Model-Driven Composition of Information Systems from Shared Components and Connectors

Stefania Leone[1,⋆], Alexandre de Spindler[2], and Dennis McLeod[1]

[1] Semantic Information Research Laboratory,
Computer Science Department, USC Los Angeles, CA, 90089-0781, USA
{stefania.leone,mcleod}@usc.edu
[2] School for Management and Law, ZHAW
CH-8400 Winterthur, Switzerland
alexandre.despindler@zhaw.ch

Abstract. We introduce CompIS, an approach, model and platform for model-driven component-based information system engineering. Our approach is based on the concept of shared components and connectors between them. To address the data-intensive nature of information systems, our components follow an extended model-view-control structure that also includes data. Component composition is based on configurable connectors, which define the collaboration logic between components and support component composition at the level of the component model, view, control and data. The CompIS UML profile allows to graphically define new components, connectors and compositions. The CompIS platform is a model-driven engineering environment, based on an extended object database that natively integrates the CompIS model. From graphical UML model definitions, the platform automatically generates application code that creates and initialises components and connectors. We present and validate our approach in the eCommerce domain.

Keywords: information system engineering, component model, model-driven engineering.

1 Introduction

Requirements towards an information system are subject to continuous change and evolution. For example, a company's initial version of their eCommerce solution may support standard online store functionality by means of product, customer and order management. Over time, they may desire to integrate support for electronic payment, extend the customer experience with support for product ratings and product recommendation, or may require more sophisticated product management functionality.

⋆ Stefania Leone's work is supported by the Swiss National Science Foundation (SNF) grant PBEZP2_140049.

While modern software engineering advocates a modular design and flexible development process to cater for such evolving requirements, information systems are still designed and developed in a rather monolithic and sequential way, where adaptation and evolution are not inherently supported. More recently, however, so-called community-driven development approaches [1] have become a popular way of providing more configurable and extensible information system platforms. In the eCommerce domain, popular open-source platforms, such as Magento Commerce[1] or osCommerce[2] follow this style of architecture. Their core, providing fully-fledged online-store functionality, can be modified, specialised and extended through extensions developed by and shared with the community. Popular Magento Commerce extensions offer support for payments via bank transfers[3] or integrate a blog[4]. While their extension mechanisms is flexible and powerful, extensions represent isolated units that only extend the core. There is no inherent mechanism that allows for flexible and modular composition scenarios, where one extension could be composed with another. For example, an extension offering advanced product management could not easily be combined with a product rating extension. The developer would rather have to get familiar with the code of both extensions and implement the composition code. In fact, these platforms typically do not even provide a systematic approach to the development of extensions and developers need to get familiar with the inner workings of the platform.

In this paper, we combine the approach taken by community-driven extensible information system platforms with ideas of component-based software engineering (CBSE). In contrast to CBSE, our component model is specifically targeted towards the data-intensive nature of information systems. At the same time, our approach overcomes the sequential nature of database design by introducing a modular and agile design process, where components may be composed at the model, view, control and data level, thus introducing modularity into the database. We build on a preliminary approach presented in [2], where we integrated support for component-based Web engineering into content management systems. The current work generalises our previous work, and introduces a model-driven approach to component-based information systems engineering.

The contributions of this paper consist of a refined, general component model for component-based information system engineering based on components and explicit connectors between them, a UML [3] extension that supports the model-driven engineering of components, connectors and compositions and our prototypical CompIS platform, which generates composed information systems from user-defined UML models and deploys it onto an extended object database that natively integrates the concept of components and connectors.

The paper is structured as follows. In Sect. 2, we present the background of our work. Section 3 introduces the general approach, followed by the design

[1] www.magentocommerce.com
[2] www.oscommerce.com
[3] www.magentocommerce.com/magento-connect/bankpayment.html
[4] www.magentocommerce.com/magento-connect/blog-community-edition.html

process in Sect. 4. In Sect. 5, we introduce the CompIS component model and in Sect. 6 we present the CompIS UML profile. The model-driven CompIS platform is presented in Sect. 7 and we give concluding remarks in Sect. 8.

2 Background

To facilitate the software design and evolution, CBSE [4, 5] postulates a modular and systematic construction of software from reusable components that can be extended, adapted and replaced. The cornerstone of CBSE is the underlying component model that defines how components are specified, constructed, assembled and deployed [6]. A component typically exhibits a set of functions and data through a well-defined interface and hides implementation details. Service-oriented architecture (SOA) [7] can be seen as a continuation of CBSE, explicitly addressing the requirements of loosely coupled, standards-based, and protocol-independent distributed computing. Services are reusable, self-contained, autonomous units with a well-defined interface, and are capable of communicating with each other via messages. Services are published to a repository by service providers and can be retrieved and consumed by service consumers.

While CBSE and SOA only offer composition at the message level, application composition may also take place at other levels. For example, a recent work in the area of web service composition proposed to also include the distributed orchestration of user interfaces (UI) [8]. They focus on extending web service standards such as WSDL [9] and BPEL [10] to integrate support for so-called UI components representing complete web applications. The extended orchestration supports the distributed synchronisation of UI components and services.

There have also been proposals for component-based database design, e.g. [11–13]. Thalheim [11] proposes the use of composable subschemas and other meta-structures to support the modelling and management of large and complex database schemas. Such components are database schemas that exhibit a similar structure and define import and export interfaces for connecting them. They also support incremental database development between versions of databases of different development phases through a mechanism that supports the importation of data from one version of a database into another, either as read-only or fully modifiable, in order to support incremental system design [14]. In our previous work [13], we showed, how structural composition can be supported through native constructs of the data modelling language.

Our current work combines composition approaches at various levels and introduces a modular and flexible approach to the development of information systems. We offer support for component-based information system engineering, where components can be composed at various levels, depending on the nature of the actual composition scenarios. We support, for example, the composition of a currency converter component with an order component at the level of the component view to integrated currency conversion into the user interface of the order component. At the same time, an electronic payment component can be composed with an order component at the message level to invoke the payment process upon order completion, while a product and rating component is structurally composed at the model level to allow products to be rated.

As stated in [15], one of the main challenges of modular system development lies in the fact that modular units may not be compatible for composition. As a consequence, the CompIS model is inspired by the Architecture Description Language (ADL) [16, 17], an approach to CBSE, where the component model consists of components and explicit connectors between them. Through the definition of explicit connectors between components, we circumvent the problem of component incompatibility. Connectors encapsulate the composition logic, exhibiting functionality ranging from simple message passing, to complex collaboration logic, such as data transformation operations, and, consequently, would allow for the composition of arbitrary components.

In line with model-driven engineering approaches for CBSE and SOA, e.g. [18–20], we have realised the CompIS model as a UML extension that refines the UML component and class models with a number of stereotypes representing the concepts defined by the CompIS model. Developers design new components and connectors using the CompIS stereotypes. We also provide a set of default connectors that can be reused and configured through model refinement and by means of Object Constraint Language (OCL) expressions [21] to realise a specific composition scenario. From the UML model definitions, our CompIS platform automatically generates application code that configures default connectors for particular compositions, and also generates and instantiates newly defined components and connectors. The generated code is deployed onto an extended object database that natively integrates our component model. By using an object database, components and connectors are defined and handled uniformly, as objects exhibiting data, structure and application logic.

Note that in contrast to model-driven Web engineering approaches, such as WebML [22], which prvide models for specifying the structure, navigation and presentation of web applications, we focus on component-based information systems engineering in general. We provide an approach to compose information systems in a model-driven way from components, representing small application units defining a model, view, control and data, and configurable connectors between them.

3 Approach

Our approach is motivated by the need to develop information systems in the context of complex and evolving requirements. For example, the development of an eCommerce platform may start with basic support for products that can be ordered by customers. In a second iteration, the system may be further extended with additional shipment and payment options, followed by advanced user experience support in the form of product ratings and recommendations.

Information system components targeting such individual requirements may already exist, either within the company's information system infrastructure or publicly available, for free or purchase. For example, a product supplier may share a product management component offering management of and access to their products. Therefore, it is of utmost importance to offer developers means and methods to construct their systems iteratively, ideally from shared components, to respond to complex and evolving requirements. Our approach targets such scenarios and supports the model-driven composition of information systems from reusable components and connectors between them. We will first give an overview of the CompIS model before we show, based on an eCommerce example, how components can be composed in a model-driven manner.

The CompIS component model is inspired by ADL, defining compositions by means of explicit connectors between components. Connectors enable composition and define the collaboration logic between components. To address the data-intensive nature of information systems, the CompIS component structure follows an extended MVC pattern that includes data. As a consequence, composition may take place at the level of the model, view, control or data. At all four levels, a component may expose so-called connection points used for compositions by the connectors. Figure 1 shows two example compositions in the

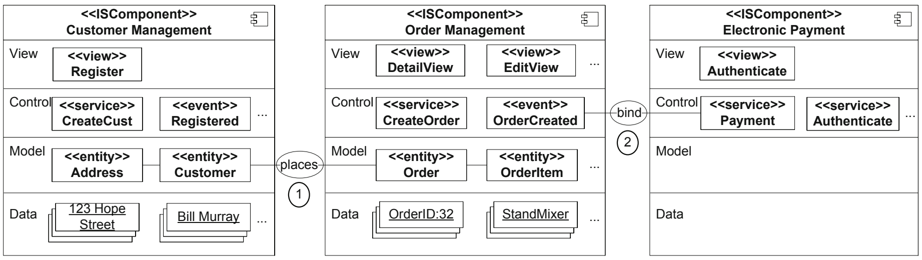


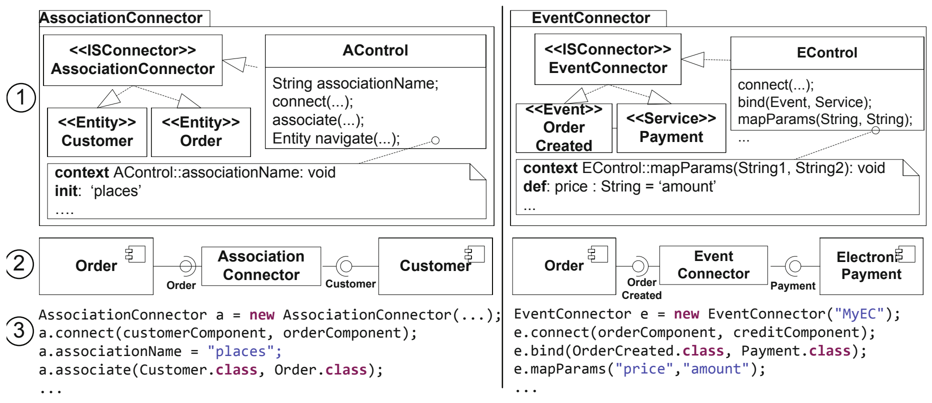
Fig. 1. Composition Example

eCommerce domain. Components are represented using the white box view of the UML component model, stereotyped with `<<ISComponent>>`. At each level, a component may expose connection points. For example, the customer management component on the left, exposes the `Register` view for composition at the view level, the service `CreateCust` and the event `Registered` for control level composition, the entities `Address` and `Customer` for composition at the model level, and the set of all `Customer` and all `Address` objects for composition at the data level. Similarly, the order management component, shown in the centre, offers a number of connection points at each level, while the electronic payment component, shown on the right, only offers connection points at the control and view level.

In (1), a composition at the model level is shown, where an association `places` is created between the `CustomerManagement.Customer` entity and the `OrderManagement.Order` entity. In (2), a composition at the control level is shown, binding the invocation of the `ElectronicPayment.Payment` service to the `OrderManagement.OrderCreated` event.

In Fig. 2, we show the steps involved in realising these compositions. On the left, the customer management and the order management components are composed through an association connector, and on the right, the event connector composes the order management and electronic payment components. Both connectors are default connectors provided as part of our model. They can be configured by the developer to define a specific composition. For the composition, we make use of refined UML component and class models.

The UML component model represents software systems by means of components and their relationships. A component's behaviour is specified in terms of provided and required interfaces. Composition is realised by connecting components over required and provided interfaces that match. Components are realised through explicit classes defined in the scope of an associated class model that defines a component's inner workings, such as the implementations of the provided interfaces. In analogy to this definition, CompIS components and



```
AssociationConnector a = new AssociationConnector(...);
a.connect(customerComponent, orderComponent);
a.associationName = "places";
a.associate(Customer.class, Order.class);
...
```

```
EventConnector e = new EventConnector("MyEC");
e.connect(orderComponent, creditComponent);
e.bind(OrderCreated.class, Payment.class);
e.mapParams("price","amount");
...
```

Fig. 2. Connector Configuration

connector are specified by means of a package where components and connectors are represented as UML components with an associated class model defining the realisation of the component. To configure a connector, the developer first refines its class model (1). If a connector composes components through *required* connection points, the developer associates the connector to those connection points via a `usage` association. In cases where a component is composed over a *provided* connection point, the developer has to realise the required interface as part of the connector definition, by means of a class that implements the methods and attributes defined by the connection point.

The composition itself is defined through an additional component model (2), modelling the composition by means of the two components and a connector between them. The composition is defined using the compact component view, where connection points are represented as interfaces. The UML socket notation stands for required and the lollipop notation for provided connection points. This implies that a component provides a connection point required by the connector and vice versa. Finally, from these two models, the code generator outputs initialisation code that configures the connector for the specific compositions (3).

The association connector, shown on the left in Fig. 2, creates an association between `Entity` connection points of two different components and, upon instantiation, allows to associate objects across components and to navigate along the association from one object to another. The association connector is represented by the `AssociationConnector` package. The `AControl` class realises the `AssociationConnector` defining its functionality. To define the composition, the association connector class model is refined. The `AssociationConnector` uses the two connection points, defined in the scope of their respective components, specified through a `usage` associations between the `AssociationConnector` and the `Customer` and `Order` entities. Note that the connection points are shown in the scope of the connector's package for the sake of this example, but are defined in the scope of their respective components' class models. `AControl` is further configured through user-defined OCL expressions. The OCL constraint shown in the example has been defined in the context of the `AControl.associate` method and consist of an attribute initialisation specifying the `name` of the association as 'places'. In a second step, the developer creates the component model, shown below the class model, which realises the composition. The association connector has been configured to require the `Customer` and `Order` connection points provided by the order management and customer management components. The code snippet, at the bottom left, defines the instantiation of the association connector and the invocation of the `associate` method with the defined association name and the two entity connection points passed as Java classes.

On the right of Fig. 2, the event connector configuration and instantiation is shown. The event connector handles the binding of an event of one component to a service of another component and takes care of the event registration and service invocation. The event connector class model is configured to require the connection points `OrderCreated` event from the order management component and `Payment` service from the electronic payment component, expressed by `usage` associations. The `EControl` class realising the `EventConnector` is further configured through OCL constraints. The OCL constraint associated with the `EControl.mapParams` method defines a variable definition `price = 'amount'` specifying the mappings between an event object attribute and a service parameter. Below, an excerpt of the initialisation code is illustrated, where the event connector is instantiated, and the `bind` method is invoked, passing the event and service as Java classes. Note that service classes define a method `invoke` which triggers the invocation of the service. The parameter mapping is based on OCL variables that map attributes to parameters by name. Upon service invocation, we make use of reflection to realise the actual mapping.

Note that developers are free to further extend the connectors with additional functionality. For example, the event connector may also define a user interface that redirects the user from the order to the payment process and back.

4 Component-Based Information System Engineering

Having introduced our approach, we will explain in more detail, how developers build their information systems in a model-driven and iterative manner from

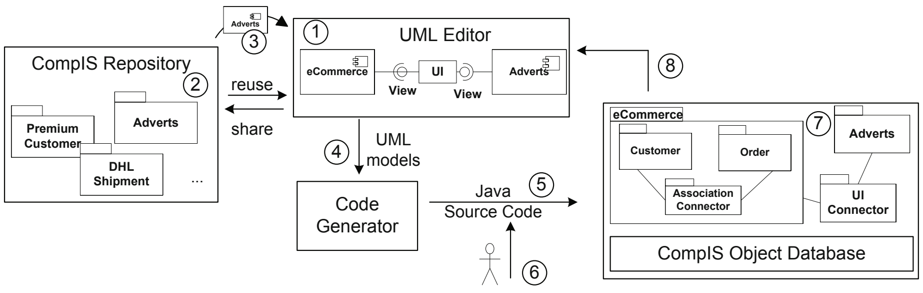


Fig. 3. CompIS Model-Driven Development Process

shared components and connectors. Figure 3 gives an overview of the CompIS development process and the involved components of the CompIS platform. A developer models and composes an information system by means of UML models using the UML editor provided by the CompIS platform (1). Our approach follows a community-based development paradigm. Components may either be designed from scratch by the developer, or imported from the community repository (2), where shared components and connectors are offered for reuse. During the design process, developers may browse the repository and import shared components and connectors into their local editor to compose them with other components. In the current example, we assume that the developer composes a local eCommerce component with a newly imported adverts component (3) using a UI connector.

Once the developer has modelled the application, the UML model descriptions are passed to the code generator (4). From the model description, the code generator generates Java source code realising the composed information system (5). Note that imported components and connectors consist of UML models and associated source code. Therefore, for the current composition, only the initialisation code of the imported adverts component and the UI connector has to be generated. However, if a developer designs new components and connectors from scratch, we generate the source code realising the component or connector structure, as well as initialisation code. Application logic not expressible via UML, such as method bodies, has to be manually added to the generated source code by the developer (6). To facilitate this task, the source code files are marked with annotations, where manual coding is needed.

The generated Java source code is deployed and runs on top of the CompIS object database (7). The CompIS object database is an extended object database that builds the basis for the deployed code providing native persistence for components, connectors and their data. The currently deployed application consists of a hierarchically composed eCommerce component that has been structurally composed from an order and customer component using an association connector. In the current iteration, the eCommerce component has in turn been composed with the advert component using a UI connector. The last composition results in an extended eCommerce main view that includes adverts.

As seen in the running example, such composition scenarios are typically the result of iterative development steps addressing new and evolving requirements. To cater for such evolving requirements and support continuous evolution, at any point in time, the developer may further extend or adapt the deployed information system by starting a new modelling iteration ⑧. In every iteration, the deployed application can be extended with new components or existing ones can be adapted or replaced. Also, developed components can be shared through the community repository. In this way, information systems are constructed modularly and collaboratively, from shared components and connectors.

5 Component Model

The CompIS component model supports the component-based design of information system. The model primitives have been designed in such a way that components represent fully functional (sub-) information systems and may be composed to build more complex information systems. Figure 4 gives an overview of our component model by means of a metamodel using UML notation. The part shaded in grey, highlights the basic component structure.

The component structure follows an extended MVC pattern that includes data. Each component may define a model, i.e. the data structure, a view defining the user interface, a control defining the application logic, and data structured according to the defined model, illustrated by the aggregation between `Component` and the corresponding parts. Note that components do not have to define all parts, but could only define a model and control, but no data and view.

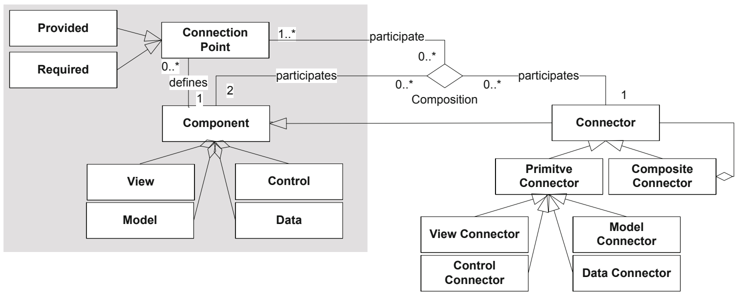


Fig. 4. Component Metamodel

To enable composition, each component defines connection points. In analogy to CBSE, we provide two types of connection points: required and provided connection points. A provided connection points represents a utility a component offers to the rest of the system. They can be seen as a signature of the component and there is no need to know about the inner workings of the component in order to make use of it. If a component needs to use utilities provided by another component in order to function, it defines required connection points. For example, an order management component may not include support for order shipment, but define a shipment interface connection point to be implemented by a third-party. A DHL shipment component may realise the required functionality and be composed using an appropriate connector that realises the collaboration logic. While components that only expose provided connection points are self-contained applications that run independently, components with required connection points need external help/support in order to function. It is therefore recommended to make sparse use of required connection points, or to opt for a different modularisation to reduce dependencies between components and thus, reduce complexity.

Figure 5 gives an overview of the currently supported connection points, grouped by level. Required connection points are shaded in grey. Note that required and provided connection points could be situated at any level. At the view level, `View` connection points represent composable view elements that expose a certain functionality or data through their user interface.

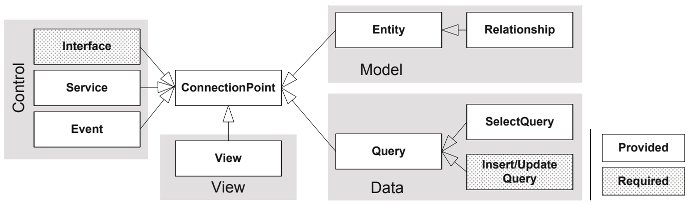


Fig. 5. Connection Point UML model

At the control level, we offer three types of connection points. `Event` connection points represent an event within a component, upon which another component can react. The event object gives access to the type of event and the involved (data) object(s). `Service` connection points define functionality offered by a component that can be invoked externally. `Interface` connection points are the counterparts of services and require an external component to provide the functionality defined by their signature of required methods and attributes.

At the model level, we rely on the ER model and its modelling constructs as connection points. `Entity` connection points give access to the structure of component data, i.e. the attributes and methods, and `Relationship` connection points represent relationships between entities. Note that a relationship is a specialisation of entity that defines a source and target attribute and methods allowing to navigate from one entity instance to a related one.

Finally, data connection points are represented by queries over the model. Such queries may be represented by SQL queries in a relational world, by XQuery expressions in an XML world, or code realising navigational data access in case of object databases. There are two types of query connection points: `Select Query` connection points give access to data, and thus are provided connection points, while `Insert/Update Query` connection points take external data as input, and thus are required connection points. To make use of a component's data in another component, a connector composes two components using a provided and required connection point, to export data from one component via select query and import it into the component using an insert/update query.

As illustrated in Figure 4, components may be composed through their connection points by explicit connectors, shown on the right. Depending on the connection points used for composition, a connector may be a data connector, a model connector, a control connector or a view connector. An association connector, for example, is a model connector, since it composes components at the model level by associating entity connection points. The connector, however, may define much more than a simple association between two entities at the model level. The connector may, for example, realise operations at the control level that allow to create, update and delete associations between entities as well as to navigate between entities along the associations. The connector may even define a view that allows such associations to be graphically created, and finally, also manage association data. Consequently, connectors may define a model, view, control and data, and are therefore defined as a specialisation of components, as shown by the specialisation association between `Component` and `Connector`.

We support composition of both components and connectors, since connectors are in turn components. The result of a composition is a new component (in fact, a connector), consisting of two sub-components plus newly defined component elements. For example, the eCommerce component has been composed from an order management and customer management component. The view of the eCommerce component is defined by the union of view elements defined by the respective components plus newly defined view elements. Generally, the composition interface of a composed component is built by the union of connection points defined by the sub-components plus newly defined ones.

Connectors may also be hierarchically composed, as illustrated by the composite pattern for connectors. For example, a composite connector between an eCommerce component and a recommendation component may consist of a data connector that composes the transformed eCommerce product and order data required by the recommendation component, and a UI connector which integrates recommendations into the eCommerce product overview view. Such composite components allow for arbitrary complex composition scenarios, where a composition may compose `n` sub-components.

To facilitate composition, we provide a set of default connectors that can be configured to adhere to a particular composition scenario. For each level, we provide default connectors tailored to compose components based on specific types of connection points. Such default connectors specify which types of connection points may be composed, and whether they are required or provided. Figure 6 gives an overview of the default connectors.

At the view level, we provide the composite connector that integrates two provided views into a composite view. With Java Swing, such views would

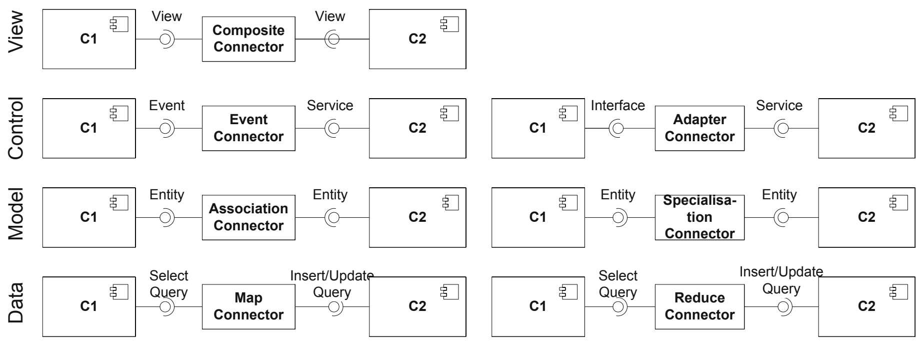


Fig. 6. Connectors

correspond to `JPane` instances that can be nested. In a web-based environment, a view may corresponds to a widget or `div` element in an HTML document.

At the control level, we provide the event connector, presented previously, that binds an event to a service, and invokes that service upon the event. The adapter connector is a realisation of the adapter design pattern. The connector connects components over an interface and a service connection point and forwards method invocation from one component to another. The connector translates internal calls to component A's interface into calls to component B's service, thus realising an implementation of interface A through service B.

At the model level, we rely on structural composition based on the underlying model's modelling concepts [13]. The association connector defines an association between two entity connection points and the specialisation connector defines an `isA` relationship between two entities. Since relationships are specialisations of entities, these two connectors may also act between relationships, thus allowing for n-ary relationships.

Finally, data connectors allow data from one component to be reused by another component. Data reuse may be defined by a mapping connector that maps the data structured according to one component's schema to the schema of another component, or by a reduce connector that transforms data from one component to a format specified by another component, thus enabling data reuse. Since connectors can be nested, data connectors can generally be defined as combinations of map and reduce functions.

As seen in Sect 3, default connectors are configured by extending and refining their class model. The connection points are specialised through usage or realisation associations from the connector class to the respective connection point classes, and through OCL constraints that further specify the composition.

6 CompIS UML Profile

UML 2.0 offers *UML Profiles* as a powerful extension mechanism that allows to extend, tailor or specialise UML to a specific domain [23]. Profiles are defined by

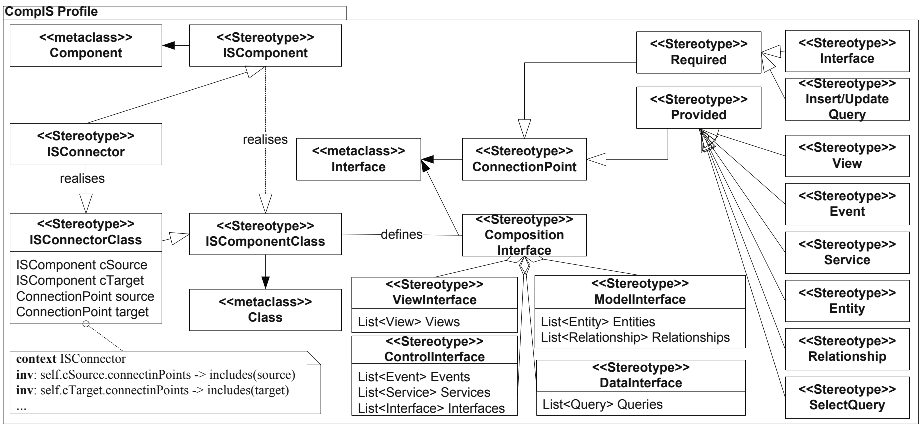

CompIS Profile
<<metaclass>> Component
<<Stereotype>> ISComponent
<<Stereotype>> ISConnector
realises
realises
<<Stereotype>> ISConnectorClass
ISComponent cSource
ISComponent cTarget
ConnectionPoint source
ConnectionPoint target
<<Stereotype>> ISComponentClass
<<metaclass>> Class
context ISConnector
inv: self.cSource.connectinPoints -> includes(source)
inv: self.cTarget.connectinPoints -> includes(target)
...
<<metaclass>> Interface
<<Stereotype>> ConnectionPoint
defines
<<Stereotype>> Composition Interface
<<Stereotype>> ViewInterface
List<View> Views
<<Stereotype>> ControlInterface
List<Event> Events
List<Service> Services
List<Interface> Interfaces
<<Stereotype>> ModelInterface
List<Entity> Entities
List<Relationship> Relationships
<<Stereotype>> DataInterface
List<Query> Queries
<<Stereotype>> Required
<<Stereotype>> Provided
<<Stereotype>> Interface
<<Stereotype>> Insert/Update Query
<<Stereotype>> View
<<Stereotype>> Event
<<Stereotype>> Service
<<Stereotype>> Entity
<<Stereotype>> Relationship
<<Stereotype>> SelectQuery

Fig. 7. CompIS UML Profile

means of a UML model that specifies new model elements as so-called stereotypes that define properties and operations. Stereotypes can be used to introduce a new terminology, new syntax, new semantics and constraints. For example, a stereotype may refine and further constraint the definition of a UML type.

Figure 7 illustrates the CompIS UML profile. UML profiles are grouped as a package, in our case the CompIS Profile package. Our profile extends and refines concepts of the UML component model and class model. First, we introduced a new stereotype `ISComponent` that refines the UML `component` metaclass, as illustrated by the extension relationship between the two concepts.

`ISComponent` is realised through the `ISComponentClass`, which extends the UML `class` metaclass. Connection points are represented by the `Connection-Point` stereotype, which is a specialisation of the UML `Interface` metaclass, and further specialised into `Required` and `Provided` connection points. We define one stereotype for each type of connection point shown in Fig. 5. `ISComponentClass` defines a composition interface represented by the `CompositionInterface` stereotype, which is an extension of the UML `Interface` metaclass. The composition interface groups the required and provided connection points according to their composition levels (`ModelInterface`, `ViewInterface`, `ControlInterface` and `DataInterface`).

The UML component model introduces a connector concept linking components either as delegation or assembly. Since our connector concept is much more flexible, allowing to define complexe collaboration logic, we have introduced a separate concept `ISConnector` to define the collaboration logic between components. The `ISConnector` stereotype is a specialisation of `ISComponent`. It is realised through the `ISConnectorClass`, which in turn is a specialisation of the `ISComponentClass`. We make use of the standard UML connectors to link components and connectors using the lollipop notation to describe the composition through required and provided connection points.

While most of the newly introduced stereotypes define attributes and methods, which may also be constraint using OCL expression for further refinement, we generally omitted them in Fig. 7 for the sake of simplicity. Only the `ISConnectorClass` stereotype exemplifies attribute and constraint definitions. The class defines four attributes, referencing the composed components `cSource` and `cTarget` and the required and provided connection points used for composition. We have defined OCL constraints ensuring that the composed components define the specified types of source and target connection points required by the connector, since the UML editor would allow for arbitrary compositions.

7 Model-Driven Composition Platform

The UML Profile and default connectors have been realised as part of the CompIS platform for model-driven, component-based information system engineering. Figure 8 gives an overview of the platform architecture. Our platform

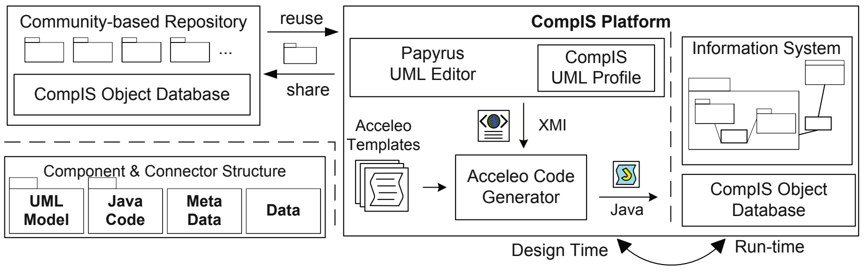


Fig. 8. CompIS Platform Architecture

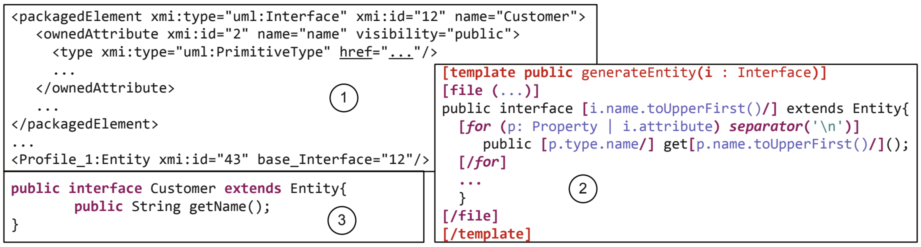
supports both the design and run-time of component-based information systems. At design-time, developers design their systems using a graphical UML editor and the CompIS UML Profile. We make use of the Papyrus Eclipse Plug-in[5], a graphical UML editor with support for profile definition, which we used to define the CompIS profile. A developer simply has to apply our profile to their model to get access to the defined stereotypes. This ensure that components, connectors and compositions follow our model description.

As part of the design process, developers may define their own components and connectors, or may reuse shared components and connectors from the community-based repository. Shared components can be imported into the developer's local platform and be used for composition. To make component and connectors shareable in a model-driven environment, their specification must include the component source code as well as the UML definition. Therefore, components and connectors are represented by the structure shown in the bottom left corner on Fig. 8, consisting of the UML model in the form of a XMI file, the source code

[5] `http://www.eclipse.org/papyrus/`

(Java files), metadata describing their purpose and connection points, and possibly also data. For example, a currency control component would be imported with actual currency conversion data that is regularly updated.

From the UML models, the code generator generates the composed information system. Code generation includes the generation and initialisation of newly defined components and connectors and the initialisation of imported components and connectors. For code generation, we rely on the Acceleo[6] plug-in, a model-to-text transformation tool following a template-based approach to code generation. Models, represented in the XMI format are transformed to text using user-defined templates.



```
<packagedElement xmi:type="uml:Interface" xmi:id="12" name="Customer">
   <ownedAttribute xmi:id="2" name="name" visibility="public">
     <type xmi:type="uml:PrimitiveType" href="..."/>
     ...
   </ownedAttribute>
   ...
</packagedElement>
...
<Profile_1:Entity xmi:id="43" base_Interface="12"/>
```

①

```
[template public generateEntity(i : Interface)]
[file (...)]
public interface [i.name.toUpperFirst()/] extends Entity{
  [for (p: Property | i.attribute) separator('\n')]
     public [p.type.name/] get[p.name.toUpperFirst()/]();
  [/for]
  ...
  }
[/file]
[/template]
```

②

```
public interface Customer extends Entity{
        public String getName();
}
```

③

Fig. 9. Template-based code generation

Figure 9 gives a simplified overview of the template-based code generation from XMI definitions. The XMI definition for a `Customer` entity connection point ① is of type `UML::Interface` and is linked to the `entity` stereotype via its XMI::ID (see last line of XMI file). In ②, an excerpt of the corresponding Acceleo code template is shown, defining the model to text transformation and the resulting Java code realising the connection point as Java interface extending the `Entity` interface ③. Note that the generated code may be extended by the developer. In fact, method bodies are left empty and marked with TODO annotations.

The code is generated for and deployed onto the CompIS platform. Note that there is no strict separation between run-time and design-time in the sense that we support continuous evolution of information systems. New components may be composed with ones that are already deployed as part of the information system, which allows for continuous evolution.

The CompIS platform is based on an extended object database which provides a metamodel extension mechanism introduced in [24]. This mechanism allows for extensions to be implemented as a native database facility rather than a layer on top of it. The database core provides the basic data management facilities used to model such extension modules. The core constructs are classes representing the concepts `Object`, `Class`, `Collection` and `Association`, extent collections for each one of these classes, associations among them and operators for the creation, retrieval, manipulation and deletion of class instances, depicted in Fig.10.

[6] `http://www.eclipse.org/acceleo/`

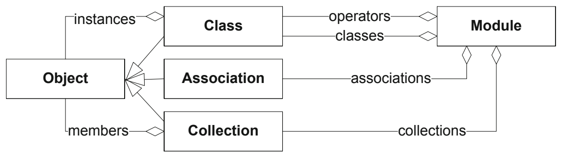


Fig. 10. Core Metamodel Concepts

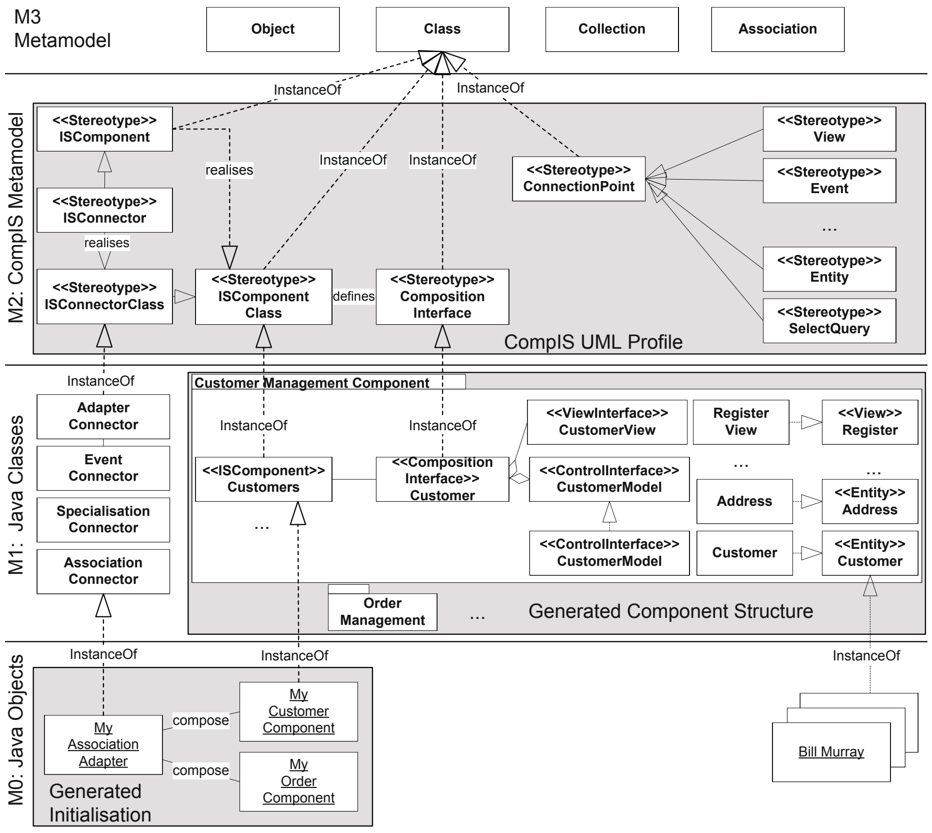


Fig. 11. MOF Levels

In order to support component definitions, compositions and deployment, we have defined a component module which implements the CompIS model by means of these core constructs. When the CompIS module is registered with the database, its classes, collections, associations and operators become part of the native database API. Consequently, we obtain a database supporting the definition, composition and deployment of components as described in Sect. 4.

Figure 11 is a summary of our approach and platform using the Meta Object Facility model (MOF) [25]. The M2 level represents the component metamodel described as UML profile. The metamodel has been realised as a metamodel extension module defined in terms of the object, class, collection and association concepts provided by the core module of the extended object database. Therefore, the core module of the object database establishes the M3 level.

The level M1 consist of the default connector and components represented by packages defining Java classes generated by our platform. The parts shaded in grey represent the source code generated from component models. In the example, the realisation of a customer component is shown. Finally, the M0 level defines the data managed by the components as well as component and connector instances configured for a particular composition through generated initialisation code.

8 Conclusion

We introduced an approach, model and platform for model-driven, component-based information system engineering, where information systems are designed from shared components and connectors. By encapsulating the collaboration logic into explicit connectors between components, we increase component compatibility and make our system resilient to component updates.

While we currently have focused on the definition of UML structure models, in the future, we plan to also look into extending interaction and behaviour models. This would support the model-driven definition of more complex collaboration logic, a more comprehensive system design and ultimately, also further reduce the amount of manual code development. We want to note that while the Papyrus UML editor is very powerful, its usability, especially when designing models based on user-defined profiles, is rather tedious. A tighter integration of the UML editor and user-defined profiles would greatly improve a developer's productivity. Finally, while we have presented our approach based on an eCommerce scenario, the approach is general and applicable to other information system domains. By doing so, the need for new connectors that for domain-specific compositions may arise, which would further complete our work.

References

1. Kazman, R., Chen, H.M.: The Metropolis Model a New Logic for Development of Crowdsourced Systems. Commun. ACM 52(7) (2009)
2. Leone, S., de Spindler, A., Norrie, M.C., McLeod, D.: Integrating Component-based Web Engineering into Content Management Systems. In: Daniel, F., Dolog, P., Li, Q. (eds.) ICWE 2013. LNCS, vol. 7977, pp. 37–51. Springer, Heidelberg (2013)
3. Rumbaugh, J., Jacobson, I., Booch, G.: Unified Modeling Language Reference Manual, 2nd edn. Addison-Wesley Professional (2010)
4. Heineman, G.T., Councill, W.T. (eds.): Component-based Software Engineering: Putting the Pieces Together. Addison-Wesley Longman Publishing Co., Inc. (2001)
5. Crnkovic, I., Sentilles, S., Vulgarakis, A., Chaudron, M.R.: A Classification Framework for Software Component Models. IEEE Transactions on Software Engineering 37(5), 593–615 (2011)

6. Lau, K.K., Wang, Z.: Software Component Models. IEEE Trans. Softw. Eng. 33, 709–724 (2007)
7. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall PTR (2005)
8. Daniel, F., Soi, S., Tranquillini, S., Casati, F., Heng, C., Yan, L.: Distributed Orchestration of User Interfaces. Inf. Syst. 37(6), 539–556 (2012)
9. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web Services Description Language (WSDL) 1.1. W3C Note (2001)
10. Organization for the Advancement of Structured Information Standards (OASIS): Web Services Business Process Execution Language (WS-BPEL) Version 2.0 (2007)
11. Thalheim, B.: Component Development and Construction for Database Design. Data Knowl. Eng. 54(1) (2005)
12. Casanova, M.A., Furtado, A.L., Tucherman, L.: A Software Tool for Modular Database Design. ACM Trans. Database Syst. 16(2) (1991)
13. Leone, S., Norrie, M.C.: Building eCommerce Systems from Shared Micro-Schemas. In: Meersman, R., et al. (eds.) OTM 2011, Part I. LNCS, vol. 7044, pp. 284–301. Springer, Heidelberg (2011)
14. Ma, H., Schewe, K.-D., Thalheim, B.: Modelling and Maintenance of Very Large Database Schemata Using Meta-structures. In: Yang, J., Ginige, A., Mayr, H.C., Kutsche, R.-D. (eds.) UNISCON 2009. LNBIP, vol. 20, pp. 17–28. Springer, Heidelberg (2009)
15. Shaw, M.: Modularity for the Modern World: Summary of Invited Keynote. In: AOSD 2011 (2011)
16. Medvidovic, N., Taylor, R.N.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Trans. Softw. Eng. 26(1), 70–93 (2000)
17. Clements, P.C.: A Survey of Architecture Description Languages. In: IWSSD 1996 (1996)
18. Orriëns, B., Yang, J., Papazoglou, M.P.: Model Driven Service Composition. In: Orlowska, M.E., Weerawarana, S., Papazoglou, M.P., Yang, J. (eds.) ICSOC 2003. LNCS, vol. 2910, pp. 75–90. Springer, Heidelberg (2003)
19. Baïna, K., Benatallah, B., Casati, F., Toumani, F.: Model-driven web service development. In: Persson, A., Stirna, J. (eds.) CAiSE 2004. LNCS, vol. 3084, pp. 290–306. Springer, Heidelberg (2004)
20. Clemente, P.J., Hernández, J., Sánchez, F.: Extending Component Composition Using Model Driven and Aspect-Oriented Techniques. Journal of Software 3(1), 74–86 (2008)
21. Object Management Group (OMG): Object Constraint Language, OCL (2010), `http://www.omg.org/spec/OCL/2.2/`
22. Ceri, S., Fraternali, P., Bongio, A.: Web Modeling Language (WebML): A Modeling Language for Designing Web Sites. Computer Networks 33(1-6), 137–157 (2000)
23. Fuentes-Fernández, L., Vallecillo-Moreno, A.: An Introduction to UML Profiles. UPGRADE, European Journal for the Informatics Professional 5(2), 5–13 (2004)
24. Grossniklaus, M., Leone, S., de Spindler, A., Norrie, M.C.: Dynamic Metamodel Extension Modules to Support Adaptive Data Management. In: Pernici, B. (ed.) CAiSE 2010. LNCS, vol. 6051, pp. 363–377. Springer, Heidelberg (2010)
25. OMG: Meta Object Facility (MOF) Core Specification Version 2.0 (2006), `http://www.omg.org/cgi-bin/doc?formal/2006-01-01`