (see Figure 6).

IV. MODEL REPOSITORY

The model repository is the main component of MORSE and has been designed with the goal to abstract from specific technologies. Thus, while concepts are taken from e.g., the Unified Modeling Language [10] and also version control systems, MORSE is particularly agnostic to specific modeling frameworks or technologies.

[1]Universal Unique Identifier (UUID) [8] is a standard for unique identifiers in (distributed) software system development. UUIDs are used in MORSE to uniquely identify models and model elements across distributed components, such as the model-aware services, the model repository, and other components using the MORSE services such as monitors.

The MORSE repository manages objects (`MObject`) such as projects (`MProject`) and artifacts (`MArtifact`) as shown in Figure 3 and 4(a). Additional MORSE object types (explained below) are shown in Figure 4(b). All `MObjects` are identifiable by `uuids` and can be associated with Dublin Core [11] metadata such as `title`, `creator`, or `date`. Note that a UUID also uniquely identifies a particular version of a MORSE object. By navigating across the `original` or `modified` relations however, previous and derived versions can be identified.

Artifacts are used to manage models and model elements (for details see below), model transformations, and MDD workflows. Besides the versioning of these, the repository supports branching (`MBranch`) and tagging (`MTag`) of projects. Note that artifacts can be shared by multiple projects as they can be associated by different tags and branches. They can be changed independently and merged later on.

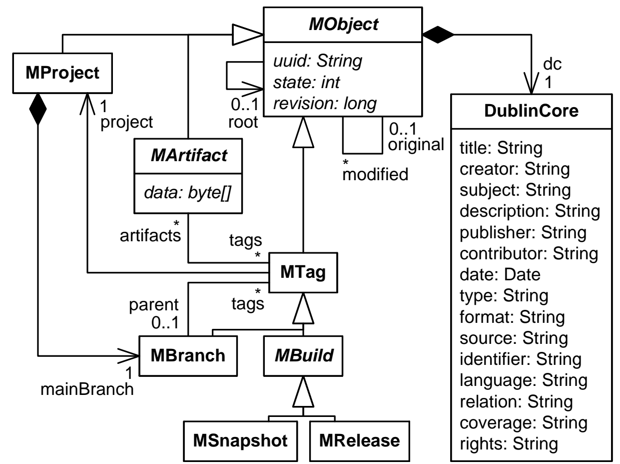
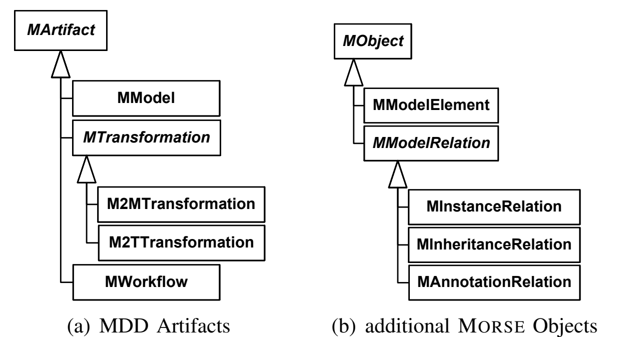


Figure 3. MORSE Objects and Projects

Typical MDD projects consist of models, transformations, and workflows. An example of a MDD development framework that works with these artifacts is openArchitectureWare (oAW) [12]. We have adopted these artifact types and support them in MORSE as shown in Figure 4(a).



(a) MDD Artifacts (b) additional MORSE Objects

Figure 4. MDD Artifacts and additional MORSE Objects

Besides the general management of MDD artifacts, MORSE particularly realizes support for models. Mod-

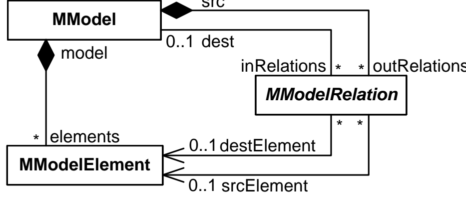


Figure 5. Model Element and Model Relations

els typically contain model elements and have relationships to other models. By capturing and keeping track of these, MORSE facilitates reflection on models, model elements, and model relations. Figure 5 illustrates `MModel`, `MModelElement`, and `MModelRelation` classes that represent these concepts. All these classes derive from `MObject` and are identifiable and versionable as such. Moreover, `MModels` are `MArtifacts` and can use the `data` attribute to save a serialized form of a model. Examples of different relations are instance-of, inheritance, and annotation relations as shown in Figure 4(b). A model relation (`MModelRelation`) has a source (`src`) and destination (`dest`) model, e.g., an annotated model is the destination model of a `MAnnotationRelation` and the annotation model depicts the corresponding source model. Besides referring to the models, a model relation may also specify actual model elements (`srcElement` and `destElement`).

While it would be possible to further specify details, e.g., of model elements, the presented concepts are sufficient for our purposes, i.e., to make models and model-elements identifiable and to capture dependencies between different models as introduced though their relations. The models that are stored and versionized within `MModels` can be retrieved in their serialized form and can further be processed by technology-specific tools, e.g., for introspection, model transformation, or model checking.

For the presented classes and concepts, the model repository exposes different services as indicated in Figure 2. Besides an administrative and resource management interface, the repository particularly offers an information retrieval interface to model-aware services. Some of its operations are listed in Table I. A UUID of the specified type is passed to the operation that usually returns MORSE objects. Most operation names are derived from the role or property names of a class. Besides these, there are some operations that allow for additional queries, e.g., the operation `getMObject.derived` returns all derived MORSE objects for the UUID of a MORSE object, i.e., the operation identifies all derived MORSE objects by traversing the `modified` relations. Sometimes it suffices to request the UUIDs instead of the objects. The last row shows an operation name with an `.uuid` suffix that can be appended for such a purpose. Besides these operations, the information retrieval interface also allows clients to pass complex queries to the persistence backend, e.g., for retrieving all artifacts of a certain branch that have been modified after a certain date. By permitting queries to be passed to and executed at the persistence backend, multiple interaction with the repository can be avoided, resulting in higher performance.

Table I
INFORMATION RETRIEVAL OPERATIONS

Return Type	Operation Name
MObject	getMObject
MObject	getMObject.root
MObject	getMObject.original
MObject[]	getMObject.modified
MObject[]	getMObject.derived
DublinCore	getMObject.dc
MMainBranch	getMProject.mainBranch
MProject	getMTag.project
MBranch	getMTag.parent
MArtifact[]	getMTag.artifacts
MTag[]	getMBranch.tags
MTag[]	getMArtifact.tags
byte[]	getMArtifact.data
MArtifact	getMArtifact.latest
MModelRelation[]	getMModel.inRelations
MModelRelation[]	getMModel.outRelations
MModelElement[]	getMModel.elements
MModel	getMModelElement.model
MModel	getMModelRelation.src
MModel	getMModelRelation.dest
UUID[]	getMTag.artifacts.uuid

V. MODEL-AWARE SERVICES

Services can interact with the MORSE repository at runtime and can profit from its reflective functionalities. We call services that interact with MORSE *model-aware*. Below we illustrate how they can interact with the repository and what information they may expose. Also we will show how model-aware services can be used by other services and how they can be created.

A. Interaction with the Repository

Model-driven, model-aware services can retrieve the MORSE objects from which they have been generated. This is achieved by embedding the UUIDs of the objects into the services, such as the UUID of the build as well as UUIDs of corresponding models, model elements, or transformations. At runtime, the service can access these UUIDs and retrieve the MORSE objects from the repository. The model-aware service typically reflects on the information and applies some logic for its further execution or uses the information for performing model-transformations.

Figure 6 illustrates a sequence diagram of a model-aware service. After a project and MDD artifacts have been created and checked into the repository, a build for the project is initiated by a client. The builder service retrieves the artifacts and generates a model-aware service, weaving corresponding UUIDs into the code for traceability. Afterward, it is deployed to a Web service framework. Next, a client invokes the service and causes it to interact with the repository. For the models it has been generated from, it needs e.g., to discover and consider new model relations such as new annotations. Therefore it passes the

embedded UUIDs of its models to the information retrieval interface of the model repository and requests for the model relations (`getMModel.inRelations`). These are then retrieved and evaluated. Relevant model relations are identified and the related models are requested. Finally, the models are processed and a response is issued to the client.

Please note that the MORSE builder in Figure 6 uses the MORSE repository to obtain the models of the model-aware services in order to generate code for them. This sequence diagram hence illustrates how an MDD tool (in this case the generator of MORSE) can make use of the MORSE architecture in the same way as other components querying models, such as monitoring components.

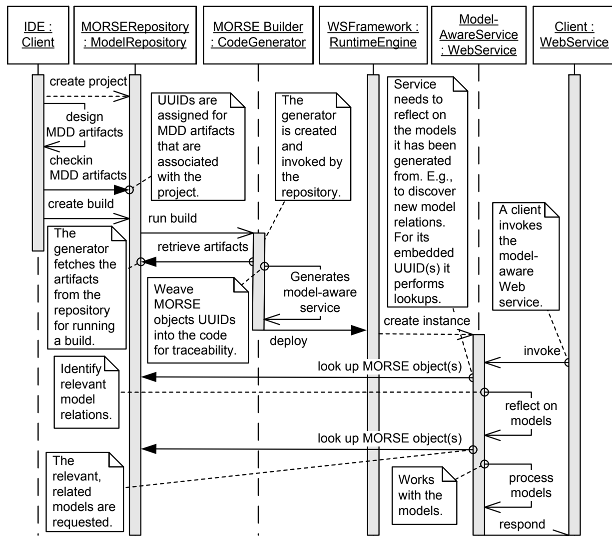


Figure 6. Sequence Diagram of a Model-Aware Service

B. Informational Operations

We have seen how model-aware services can interact with the repository, e.g., after invocation. Besides this, model-aware services may also offer information on them to other services (see Λ in Figure 1), i.e., disclose the MORSE objects they have been created from or are related to. For such model-aware services, we propose the operations displayed in Table II. The parameter passed to the operations specifies the type of the MORSE object in question, such as `MBuild`, `MProject`, or `MModel`. As a result, the service returns MORSE objects of the respective type. For example, in order to retrieve the models from which the model-aware service has been generated the `getMObject` operation is called with the parameter `MModel`. With the proposed operations it is possible to interrogate a model-aware service for its build, originating project, and MDD artifacts.

C. Integrating with Model-Aware Services

A service or process may integrate with and use model-aware services (see also B in Figure 1). A Model-aware service can support model-driven systems in the sense that

Table II
MODEL-AWARE SERVICE OPERATIONS

Return Type	Operation	Parameter
MObject[]	getMObject	type of MObject
UUID[]	getMObject.uuid	type of MObject

it can lookup and work with the MDD artifacts they have been generated from. During runtime, it receives events from these systems that contain MORSE identifiers and queries the model repository. For this kind of model-aware services, i.e., services that lookup MORSE objects for events they receive, we propose the operations displayed in Table III.

Table III
RECEIVE LOOKUP EVENT OPERATION

Return Type	Operation	Parameters
void	receiveMObject.uuid	UUID[]
MObject	getMObject	UUID

As an example let us consider processes with BPEL as a target technology for a process-driven SOA. Such processes can be generated from platform-independent, conceptual models [13]. The model repository manages the respective MDD projects and artifacts. In general, processes and process engines do not interact with the model repository or model-aware services. However, they can *integrate* with model-aware services in the sense that the latter receive events from the engine that hold the UUIDs of MORSE objects. Thus, these UUIDs have to be supplied as traceability information to the process.

For example, BPEL extensions provide a standard way to realize this. Listing 1 shows an excerpt of a BPEL process[2] with a BPEL extension for mapping code elements of the BPEL process to MORSE object identifiers. The `traceability` element, that indicates the UUID of the `build` as an attribute, is a sequence of `rows` that maps BPEL elements to the `uuids` of corresponding MORSE objects. The XML Path Query Language (XPath) [14] is chosen as the default query language for selecting the XML elements of the BPEL code. For extensibility, an optional `queryLanguage` attribute, that has the same semantics as in BPEL (cf. Section 8.2 of [15]), can specify an alternative query language or XPath version.

```
<process name="DeploymentProcess">
  <extensions>
    <extension mustUnderstand="yes"
      namespace="http://xml.vitalab.tuwien.ac.at/ns/morse/traceability.xsd"
    />
  </extensions>
  <import importType="http://www.w3.org/2001/XMLSchema"
    namespace="http://xml.vitalab.tuwien.ac.at/ns/morse/traceability.xsd"
    location="http://xml.vitalab.tuwien.ac.at/ns/morse/traceability.xsd"
  />
  <morse:traceability
     build="56810150-5bd8-4e8e-9ec5-0b88a205946b">
    <row query="/process[1]"
        queryLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0">
      <uuid>c6d2a636-747d-4c1b-8b7a-b32f59f0ac8c</uuid>
      <uuid>e4963cf9-f4d3-4f72-abe5-f3a4e2e26c30</uuid>
      <uuid>808ffa5d-d03e-465f-b931-0ada1d3b29d3</uuid>
      <uuid>5ba40ed1-3039-47c1-ba69-6c7a0362907a</uuid>
```

[2]For simplicity reasons most XML namespaces have been omitted.

```
    </row>
    <row query="/process[1]/sequence[1]/receive[1]">
      <uuid>d923339a-ef5d-455c-9fa7-8be23df55891</uuid>
    </row>
    <row query="/process[1]/sequence[1]/receive[1]/@variable[1]">
      <uuid>b52e218c-988e-418b-ad92-87aa533b1387</uuid>
    </row>
  </morse:traceability>
  <sequence>
    <!-- ... //-->
  </sequence>
</process>
```

Listing 1. BPEL Process with an Extension for MORSE Traceability

Note that this traceability information can annotate any XML based target code and can often be supplied as an inline extension[3] and does not have to be defined in a separate file. As a consequence, our approach is not limited to BPEL but can directly be applied to other XML and Web service based technologies and standards.

The traceability information can also be applied to programming languages such as Java or C#, e.g., as annotations to classes, interfaces, methods, and parameters. These annotations can be added to the source code (cf. [16]) or can be realized exogenously in an annotation file that decorates annotated classes.

In our BPEL example, during generation time, the MORSE builder weaves UUIDs of the MORSE objects into BPEL code (see A in Figure 1). At deployment time, the BPEL engine needs to support the BPEL extension, i.e., for the namespace that is used for the MORSE traceability extension, there is an implementation at the BPEL engine in place. At runtime, this extension submits events that contain the identifiers of e.g., the process or process activities, an event type, and optional further properties. Some events of interest are process instantiation and termination and pre-events and post-events for the execution of activities. Finally, the events are received by model-aware services that look-up the MORSE objects for e.g., monitoring, auditing, reporting, or business intelligence scenarios.

D. Creating Model-Aware Services

The MORSE builder supports the generation of the presented types of model-aware services, i.e., it creates WSDL[4] interfaces and Java implementations, as follows:

- For any build of a MDD project, a dedicated, standalone model-aware Web service can be generated that provides information on the build, the project, and its artifacts. For the endpoint of such a Web service we propose a Uniform Resource Identifier [18] that ends with `/MBuild` as a naming convention.
- For generating model-aware services, templates can be reused in projects for extending the interface with the desired operations, for embedding the UUIDs to the service, and for generating the service requester implementation for interacting with the repository.

[3]Supposed that such extensibility is provided with an `any` element in the XML schema.

[4]Web Services Description Language [17]

For services and processes that rely on model-aware services, UUIDs need to be supplied for services or processes as well. During generation time, the MORSE builder creates a traceability mapping that can be embedded into XML documents as demonstrated. Although we have shown an integration with model-aware services using model-driven BPEL processes, this approach can similarly be applied to different technologies and frameworks, e.g., message interceptors for Web services.

VI. CASE-STUDY

In order to demonstrate the applicability of the presented approach, we explain a case study. In this case study a European telecommunication company offers rich multimedia services to customers. Particularly, the company's customer can subscribe to content such as video or audio streams and files, i.e., the customer can, e.g., download tracks from music albums and watch movies.

In this context, licensing information on the content constitutes a crucial issue: (under which conditions) is the customer allowed to access a resource? Often, e.g., broadcasting content can only be obtained if a user executes the request to access such content within the country of the broadcasting company. Other possible restrictions are the contract and status of the customer. Similarly, payment depends on various factors such as the price of the requested content, the conditions of the customers contract and/or of effective special offers.

The telecommunication company decided to apply MDD technologies for its services. Therefore, it designed and uses various models, e.g., for licensing and payment information. When the company modifies its business models for multimedia content, e.g., introduces a special offer, or if it changes the licensing information, it creates or modifies models accordingly. For enabling runtime services to dynamically work with these models, the company employs MORSE, i.e., models are stored within the model repository and the SOA contains model-aware services that interact with the repository. As a result, the company is able to address the following issues:

- The current price for the content as well as effective conditions such as originating from valid special offers or the customers contract conditions are considered for calculating the price. A special offer can simply be introduced as a new model-relation, i.e., the service does not have to be modified or redeployed.
- Access is granted as specified in the effective licensing model.
- For analyzing customer services the corresponding processes are monitored and related to their originating models.

A `ConsumeMultimediaProcess` is initiated when the user wants to download some multimedia content. This process invokes an `AccountingService` and a `LicensingController` service and returns detailed information on payment and licensing to the user for acceptance. Both services retrieve the model instances from the repository and apply an algorithm. If necessary,

also the algorithms can be stored as models and can dynamically be retrieved and executed by the services. We will demonstrate such a model-aware service in more detail by focusing on the payment service.

Besides this orchestration, the process also notifies a monitor by transmitting the UUID of the model from which it has been generated. From the process models BPEL code is generated with a BPEL extension that notifies the monitor at invocation, as explained in Section V-C. A monitor collects this information from various processes and process versions and correlates them for generating statistical reports (containing e.g., process versions, number & time of invocations, and duration).

After the `AccountingService` is invoked, it calculates a price with the effective payment and content models according to the conditions of the contract and of effective special offers. The models are retrieved from the MORSE repository (see also Figure 6). Figure 7 shows models that are processed by Algorithm 1 for calculating the price. Besides customer, content, and purchase information, the models store conditions of a contract and optionally of special offers. For calculating the price, first the effective conditions are determined by applying present special offers. If the flatrate condition is valid, the customer will not be charged for the download. Similarly, a customer may download a content that he already retrieved within the last day for free. Otherwise, he will be charged the price of the content by considering a discount if he exceeds his free download volume.

Algorithm 1: Payment Algorithm

Input: r ∈ Content, c ∈ Contract
Output: price ∈ Price

```
1  begin
2      cc ⟵ c.conditions;
3      for special ∈ c.subContracts→forAll(sc|isValid(sc)) do
4          applyConditions(special.conditions, cc);
5      if cc.flatrate then
6          return 0;
7      if ∅ ≠ c.purchases→forAll(p|p.date+24h>now() ∧ p.content=r) then
8          return 0;
9      if getTotalDownloadVolume(c) < c.conditions.freeDowloadVolume
       then
10         return 0;
11     else
12         return r.price ∗ cc.discountFactor;
13 end
```

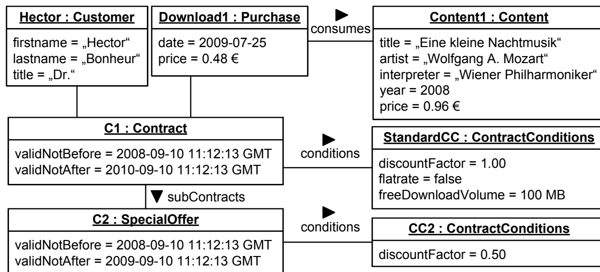


Figure 7. Payment Model Instances

After the `ConsumeMultimediaProcess` determined the licensing conditions and calculated the price, the customer is informed and can decide to accept the terms for eventually purchasing the multimedia content. In case access is denied as caused by some licensing condition, the user can be provided with detailed information for understanding the reason. Similarly, he can retrace the payment information by reflecting on the models.

VII. RELATED WORK

ModelBus [19] is a model-based tool integration framework that, like the MORSE repository, aims to support MDD. It addresses the heterogeneity and distribution of model tools and realizes transparent model update. Designed as an open environment, ModelBus focuses on integrating functionality such as model verification, transformation, or testing into a service bus. The MORSE repository, with another focus, comes with explicit support for the management of MDD projects. Workflows, that cover processes of MDD, have to be defined on top of ModelBus. In contrast, MORSE focuses on runtime services and processes and their integration and interaction with the repository.

ModelCVS [20] aims at model-based tool integration. ModelCVS and the MORSE repository unlike ModelBus have a centralized information management in common. Instead of aiming at reconciling a multitude of modeling tools' languages and the integration of arbitrary (legacy) tools, MORSE concentrates on some selective concepts such as relations between models.

Odyssey-VCS 2 [21] is an EMF based model repository after initially relying on the NetBeans Metadata Repository [22]. In contrast to this and most model repositories, the MORSE repository abstracts from technologies, focuses on MDD projects, and targets at integration with services. Apart from the model repository, MORSE with its model-aware services establishes a service-oriented approach that has not yet been presented at large.

In contrast to the mentioned model repositories and model-based tool integration frameworks, Moogle [23], a model search engine, realizes an inverse approach of indexing and potentially managing models. It allows for complex queries and can help finding relevant models for MDD projects during design time. As such it is not (yet) suited for our purposes that target runtime systems.

VIII. SUMMARY

We have presented MORSE, the Model-Aware Service Environment, for facilitating services to dynamically reflect on models. For this purpose we have proposed a model repository that manages MDD projects and supports the identification of and reflection on models, model elements, and model relationships. Moreover, we have presented model-aware services that interact with the model repository for dynamic information retrieval on models and MDD projects. Also, we have demonstrated how services and processes can integrate with model-

aware services, i.e., how traceability information is introduced and transmitted. Finally, we have showcased our contributions with a case study.

While in this work we focused on model-aware services and their combination with other services, e.g., for monitoring purposes, the presented services of the model repository are not only of interest for the runtime but also for design time components. With appropriate tool-support, the MORSE repository constitutes a common space for developers in a distributed environment for storing and managing MDD projects and models that can be consulted by the runtime, i.e., model-aware services. This use of MORSE was demonstrated when we illustrated the internal architecture of MORSE in Section V (see Figure 6).

ACKNOWLEDGMENTS

This work was supported by the European Union FP7 project COMPAS, grant no. 215175.

REFERENCES

[1] *MDA Guide Version 1.0.1*, http://www.omg.org/cgi-bin/doc?omg/03-06-01, June 2003, [accessed in July 2009].

[2] M. Völter and T. Stahl, *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, 2006.

[3] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Comput. Surv.*, vol. 37, no. 4, pp. 316–344, 2005.

[4] E. Oberortner, U. Zdun, and S. Dustdar, "Domain-specific languages for service-oriented architectures: An explorative study," in *ServiceWave*, ser. Lecture Notes in Computer Science, P. Mähönen, K. Pohl, and T. Priol, Eds., vol. 5377. Springer, 2008, pp. 159–170.

[5] S. Stein, S. Kühne, J. Drawehn, S. Feja, and W. Rotzoll, "Evaluation of OrViA framework for model-driven SOA implementations: An industrial case study," in *BPM*, ser. Lecture Notes in Computer Science, M. Dumas, M. Reichert, and M.-C. Shan, Eds., vol. 5240. Springer, 2008, pp. 310–325.

[6] A. Bercovici, F. Fournier, and A. J. Wecker, "From business architecture to SOA realization using MDD," in *ECMDA-FA*, ser. Lecture Notes in Computer Science, I. Schieferdecker and A. Hartman, Eds., vol. 5095. Springer, 2008, pp. 381–392.

[7] P. Mayer, A. Schroeder, and N. Koch, "MDD4SOA: Model-driven service orchestration," in *EDOC*. IEEE Computer Society, 2008, pp. 203–212.

[8] International Telecommunication Union, *ISO/IEC 9834-8 Information technology – Open Systems Interconnection – Procedures for the operation of OSI Registration Authorities: Generation and registration of Universally Unique Identifiers (UUIDs) and their use as ASN.1 object identifier components*, http://www.itu.int/ITU-T/studygroups/com17/oid/X.667-E.pdf, September 2004, [accessed in July 2009].

[9] G. Booch and A. W. Brown, "Collaborative development environments," *Advances in Computers*, vol. 59, pp. 1 – 27, 2003.

[10] ISO, *ISO/IEC 19501:2005 Information technology – Open Distributed Processing – Unified Modeling Language (UML), v1.4.2*, http://www.omg.org/cgi-bin/doc?formal/05-04-01, April 2005, [accessed in July 2009].

[11] J. Kunze and T. Baker, "The Dublin Core Metadata Element Set," http://www.ietf.org/rfc/rfc5013.txt, The Internet Engineering Task Force, Request for Comments, August 2007, [accessed in July 2009].

[12] openArchitectureWare.org, "openArchitectureWare," http://openarchitectureware.org, openArchitectureWare.org, [accessed in July 2009].

[13] H. Tran, U. Zdun, and S. Dustdar, "View-based and model-driven approach for reducing the development complexity in process-driven SOA," in *Intl. Working Conf. on Business Process and Services Computing (BPSC'07)*, ser. Lecture Notes in Informatics, vol. 116, sep 2007, pp. 105–124.

[14] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon, "XML path language (XPath) 2.0," http://www.w3.org/TR/xpath20/, W3C, W3C Recommendation, January 2007, [accessed in July 2009].

[15] "Web service business process execution language version 2.0," http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html, OASIS Web Services Business Process Execution Language (WSBPEL) TC, OASIS Standard, January 2007, [accessed in July 2009].

[16] A. Buckley, "A metadata facility for the Java™ programming language," http://www.jcp.org/en/jsr/detail?id=175, Sun Microsystems, Inc., Java Specification Requests, September 2004, [accessed in July 2009].

[17] "Web services description language (WSDL) version 1.1," http://www.w3.org/TR/wsdl, W3C, W3C Note, March 2001, [accessed in July 2009].

[18] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform resource identifier (URI): Generic syntax," http://www.ietf.org/rfc/rfc3986.txt, The Internet Engineering Task Force, Request for Comments, January 2005.

[19] P. Sriplakich, X. Blanc, and M.-P. Gervais, "Supporting transparent model update in distributed case tool integration," in *SAC*, H. Haddad, Ed. ACM, 2006, pp. 1759–1766.

[20] G. Kramler, G. Kappel, T. Reiter, E. Kapsammer, W. Retschitzegger, and W. Schwinger, "Towards a semantic infrastructure supporting model-based tool integration," in *GaMMa '06: Proceedings of the 2006 international workshop on Global integrated model management*. New York, NY, USA: ACM, 2006, pp. 43–46.

[21] L. Murta, C. Corrêa, a. G. P. Jo and C. Werner, "Towards Odyssey-VCS 2: Improvements over a UML-based version control system," in *CVSM '08: Proceedings of the 2008 international workshop on Comparison and versioning of software models*. New York, NY, USA: ACM, 2008, pp. 25–30.

[22] M. Matula, "NetBeans metadata repository," http://mdr.netbeans.org, NetBeans Community, [accessed in July 2009].

[23] D. Lucrédio, R. P. de Mattos Fortes, and J. Whittle, "MOOGLE: A model search engine," in *MoDELS*, ser. Lecture Notes in Computer Science, K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, Eds., vol. 5301. Springer, 2008, pp. 296–310.