Journal of Web Engineering, Vol. 13, No. 3&4 (2014) 183–202
© Rinton Press

COMPONENT-BASED WEB ENGINEERING USING SHARED COMPONENTS AND CONNECTORS

STEFANIA LEONE[1, a], ALEXANDRE DE SPINDLER[2], MOIRA C. NORRIE[3] and DENNIS MCLEOD[1]

[1]Semantic Information Research Laboratory, University of Southern California
Los Angeles, CA, 90089-0781, USA
{stefania.leone|mcleod}@usc.edu

[2]Center for Information Systems, Zurich University of Applied Sciences
8400 Winterthur, Switzerland
alexandre.despindler@zhaw.ch

[3]Institute for Information Systems, ETH Zurich
8092 Zurich, Switzerland
norrie@inf.ethz.ch

Today, web development platforms often follow a modular architecture that enables platform extension. Popular web development frameworks such as Ruby on Rails and Symfony, as well as content management systems (CMS) such as WordPress and Drupal offer extension mechanisms that allow the platform core to be extended with additional functionality. However, such extensions are typically isolated units defining their own data structures, application logic and user interfaces, and are difficult to combine. We address the fact that applications need to be configured more freely through the composition of such extensions. We present an approach and model for component-based web engineering based on the concept of components and connectors between them, supporting composition at the level of the schema and data, the application logic and the user interface. We have realised our approach in two popular web development settings. First, we demonstrate how our approach can be integrated into web development frameworks, thus bringing component-based web engineering to the developer. Second, we present, based on the example of WordPress, how advanced end-users can be supported in component-based web engineering by integrating our approach into CMS. The applicability of our approach in both settings demonstrates its generality.

Keywords: Component-based Web Engineering, Web Development Frameworks, Content Management System, WordPress, Symfony

1 Introduction

Nowadays, web information systems are typically developed in the context of complex and evolving requirements, where systems are continuously adapted and extended. In the domain of the Web, community-driven development [8] has become a popular way of providing more configurable and extensible web platforms. Ruby on Rails[b]and Symfony[c]are examples of web application frameworks targeted at professional web developers. Typically, such frameworks

[a]Stefania Leone's work is supported by the Swiss National Science Foundation (SNF) grant PBEZP2_140049.
[b]http://rubyonrails.org/
[c]http://symfony.com/

foster a model-view-controller (MVC) application design and adhere to modular architectures, where developers are free to install shared modules in order to extend the framework core. Similarly, content management systems (CMS) such as WordPress[d] and Drupal[e] follow such a paradigm by offering the possibility to install plug-ins that extend the platform core with additional functionality. The WordPress Plug-in Directory[f] hosts tens of thousands of plug-ins developed by the WordPress community, providing functionality ranging from site access statistics, over sophisticated photo galleries to full-fledged eCommerce solutions. Since CMS are frequently used by small companies and individuals, the developers can often be categorised as advanced end-users with a limited set of technical skills. Modern CMS platforms such as WordPress therefore allow plug-ins to be selected and installed through graphical administrator interfaces.

Such extensions may define their own data structure, application logic and user interfaces. Although extremely powerful when it comes to extending the platform core, extensions are typically difficult to combine. In the case of CMS, extensions represent isolated units and there is no systematic approach to composing them with other extensions. For example, a small company that runs their online shop based on a WordPress eCommerce plug-in, e.g. WooCommerce,[g] might want to perform a customer satisfaction survey by using a survey plug-in such as WordPress Simple Survey.[h] Ideally, the survey plug-in could directly integrate the customer data managed by the eCommerce plug-in with its own participant data. However, the current WordPress application model would require the company's developer to familiarise themself with the code of the eCommerce and survey plug-ins and programatically extract and map the customer data from the eCommerce plug-in to the participant data defined by the survey plug-in. This is a task that goes far beyond the skills of a typically WordPress user.

While web development frameworks such as Symfony provide basic support for collaboration between extensions, the collaboration is typically restricted to message passing at the service level. There is no explicit and systematic support for other types of collaboration, such as a collaboration of survey and eCommerce extensions where data is exchanged and integrated. While professional developers have the skills of familiarising themselves with the code of the two extensions and realising the collaboration code, this is a cumbersome and inefficient task, which also compromises the system's update resilience. The collaboration logic, which may become arbitrarily complex, needs to be integrated as part of one of the two extensions. If the developer later decides to install an update of that extension, they would either have to manually copy and reintegrate the collaboration code into the new version, or the collaboration logic would be lost.

This article is an extended version of our previously published work [12], where we showed how such compositions could be supported in the setting of CMS by using a well-defined component model. In order to build an application, components are composed through configurable connectors between them that encapsulate collaboration logic. We introduce different connector types supporting compositions at the schema and data level, at the level of the application logic, and at the level of the user interface. In [12], we described how we imple-

[d] http://wordpress.org
[e] http://drupal.org/
[f] http://wordpress.org/extend/plugins/
[g] http://wordpress.org/extend/plugins/woocommerce/
[h] http://wordpress.org/extend/plugins/wordpress-simple-survey/

mented the approach for the WordPress platform, offering a graphical composition plug-in that enables components to be composed based on configurable connector types. In parallel, we have extended an object database with our component model and investigated how our approach can be used to support model-driven information system engineering [10].

This article extends and generalises our previous work by showing how the approach can be applied in popular web development frameworks as well as CMS platforms. In this way, we are able to offer component-based web engineering to a wide range of professional developers: our CMS extension and its graphical composition tool is targeted at advanced end-users with a mix of design and technical skills, while our web development framework extension is tailored towards experienced web developers.

We give an overview of the background in Sect. 2, followed by an introduction to our approach in Sect. 3. We present the component and composition model in Sect. 4, and follow this with an extended example illustrating its use in Sect. 5. Section 6 describes how we integrated our approach into the web development framework Symfony, while Sect. 7 describes how we integrated it into the WordPress platform, including the presentation of the composition plug-in. Concluding remarks are given in Sect. 8.

2 Background and Related Work

Over the years, numerous frameworks and approaches for designing and developing web information systems have been proposed. Popular web development frameworks such as Ruby on Rails, Zend[i]or Symfony facilitate web application development in terms of MVC application designs and high-level abstractions for recurring application requirements such as HTTP request and response handling, persistence and user interface generation. These frameworks typically propagate a modular development approach based on loosely-coupled modules that interact at the service level as advocated by service-oriented architecture (SOA) [5]. Such collaborations create explicit dependencies between components. Also, collaboration is limited to message passing, while a developer's requirement may call for different or more complex forms of collaboration, such as data exchange and transformation as required by the composition between an eCommerce extension and a survey extension, or collaboration at the user interface level.

In parallel, CMS became popular for providing simple ways of creating websites and publishing their content. Platforms such as WordPress and Drupal provide graphical administrator interfaces, which support the design of content and layout in terms of general publishing units and presentation styles. The extension mechanisms inherent to these platforms allow for the integration of arbitrary data and services in order to support the creation of more complex web information systems. The configuration and use of plug-ins is typically performed through the administrator interface. While web development frameworks at least offer service-based collaboration, in the case of CMS, there is no support for composing plug-ins. Also, there is little control or conventions with respect to the plug-in internals and, in the case of WordPress and Drupal, developing and composing plug-ins requires knowledge of PHP as well as an in-depth understanding of the underlying CMS and its inner workings.

A number of approaches support web application development from reusable components. With WebComposition [6], web applications are built through hierarchical compositions of

[i] http://www.zend.com

reusable application components. Similarly, web mashups are composed through the orchestration of reusable, self-contained services that interact at the message level and may span multiple applications and organisations. Mashup editors offer graphical tools as an alternative to programmatic interfaces. These tools support the composition process, both for general, e.g. [4, 16, 3] and domain-specific mashup creation, e.g. [7, 1]. While some mashup editors help users to integrate information from distributed sources, others provide support for building new applications from reusable components. For example, MashArt [3] enables advanced users to create their own applications through the composition of user interfaces, application components and data sets. The focus is on integrating existing components through event-based composition, where components can react to events of other components.

We build on and extend these ideas. In contrast to previous work, our approach offers component-based web engineering based on the definition and configuration of explicit connectors that encapsulate the collaboration logic between components. As stated in [17], 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, our component model is inspired by the Architecture Description Language (ADL) [15, 2]. ADL is an approach to component-based software engineering [9], where the component model consists of components and explicit connectors between them. Our connectors encapsulate the composition logic between components, exhibiting functionality ranging from simple message passing to complex collaboration logic such as data transformation operations. The definition of explicit connectors thus allows for arbitrary components to be composed: connectors act as enabler for component collaboration and by that, we circumvent the problem of component incompatibility.

We introduce different types of connectors, which can be configured to define the composition required in a particular composition scenario. For example, a schema connector could be configured to support the structural composition of the eCommerce and survey components, such as the integration of shop customers and survey participants. We will show how advanced end-users using CMS as well as programmers using web development frameworks can equally benefit from this approach.

Our approach and model is not dissimilar to the application model introduced by the Google Android platform[j]for developing and running mobile applications. Their application model propagates the reuse of different types of application components across multiple applications, where application compositions are configured through so-called intents that define the glue code between the various components. While intents allow base values to be passed among components in the form of key-value pairs, our connectors generalise this approach and may define arbitrarily complex collaboration logic between components.

3 Approach

We introduce a component-based approach to web engineering based on ideas of ADL where applications are modelled based on reusable components and explicit connectors between them. Components may provide arbitrary functionality and define their own data structure, application logic and user interface.

We introduce our approach based on the example of a company that makes use of a CMS extended with an eCommerce component for their online business. Figure 1 gives an

[j] http://developer.android.com/guide/

overview of this scenario. The eCommerce component, in the centre, enables the company to create and manage an online store, including product, customer and order management. The component defines a schema that represents the eCommerce application domain by means of entity types and relationships, and manages data structured accordingly. Furthermore, the component defines application logic by means of methods and events which implement the online store functionality. This functionality is made available to the shop customers by means of a graphical user interface.

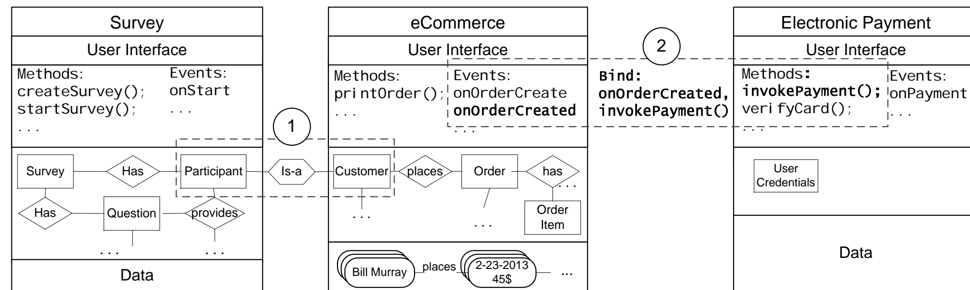

Fig. 1. Composition Scenario

In order to evaluate customer satisfaction, the company decides to perform a customer satisfaction survey. However, they would like to reuse their customer data when performing the survey. For this purpose, they selected a survey component—shown on the right—that offers the required functionality to define and carry out surveys. The survey component, in turn, consists of a user interface, application logic and a schema describing its data. However, the company wants to avoid having two separate user entities and therefore creates a connection between the eCommerce customer and the survey participants.

The connector (1) on the left in Fig. 1 defines the composition between the two components. It is a specialisation connector that defines an `is-a` relationship between the `Customer` entity of the eCommerce component and the `Participant` entity of the survey component. Through this specialisation connector, the customer data can be reused as participant data for the survey without the need for additional development or configuration efforts.

Figures 2 and 3 illustrate, by means of screenshots taken from our graphical component composition tool, how a user configures a specialisation connector through a graphical composition wizard. The composition tool allows for arbitrary components to be composed using configurable default connectors. We assume that the user already selected the components to be composed as well as the default specialisation connector required for the composition. This selection configures the composition wizard with the required metadata about the components to be composed. Figure 2 shows how the user creates the actual `is-a` relationship by selecting the `Customer` entity of the eCommerce component and the `Participant` entity of the survey component. Furthermore, the user sets the `Customer` entity to become the parent entity by checking the *parent* checkbox. Next, the user has the possibility to define attribute mappings between the attributes of the two entity types. In our example, while both entities share attributes for person names, the names of these attributes do not match.

Figure 3 shows how such attribute mappings are created. The user is about to create a mapping between the `User.firstname` and the `Participant.forename` attributes. At the

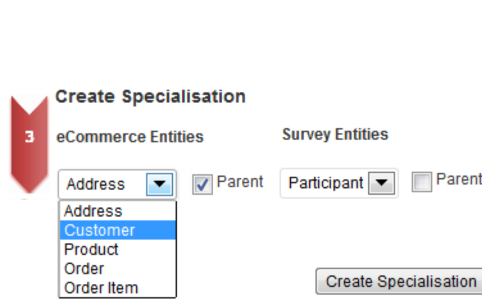

Fig. 2. Specialisation Screenshot

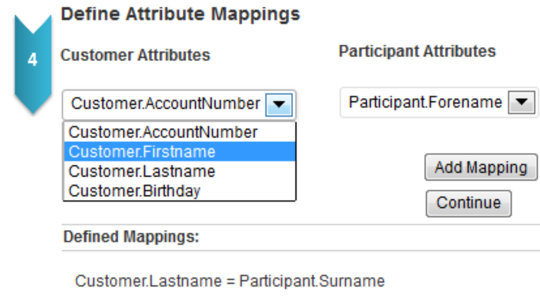

Fig. 3. Attribute Mapping Screenshot

bottom of the figure, the list of mappings defined so far is shown, including the attribute `User.lastname` that was previously mapped to the `Participant.surname` attribute. With these mappings, the specialisation connector ensures that, each time the name of a survey participant is accessed, the corresponding customer name from the eCommerce component is retrieved and displayed. Note that, in this example, the specialisation does not require any data mapping operations to be executed, since the survey component does not yet contain data. However, when composing two components containing data, the specialisation definition implies the execution of data mapping along with the definition of conflict resolution strategies, which is also supported by our composition wizard.

While this is the basic functionality provided by the specialisation connector, advanced users are free to extend the configured connector programatically. For example, the connector could be extended for data mining by defining queries that join survey data with customer data in order to answer questions such as "Do customers who selected answer (a) in question 4 buy similar products?".

In a second step, the company decides to offer support for electronic payments, a functionality that is not provided by the eCommerce component. For this composition, the eCommerce component is composed with an electronic payment component, shown on the right of Fig. 1. The event registration connector (2) operates at the application logic level, based on events and callback methods. Figure 4 and 5 show the steps involved in configuring such a connector. Again, we assume that the user has already selected the components to be composed and the connector type, which configures the composition wizard for this specific composition scenario. Furthermore, the user has decided that the electronic payment component should be invoked as a result of an event that occurs in the eCommerce component. The screenshot in Fig. 4 shows how a user defines a binding between events and handling by selecting an event and method. In the current example, the user selected the `onOrderCreated` event from the survey component. According to the description shown below the drop-down menu, the event gives access to the order created as well as the order attributes. On the left, the user selected the `invokePayment` method of the electronic payment component. As a result, the description of the method and its parameters is displayed, explaining that the method takes two parameters `amount` and `currency`.

Once the basic binding is created, the user may define mappings between the event object

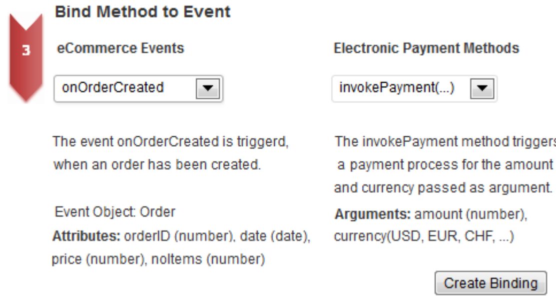

Fig. 4. Binding Creation Screenshot

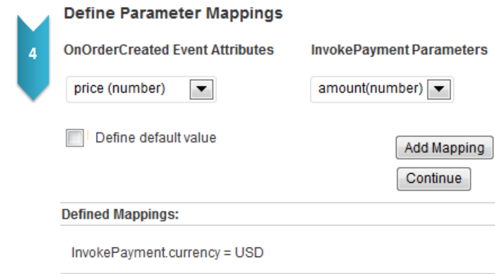

Fig. 5. Parameter Mapping Screenshot

attributes and the method parameters, as shown in Fig. 5. In the current example, the user intends to map the `price` attribute of the order object to the `amount` parameter of the `invokePayment` method. Furthermore, a list of all mappings created so far is illustrated at the bottom. The user already created a mapping for the currency parameter by assigning default value "USD" to it. Note that users are free to define such default values for parameters in cases where the attributes and parameters do not match or are incompatible, and we support basic type transformation.

As these two composition examples illustrate, connectors provide the glue between components and are configured by the user to adhere to a particular composition scenario. The definition of explicit connectors thus enables incompatible components to collaborate. We offer different types of connectors that support the composition at various levels of a component. Figure 6 gives an overview of the composition levels and shows, from left to right, that connectors may be used at the data level, the schema level, the application logic level and the user interface level. We provide connector types for all these levels and present our component model including the various connector types in the following section.

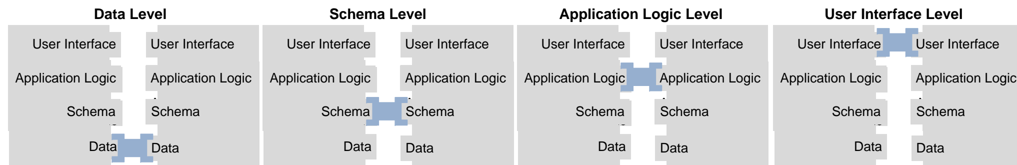

Fig. 6. Composition Levels

4 Component Model

A component is an application providing arbitrary functionality to its users. Components may be composed with other components using explicit connectors between them in order to form more complex applications. The general component model along with the composition interface is shown on the left in Fig. 7, while the eCommerce component introduced in Sect. 3 is shown on the right as an example following this model.

Formally, a component is defined as a tuple of four elements:

$$Component = \langle Schema, Data, Application\ Logic, User\ Interface \rangle$$

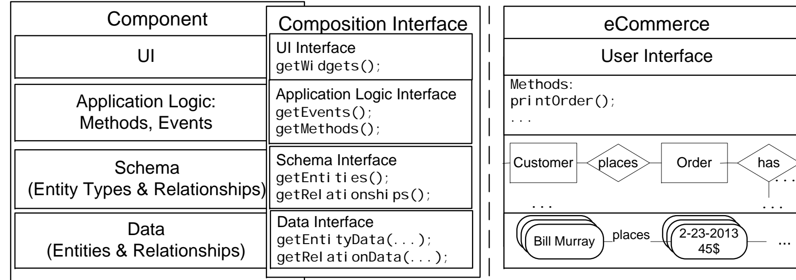

Fig. 7. Component Example

The *Schema* is a data model instance describing the component *Data* in terms of a set of entity types $\{E_1, \ldots, E_M\}$ and relationships $\{R_1, \ldots, R_N\}$. The *Application Logic* includes a set of methods $\{m_1(), \ldots, m_U()\}$ implementing the application logic and events $\{e_1, \ldots, e_V\}$ related to these methods. Components typically contain basic CRUD methods supporting the management of their entity types and relationships, as well as higher-level methods providing domain-specific functionality. Component developers are free to define an arbitrary number of events triggered by the execution of such methods. For example, a component may define events marking the start and end of CRUD method executions.

Finally, the *User Interface* defines the graphical user interface. In CMS, the user interface is typically specified by layout themes defining the general presentation of the provided publishing units for the complete web site. As part of the user interface, components may define a set of views $\{V_1, \ldots, V_N\}$ displaying specific component data or providing component services to the users. Views represent complete user interfaces including user interface controls, layout and style templates. Note that components do not necessarily specify multiple or all of these four elements. For example, while the eCommerce component specifies *Schema*, *Application Logic* and *View* elements, other components may for example only specify *Application Logic* and *Schema* elements.

Components expose a composition interface which defines in which way they may be composed with other components. In order to implement such an interface, component developers need to specify which of the component elements they wish to make available for composition. Component interfaces do not need to expose component elements at all levels. At the schema-level, the interface specifies the subset $\{E_i, \ldots\} \subseteq \{E_1, \ldots, E_M\}$ of composable entity types and the subset $\{R_j, \ldots\} \subseteq \{R_1, \ldots, R_N\}$ of composable relationships. The specification of the data available for composition consists of a query Q over the composable schema elements. Similar to the schema interface definition, application logic made available for composition is defined in terms of subsets $\{m_k(), \ldots\} \subseteq \{m_1(), \ldots, m_U()\}$ and $\{e_l(), \ldots\} \subseteq \{e_1, \ldots, e_V\}$. Finally, user interface views are exposed in terms of the subset $\{V_i, \ldots\} \subseteq \{V_1, \ldots, V_N\}$.

In Fig. 7, a programmatic representation of the composition API is shown, with getter methods to access the defined subsets of composable views, methods, events, schema elements and data.

Connectors specify how components collaborate and at which level. For example, the specialisation connector presented in Sect. 3 defines an `is-a` relationship at the schema level, and the event registration component binds a callback function to an event at the application

logic level. Figure 8 shows the basic types of connectors—categorised based on the composition level. The view connector, shown at the top, supports composition at the UI level through the integration of views between components. The connector forms the union of views defined as $User\ Interface := \{V_1, \ldots\} \subseteq UserInterface_A \cup UserInterface_B$. In the example in Fig. 8, the connector integrates a view of component A into the user interface of component B.

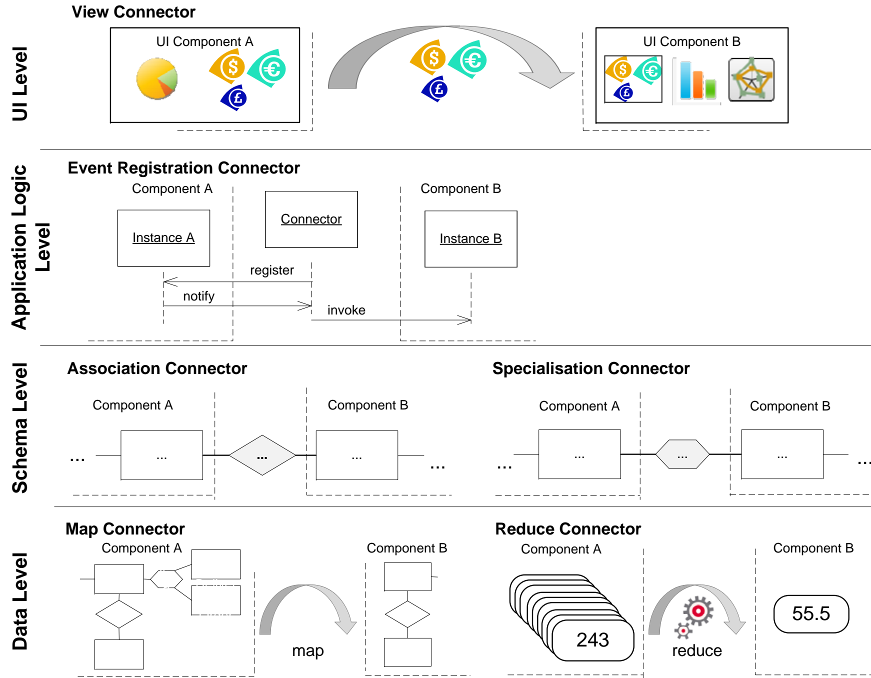

Fig. 8. Basic Connector Types and Composition Scenarios

At the application logic level, Fig. 8 illustrates the event registration connector based on a UML sequence diagram that reflects the collaboration between components and connectors in an event-based setting. The connector is specified as $Application\ Logic := \{m_1()\{e_i \rightarrow m_j()\}, \ldots\}$ defining functions binding events of component A to methods of component B.

Schema-level connectors compose components based on schema elements, such as specialisation and association [13]. As shown in Fig. 8, a specialisation connector defines an `is-a` relationship establishing a specialisation relationship among entity types from different component schemata and the association connector defines a relationship between two entity types from different components. More generally, a schema connector may define arbitrary schema elements among component entities $Schema := \{\{E_1, \ldots\}, \{R_1, \ldots\}\}$.

Finally, data connectors allow data from one component to be reused by another component. As shown in Fig. 8, data reuse may be defined by a mapping connector that maps the schema of one component to the schema of another component, or by a reduce connector that transforms data from one component to a format specified by another component. Generally, data connectors may be defined as combinations of map and reduce functions of the form $Data := \{map()\{E_i.a_j \leftarrow reduce(E_k.a_n, E_l.a_m)\}, \ldots\}$. Such map and reduce functions may in turn be bound to data mapping connector events to define whether the mappings should

occur once, multiple times or periodically.

Note that we have given a minimal specification of the various connector types, while they may define richer functionality. For example, the association connector may also define application logic in the form of CRUD methods in order to create associations, as well as a view that allows new associations to be created and viewed graphically. Similarly, a reduce connector may define a user interface, where the reduce function could be configured.

As seen with these examples, connectors define the same building blocks as components and, thus, can be seen as a special type of component, where the functionality is not targeted at the application domain, but rather at the composition of domain-specific application units.

Figure 9 shows the metamodel of our component model. A component defines a user interface, application logic, schema and data, and, depending on the implementation technology, these elements may be realised in different ways. A connector is a sub-type of component, and therefore, they can in turn be composed. Connectors are classified according to their supported composition level, which defines the access points of a connector. A concrete connector is an instance of such a connector type and is instantiated with values that are particular to a composition scenario. For example, a specialisation connector will be instantiated with an `is-a` relationship between two entity types.

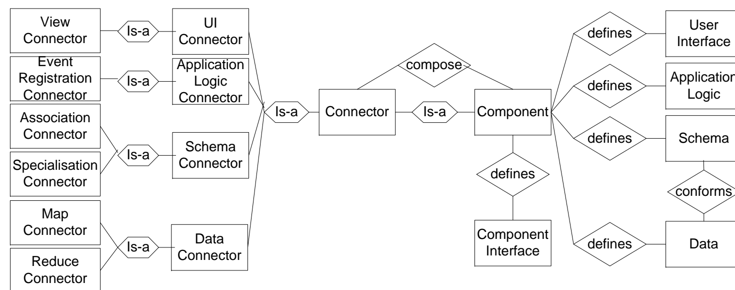

Fig. 9. Component Metamodel

5 Extended Example

We will now present an extended example of an eCommerce application composition to illustrate in more detail how the approach can be applied in practice. Figure 10 shows the composition scenarios on the left and the involved connector configurations on the right.

As a first step, the eCommerce component is composed with a review component in order to allow products to be reviewed by customers. The composition is based on the association connector shown in (1). This connector defines the association between the product and the review entity including the cardinality constraints. A product may have *0* or *n* reviews, while a review relates to exactly one product.

In a second step (2), the eCommerce component is composed with the survey component through a specialisation connector as described in Sect. 3. On the left, the specialisation is defined by means of an `is-a` relationship and the two attribute mappings. To evaluate the outcome of the survey the company makes use of a spreadsheet component. The two components are composed at the data level (3), where the data of the survey component is mapped to the data format of the spreadsheet component. The mapping is specified by a

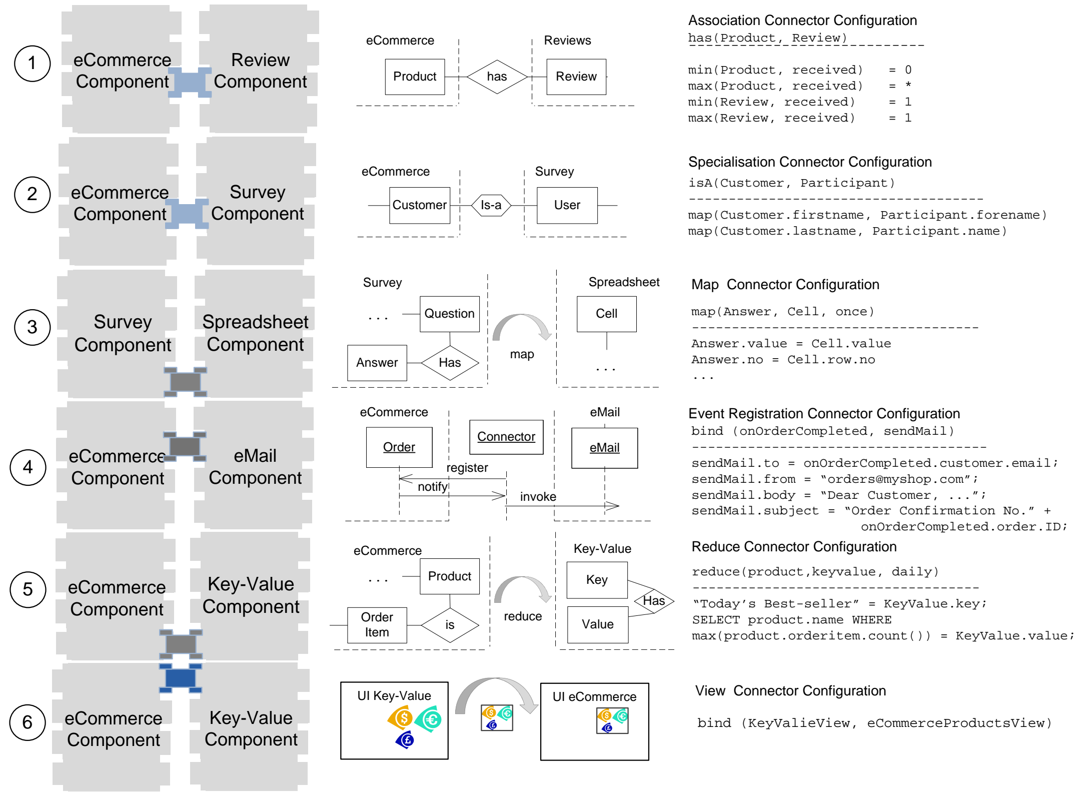

Fig. 10. Scenario Application

number of attribute mappings between the entities of the survey component and the entities of the spreadsheet component, where the survey questions and answers are mapped to the cell entity of the spreadsheet component.

In order to send customers order notifications, the eCommerce component is composed with an email component (4). An event registration connector is configured in such a way, that, upon the completion of an order, an order confirmation email is sent to the respective customer. This functionality is achieved by binding the event `onOrderCompleted` to the `sendEmail` method and defining mappings between event attributes and method parameters.

Finally, the company would like to display the current best-selling product next to the products overview page. To do so, the eCommerce component is composed with a Key-Value component, which provides the functionality of managing and displaying key-value pairs. To realise the desired functionality, the two components are connected at two levels. In (5), a reduce connector is configured in such a way, that, once a day, the best-selling product of the day is selected from the eCommerce component and stored as value to the key "Today's best-seller" to the Key-Value component, as shown in the reduce connector configuration in Fig. 10. To present the best-selling product on the eCommerce products overview, in (6), the two components are also composed at the user interface level through a view connector. The connector configuration defines that the key-value view displaying the best-selling product is

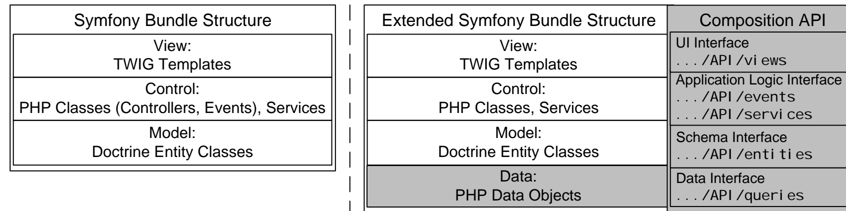

Fig. 11. Symfony Bundle Structure and Extension

placed alongside the eCommerce products overview.

6 Component-based Web Development Frameworks

In order to offer support for component-based web engineering to web developers, we demonstrate the integration of our approach with web development frameworks. With our approach, developers compose web applications from shared components and configurable connectors between them. We show how we integrated our approach with Symfony, a community-driven PHP-based framework. The integration of our approach with other frameworks supporting MVC application design and modular extensibility can be done analogously.

Web applications developed with Symfony are organised as so-called bundles. A bundle represents a fully functional web application following the MVC pattern. Bundles may exhibit arbitrary functionality, may access a database, make use of functionality provided by other bundles and offer their functionality to the user via web interfaces. Bundles may also provide framework functionality. In fact, the Symfony framework itself is based on a modular architecture, consisting of a set of loosely coupled bundles defining the core framework functionality. For example, the HTTP Fundamental bundle defines an object-oriented layer for the HTTP request and response handling, the Form bundle facilitates the HTML form creation, processing and reuse, and the Event Dispatcher bundle provides an event system.

The fact that bundles follow the MVC pattern is reflected in the bundle structure. A bundle defines a root folder and a number of subfolders containing the source code files for the application's model, controllers and views, as well as additional configuration files. The Symfony bundle structure is shown in the left part of Fig. 11. For the realisation of an application's model, developers may use the Doctrine bundle, which supports data management by means of an object-relational mapping (ORM) layer. With Doctrine, a bundle's model is represented by PHP objects referred to as entities that are mapped to the underlying database schema. A bundle's control is realised by bundle-specific controller classes that extend the Symfony `Controller` class, and other PHP classes implementing the bundle's application logic. Entities typically define their own controller classes providing basic CRUD functionality for their creation, retrieval, update and deletion. Finally, an application's view is typically realised by means of so-called TWIG templates, associated style sheets and JavaScript files. Symfony uses the TWIG template engine that supports the rendering of templates. Templates are associated to a controller function and fetched and processed upon the function's invocation when responses need to be rendered.

In order to speed up development and facilitate bundle creation, developers can make use of a generator tool that interactively generates a bundle. This generator creates the

Table 1. Symfony Concepts for Component-based Web Engineering

Component Model Concepts	**Symfony Concepts**
Component and Connector	Bundle
View	TWIG Template (and associated Files and Controller)
Method	PHP Function, Service (e.g. in Controller)
Event	Symfony Event
Entity and Relationship	Doctrine Entity, ORM and Database Schema
Data and Query	PHP Objects and Queries (e.g. SQL)
Composition API	URL of the form `/{bundlename}/api/[type]`

bundle structure in terms of folders containing source code files for default controllers, TWIG templates and bundle configurations. Also, a bundle's entities, the corresponding database schema and ORMs can be generated via this generator. As part of the generation process, the generated bundle is registered with the framework. Developers may develop their own bundles or install and reuse bundles shared by the community. The bundle installation process in Symfony is usually managed via Composer,[k] a tool for PHP dependency management and shared bundles may provide their own installation instructions.

Collaboration among Symfony bundles is achieved through dependency injection. Symfony provides a mechanism where controllers and other PHP classes can be registered as services. A central service container manages the creation and configuration of services defined by the registered bundles. Through the service container interface, services can be accessed and invoked across bundles and thus enable loosely coupled collaboration between bundles.

6.1 Realisation

We extended the Symfony application model with our component model to support component-based web engineering. With our extension, bundles can be composed to not only collaborate through services, but developers may define arbitrary complex collaboration logic—at the level of the user interface, application logic, schema and data—by means of connectors. Table 1 summarises how our component model concepts are realised through native concepts or extensions of the Symfony application model. On the left of Fig. 11 we illustrate the extended Symfony bundle structure coloured in grey.

Components are represented by Symfony bundles. Symfony's MVC bundle structure is an ideal match to our component structure defining user interface, application logic and schema. At the user interface level, views are represented by Symfony's TWIG templates, associated stylesheets, Javascript and controllers. At the application logic level, events are represented by user- or system-defined Symfony events supported by the Event Dispatcher bundle. Methods are represented by regular PHP functions typically defined as part of a bundle's controller or in a separate PHP class. Functions may be registered with the service container in order to expose them to other bundles. The schema defined by our component model is covered by the Doctrine bundle. Component entities and relationships are represented by Doctrine entity classes, the corresponding database schema and ORMs. Finally, data corresponds to PHP objects that are instances of the schema entities. Data may be exposed for sharing by means of queries over a bundle's schema. Here, the query language depends on the type of the underlying database, e.g. SQL for relational databases.

To support component composition, we extended the Symfony bundle definition with

[k] http://getcomposer.org/

a composition API. The composition API is accessible through a rest-based API defined and implemented by every component, giving access to the component elements shared for composition. A component's exposed elements can be accessed via the following URL pattern {`component-URL`}`/api/[element-type]`. For example, a component's shared events can be accessed via {`component-URL-Pattern`}`/api/events`. Thus, the URL

`www.myshop.com/../ecommerces/api/events`

returns an array specifying all exposed events defined by the eCommerce component. Similarly, shared views, services, entities and data can be accessed via analogous URLs. The composition API is realised by a configuration file, created by the bundle developer, which is fetched by the bundle's controller upon request of the composition API URL.

Connectors specifying the collaboration logic between components are also realised as bundles. Connectors define and encapsulate the composition logic between components through the components' shared elements. We implemented configurable connector bundles, one for each connector type. While we introduced connector configurations through a graphical composition interface in Sect. 3, in the case of Symfony, the configuration is based on connector templates that are configured manually by the developer, as shown in the next section.

6.2 Proof-of-concept application

As a proof of concept, we implemented the example application shown in Fig. 10 using the extended Symfony framework. The scenario components were realised as extended Symfony bundles and we configured the connector templates for the various composition scenarios.

When defining a new composition, instead of inspecting all the code of the involved components, the developer simply inspects the components composition API for suitable shared elements and then configures the connector template accordingly. For example, the collaboration between an eCommerce and email component, defining that upon the `OrderCompleted` event of the eCommerce component, the `sendMail` function of the email component should be invoked, can be configured using the event registration connector template. Below, we show an excerpt of the template. The event listener class encapsulated the collaboration logic defined by the event registration connector, which is a binding between an event and a service. The class is parametrised using extended Backus Naur notation and contains comments that support the developer when configuring the connector. The class defines a callback function that takes an event object as argument, performs the mappings between event object attributes and service parameters and invokes the service using the mapped parameters.

```
class <EVENT_LISTENER_CLASS_NAME> {
  public function <EVENT_NAME>(<EVENT_TYPE> $event) {
    //TODO ATTRIBUTE PARAMETER MAPPING
    //invoke function
    $service = $this->get("<SERVICE_NAME>");
    $service-><FUNCTION_NAME>([PARAMETER {, PARAMETER}]);
  }
}
```

To configure the connector, the developer inspects the composition API of the two involved components and decides on the shared elements to be used for the composition. The configured connector is shown in the code snippet below. The class `NotificationEventListener` defines a callback function `onOrderCompleted`. The `onOrderCompleted` function is invoked

by the Event Dispatcher upon the `OrderCompletedEvent` event of the eCommerce component. The function takes the event object as argument, performs the mapping between event object attributes and function parameters and invokes the `sendMail` function of the email component. The `sendMail` function is accessible trough the `shop.email` service defined by the email component.

```
class NotificationEventListener {
  public function onOrderCompleted(OrderCompletedEvent $event) {
    // get order information  from event object
    $to = $event->getCustomerEmail();
    $from = ‘‘orders@myshop.com’’
    ...
    // invoke function
    $service = $this->get("shop.email");
    $service->sendMail(array("to" => $to, "from" => $from, ....));
    // ...
  }
}
```

A developer is free to further extend the connector with any additional functionality, if required. For example, data could be stored to the database, or some other services could be invoked. Note that the event listener class `NotificationEventListener` needs to be manually registered in Symfony's configuration file as part of the event registration.

As a second example, we present the configuration of the view connector between the eCommerce and the key-value components. As a result of this composition, the key-value view is displayed alongside the eCommerce products overview to display the best-selling product of the day. The code snippet below illustrates an excerpt of the view connector template, where two views are composed in such a way, that one view is displayed as the main content of the web page, while the other view is placed in the sidebar. Note that the double curly brackets include the invocation of the render function and the template's controller. Through this inclusion, the original content presented by the template is shown.

```
{% extends ’::base.html.twig’ %}
{% block body -%}
  <div id="content">{{ render(controller(<TWIG_TEMPLATE_NAME>)) }}</div>
  <div id="sidebar">{{ render(controller(<TWIG_TEMPLATE_NAME>)) }}</div>
{% endblock %}
```

Below, a snippet of the configured view connector is shown, defining that the products overview, represented by the `eCommerceBundle:Product:index`, is shown as part of a main content `div` element, while the key-value view `KeyValueBundle:KeyValue:index` is rendered next to the products overview in a sidebar `div` element.

```
{% extends ’::base.html.twig’ %}
{% block body -%}
  <div id="content">{{ render(controller(‘eCommerceBundle:Product:index’)) }}</div>
  <div id="sidebar">{{ render(controller(‘KeyValueBundle:KeyValue:index’)) }}</div>
{% endblock %}
```

As shown with these two examples, component composition via configurable connector bundles significantly saves development efforts. Without our extension, bundles could only be composed at the service level. Any other type of collaboration between bundles would involve detailed code inspection of the components and the manual implementation of the composition logic as part of one of the involved components, which causes further problems when component updates are available.

Table 2. WordPress Concepts for Component-based Web Engineering

Component Model Concepts	Wordpress Concepts
Component and Connector	Plug-in
View	WordPress Widget
Method	PHP Function
Event	Hook
Entity and Relationship	DataPress Entity
Data and Query	PHP Objects and Queries (e.g. SQL)
Composition API	Extension of Plug-in Header

While currently, the connector configuration is done manually, connector templates can be used as a starting point for automatic connector configuration, either based on Symfony's console-based configuration tool, or a graphical composition tool as shown in Sect. 3.

7 Component-based Content Management Systems

To show how component-based web engineering could be provided to developers using CMS platforms, we extended WordPress with support for our approach. We first give a short introduction to the WordPress plug-in mechanism before presenting our extension and the composition plug-in that supports composition through a graphical user interface.

The WordPress plug-in mechanism allows the WordPress blogging model to be extended in terms of data structure, application logic and presentation by hooking into the WordPress core. A number of hooks are provided as part of the platform core, which allow plug-ins to inject additional functionality, data structures and presentation into the WordPress core execution environment. Hooks may represent plug-in lifecycle events such as their installation or uninstallation, as well as administrative or end-user activities including the creation, manipulation, retrieval, selection, display and deletion of posts, pages or plug-in-specific data.

Typically, the plug-in code includes functions for creating and deleting database tables, for inserting and selecting table data and the assignment of these functions to particular hooks. Plug-in developers are free to define their own hooks, which enables plug-ins to react to events of other plug-ins. For the user interface, plug-ins rely on the WordPress publishing process and themes that define the structure and layout of the complete web site. A plug-in may, however, define widgets that can be placed in various areas of the user interface.

In order to install a plug-in, the files containing the plug-in code, typically one or more PHP and JavaScript files, are uploaded into the target WordPress platform through the WordPress administrator dashboard where they can be activated and deactivated. Upon activation, the additional functionality, data structures and presentation facilities become part of the WordPress platform and are available for immediate use.

7.1 Realisation

We extended the WordPress plug-in model to adhere to our approach, as summarised in Table 2. At the level of the application logic and user interface, the WordPress plug-in model matches our component model. At the level of the user interface, plug-ins may define widgets, which correspond to our view definition. The WordPress core handles the generation of the user interface from themes including the placement of such widgets. Similar to Symfony, application logic is represented by PHP functions and events. At the data and schema level, however, WordPress only supports a basic notion of types and data may be stored in arbitrary

ways and formats. Also, plug-ins do not define a composition API, as defined by our approach.

In order to integrate support for richer schema definition, we built on our previous work [11] where we introduced DataPress, a WordPress plug-in which supports the generation of tailored WordPress plug-ins from user-defined ER models. With DataPress, a user graphically defines an application domain by means of ER models and DataPress automatically generates a plug-in that allows data to be managed accordingly. For each defined entity type and relationship, DataPress generates data structures, CRUD functions and user interface components supporting the creation and management of the data. By building on this approach, we not only gain support for ER modelling, but we can also extend the automatic generation of plug-ins in order to conform to our component model. We additionally generate two hooks for each of the generated CRUD operations—a "before" and "after" hook. For example, for the creation of an order entity, the two hooks `onOrderCreate` and `onOrderCreated` are generated. Furthermore, we generate an extended plug-in header defining the composition API, which gives access to the composable plug-in elements. The plug-in header specifies the names of the entities, relationships, functions, events and widgets defined by a plug-in and the user can simply remove elements that should not be made available for composition. The configuration of a connector for a particular composition scenario is based on such names defined in the respective plug-in header.

In order to enable advanced end-users to compose components, we provide a composition plug-in that supports the graphical composition process, as illustrated in the screenshots in Sect. 3. Figure 12 gives an overview of the composition plug-in architecture. The composition plug-in, shown in the centre, is a regular WordPress plug-in that is integrated into the dashboard. It provides access to locally installed plug-ins, shown on the left, and the connector type plug-ins, shown on the right. It builds on an extended version of DataPress and

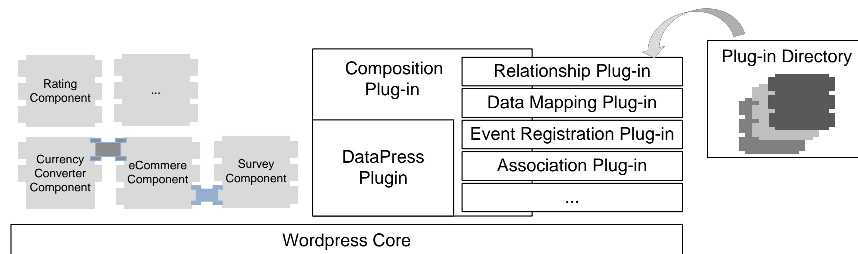

Fig. 12. Composition Plug-in Architecture

supports the generation of plug-ins from user-defined ER models structured according to our component model. Using this composition plug-in, new plug-ins can be composed with the installed ones by configuring one of the provided connector types. Assuming that all plug-ins are structured according to our approach, a user could also download, install and compose plug-ins from the WordPress Plug-in directory, shown on the left.

Each connector type is realised as a parameterised plug-in—similar to the connector templates presented in Sect. 6—which gets "instantiated" upon composition. The composition plug-in automatically generates and installs the configured connector plug-ins.

7.2 Proof-of-concept application

As already mentioned, with our composition plug-in connectors are graphically configured, and generated and installed automatically. As a proof of concept, we have realised the example application presented in Sect. 10 using our composition plug-in and we will show examples of the generated connectors. First, we show a configured event registration connector that corresponds to the configuration shown in Fig. 10, where a notification email is sent to the customer, once they completed the order.

```
/* Plug-in Name: NotificationConnector*/
...
add_action('onOrderCompleted', 'sendEmailTemp', 1, 2);

function sendEmailTemp($orderID, $email) {
  $to = $email;
  $subject = 'Order Confirmation no.'.$orderID;
  $from= 'orders@myshop.com';
  $body= 'Dear Customer....';
  sendEmail($to, $from, $subject, $body);
}
```

Through the configuration process, the connector was named *Notification Connector* and the event and function names to be bound together were injected into the plug-in template. In WordPress, the `add_action` function registers a specific hook with a specific function. The `add_action` function further defines the priority of the funtion invocation, as well as the numbers of arguments that are passed from the event to the function. While WordPress assumes that the number and types of attributes provided by the hook match the parameters of the callback function, we generalised this approach by giving the user the possibility to define attribute-parameter mappings, as shown in Fig. 5. In this example, the event `onOrderCompleted` defines two attributes while the function `sendEmail` takes four parameters. The mappings are reflected in the connector code. When an `onOrderCompleted` event is triggered, a helper function `sendEmailTemp` function is invoked, accepting the two attributes defined by the `onOrderCompleted` event as parameters. The function implements the defined attribute mappings and invokes the actual `sendEmail` function using these mappings. In this example, the attribute `$to` from `onOrderCompleted` is mapped to the parameter `$email`, and the parameters `$email, $from, $subject` are set to the default values. Note that more advanced users are free to extend a configured connector plug-in with additional code.

As a second example, we show the configured reduce connector described in Fig. 10 step (5) defines a query that selects the best-selling product of the eCommerce component and maps and stores it as a key value pair of the key value component The generated connector configuration defines the following code snippet, where the result of the SELECT query executed on the eCommerce component is mapped to the values that will be inserted into the key value component. According to the connector configuration, the reduce function is invoked once a day. Note that the reduce function shown in this snippet is a simplified version kept short due to the lack of space.

```
function reduce($select, $insert){
  global $wpdb;
  $results = $wpdb->get_results($select, ARRAY_A);
  foreach ($results as $result) {
    $wpdb->insert($insert, $result);
  }
}
```

As a last example, we show a view connector that injects a widget from one component into the user interface of another component, based on a user's configuration. In Fig. 10 step (6) the view connector is configured in such a way, that the key value widget is displayed in the default sidebar when the eCommerce main view is displayed. To support the selective display of widgets, we make use of a third-party widget *Widget Logic*[l] that extends the Wordpress widget functionality with the possibility to control with which pages a widget should be displayed. Upon activation, the configured view connector stores the widget-view mapping in the *Widget Logic* database table. The *Widget Logic* plug-in hooks into the WordPress publishing process in such a way that every time the eCommerce products overview is displayed, the key value widget is displayed alongside the eCommerce products overview.

While the plug-ins representing the configured event registration connector only defines application logic between two components and the widget connector only stores widget placement information, other connectors may be much richer and also define their own schema, data and views. For example, the association connector, which associates reviews to products in our example, also creates a database table as part of its installation process where associated pairs of entities are stored. Furthermore, the connector also defines a widget allowing users to graphically create associations. This is described in detail in [11]. As part of the association connector configuration, the user also configures the placement of the association widget, for which we rely on the Widget Logic widget. In the current example, the widget is placed alongside the product view, but it could also be configured to be visible along with both composed components, as part of a dynamic sidebar injected into the layout theme, or as part of the dashboard.

With our approach we enable the collaboration between plug-ins based on these configurable connectors. By doing so, we minimise development efforts: we circumvent the detailed inspection of plug-ins to be composed and the manual implementation of the composition logic as part of one of the involved plug-ins, which would causes problems, when plug-in updates are available.

8 Conclusion

Our approach, model and implementation is a practical solution to integrating component-based web engineering with CMS and web development frameworks supporting MVC application design and modular extensibility. By defining components and explicit connectors between them, we not only circumvent possible incompatibilities between components, but we also ensure that composed systems are resilient to component updates, since the composition logic is completely encapsulated within the connector code.

We see our work as a further step towards providing systems that are easy to develop [14]. The application of our approach in the domain of CMS as well as its integration with web development frameworks showcases the generality of our approach. We have shown, by means of proof-of-concept applications, how our approach simplifies the development of web information systems. Although the composition process through graphical user interfaces is clearly more limited than the programmatic implementation of connectors, the scenario presented has shown that a small set of simple connector types covers a wide range of composition scenarios, and thus allow for complex applications to be developed.

[l] http://wordpress.org/plugins/widget-logic/

While we applied our approach in an eCommerce scenario, we are exploring the application of our approach to diverse domains for the purpose of testing, experimenting, and refining it to support a high level of general applicability. For example, new domains may call for new connector types, which could be added at any level.

References

1. O. Chudnovskyy, T. Nestler, M. Gaedke, F. Daniel, J. Fernández-Villamor, V. I. Chepegin, J. A. Fornas, S. Wilson, C. Kögler, and H. Chang (2012), *End-user-oriented Telco Mashups: The OMELETTE Approach*, Proc. World Wide Web Conf. (WWW'12)(Companion Volume).
2. P. C. Clements. (1996), *A Survey of Architecture Description Languages*, Proc. Intl. Workshop on Software Specification and Design (IWSSD 1996).
3. F. Daniel, F. Casati, B. Benatallah, and M.-C. Shan (2009), *Hosted Universal Composition: Models, Languages and Infrastructure in mashArt*, Proc. Intl. Conf. on Conceptual Modeling (ER'09).
4. R. Ennals, E. Brewer, M. Garofalakis, M. Shadle, and P. Gandhi (2007), *Intel Mash Maker: Join the Web*, ACM SIGMOD Record, Vol.36(4) pp.27-33.
5. T. Erl (2005),*Service-Oriented Architecture: Concepts, Technology, and Design*, Prentice Hall PTR.
6. H.-W. Gellersen, R. Wicke, and M. Gaedke (1997), *Webcomposition: An object-oriented support system for the web engineering lifecycle*, Computer Networks, Vol.29(8-13) pp.1429-1437.
7. M. Imran, S. Soi, F. Kling, F. Daniel, F. Casati, and Ma. Marchese (2012), *On the Systematic Development of Domain-Specific Mashup Tools for End Users*, Proc. Intl. Conf. on Web Engineering (ICWE'12).
8. R. Kazman and H.-M. Chen (2009), *The Metropolis Model a New Logic for Development of Crowdsourced Systems*, Commun. ACM, Vol.52(7), pp.76-84.
9. Kung-Kiu Lau and Zheng Wang (2007), *Software Component Models*, IEEE Trans. Softw. Eng., Vol. 33, pp. 709-724.
10. S. Leone, A. de Spindler, and D. McLeod (2013), *Model-driven Composition of Information Systems from Shared Components & Connectors*, Proc. Intl. Conf. on Cooperative Information Systems (CoopIS'13).
11. S. Leone, A. de Spindler, and M. C. Norrie (2012), *A Meta-Plugin for Bespoke Data Management in WordPress*, Proc. Intl. Conf. on Web Information Systems Engineering (WISE'12).
12. S. Leone, A. de Spindler, M. C. Norrie, and D. McLeod (2013), *Integrating Component-based Web Engineering into Content Management Systems*, Proc. Intl. Conf. on Web Engineering (ICWE'13).
13. S. Leone and M. C. Norrie (2011), *Building eCommerce Systems from Shared Micro-Schemas*, Proc. Intl. Conf. on Cooperative Information Systems (CoopIS'11).
14. H. Lieberman, F. Paterno, and V.r Wulf (Eds.) (2006), *End User Development (Human-Computer Interaction Series)*, Springer.
15. N. Medvidovic and R. N. Taylor (2000), *A Classification and Comparison Framework for Software Architecture Description Languages*, IEEE Trans. Softw. Eng., Vol. 26(1) pp. 70-93.
16. S. Murthy, D. Maier, and L. Delcambre (2006), *Mash-o-Matic*, Proc. ACM Symposium on Document Engineering (DocEng'06).
17. M. Shaw (2011), *Modularity for the Modern World: Summary of Invited Keynote*, Proc. Intl. Conf. on Aspect-Oriented Software Development (AOSD'11).