Concern-Specific Languages in a Visual Web Service Creation Environment

Mathieu Braem [1] Niels Joncheere [2] Wim Vanderperren [3]
Ragnhild Van Der Straeten [4] Viviane Jonckers [5]

System and Software Engineering Lab
Vrije Universiteit Brussel
Pleinlaan 2
B-1050 Brussel, Belgium

Abstract

This paper presents a high-level, visual Service Creation Environment (SCE) for web services. The SCE introduces two main concepts: services and composition templates. Composition templates are abstract descriptions of reusable compositions containing several placeholders for services. Services are verified to be compatible with the composition template when a service is mapped onto a composition template. The SCE supports the modularization of crosscutting concerns using both the general-purpose AOP language Padus and several concern-specific languages. Aspects can be visually deployed on a target composition template or service, which automatically triggers the weaving process.

Key words: Service-Oriented Architecture, Concern-Specific Languages, Aspect-Oriented Software Development, Web Services

1 Introduction

Over the last years, *web services* [2] have been gaining a lot of popularity as a means of integrating existing software in new environments. Basic web services can be created by exposing existing applications to the internet using XML front-ends. By composing a number of basic web services, new web services can be created that provide more advanced functionality. These compound web

[1] Email: mbraem@vub.ac.be
[2] Email: njonchee@vub.ac.be
[3] Email: wvdperre@vub.ac.be
[4] Email: rvdstrae@vub.ac.be
[5] Email: vejoncke@ssel.vub.ac.be

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

services can then be used by other web services, further improving software reusability.

Originally, the only way to compose web services was by manually writing the necessary glue-code in programming languages such as C and Java. It quickly became clear, however, that a composition of web services is more naturally captured by dedicated *workflow languages* [14] than by general-purpose programming languages. Today, the most popular workflow language with regard to the composition of web services is the *Business Process Execution Language* (WS-BPEL) [3]. WS-BPEL processes are platform- and transport-independent, and are expressed using XML. Recently, a higher-level visual notation for WS-BPEL, called the Business Process Modeling Notation (BPMN) [36], has been proposed.

Meanwhile, *aspect-oriented software development* (AOSD) has been proposed as a means of improving *separation of concerns* [26] in software. AOSD is based on the observation that a number of concerns in software (such as logging [18] and billing [13]) cannot be modularized using object-oriented software development: a program can only be decomposed in one way (i.e., according to the class hierarchy), and concerns that do not align with this decomposition end up scattered across the program and tangled with one another. This problem is dubbed "the tyranny of the dominant decomposition" [25]. AOSD allows expressing such *crosscutting concerns* in well-modularized *aspects*, so that adding, modifying or removing such concerns does not require changes to the main program.

Initial research on AOSD has concentrated on applying its principles to the object-oriented programming paradigm. Arsanjani *et al.* [4] and others [10,12,35] have shown that AOSD has a lot of potential in a web services context, too.

Although workflow languages are better suited for web service composition than general-purpose programming languages, they still require a large amount of in-depth technical knowledge. In order to facilitate service composition without requiring such in-depth technical knowledge, a higher level of abstraction is required. We therefore propose a visual *service creation environment* (SCE), which allows user-friendly configuration of web service compositions using reusable *composition templates*, and which supports encapsulating crosscutting concerns using both general-purpose and concern-specific aspect languages. This environment is implemented as a plug-in for the Eclipse platform [15].

The outline of the paper is as follows: Section 2 explains the motivation for the SCE, and Section 3 provides an overview of the SCE architecture. Next, the support for concern-specific languages in the SCE is presented. Section 5 describes related work, and Section 6 states our conclusions and future work.

2 Motivation for the Service Creation Environment

The research presented in this paper is conducted in the context of the WIT-CASE project, which is partly funded by Alcatel Belgium, a telecom company, and by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

In the telecom community, the *service delivery platform* (SDP) is a central ICT infrastructure that is targeted at the development, deployment and execution of value-added telecom services by network operators as well as by third-party service providers. Our approach focuses on the development and configuration of service compositions in a visual *service creation environment* (SCE). A number of key requirements for such an environment stated by the telecom partner are:

- A *faster introduction of new services* is needed *through innovative service creation mechanisms* ranging from using open ICT programming environments to service composition tools, reuse of common components and integration of service logic with business applications.
- *Fast and easy modification of service and business logic of baseline services* is required.
- The ability to *offer service bundles to customers as a strategic move against competition* is also required. This requirement implies that common capabilities must be offered on which these services of the service bundles can rely, e.g., common billing and the capability to offer flexible tariff packages for the grouped services.
- A last requirement is the *reduction of operational expenses by service providers*. Therefore, different end-user services should as much as possible be based on generic reusable building blocks. Furthermore, service providers want to build end-user services and applications on top of an integration platform instead of deploying a collection of out-of-the box end-user services and applications. An integration platform offers the additional benefit of integration with legacy network infrastructure.

In order to meet the above stated requirements, the SCE needs to allow the configuration of service compositions on a high level of abstraction in order to facilitate application development without in-depth technical knowledge of the involved services. In order to achieve this, the following objectives are pursued:

- A visual SCE that enables easy plug-and-play composition of both internal and external services.
- The SCE has to guide the service composition process by providing feedback about the correctness of the resulting service composition.
- The SCE has to allow describing management concerns (e.g., billing) of the resulting service composition in a concise and declarative manner.

The current state-of-the-art is insufficient for supporting the envisioned SCE. Typical workflow languages (such as WS-BPEL) provide visual GUIs in order to facilitate the creation of workflows. However, these GUIs are nothing more than a visual interface on top of the language. Examples of such GUIs are BPWS4J [6] and Oracle BPEL Designer [24]. There is no support for guiding the service composition process to a correct service composition. Furthermore, management concerns still have to be encoded in the workflow itself, which results in a workflow that is tangled with several secondary concerns, and as such makes the resulting workflow more complex. In the next section, we introduce our SCE and show how the above stated objectives are met.

3 The Service Creation Environment

In this section, we first introduce the architecture of our visual SCE. Secondly, we explain how the SCE supports AOSD. Next, the GUI of the SCE is presented, followed by an explanation on how services can be composed, on how service compositions can be verified, and finally, on how services can be deployed.

3.1 Architecture of the SCE

Figure 1 gives an overview of the architecture of the SCE. The SCE contains three repositories:

- A first repository contains a set of *documented services*. The services contained in this repository are the basic building blocks of the SCE. Each service is described by a WSDL file. In addition, each service is documented by a WS-BPEL process that specifies the external protocol information and by a description of basic quality of service requirements.
- Another repository contains a set of documented *composition templates*. These templates are specified in WS-BPEL. The templates are abstract descriptions of web service compositions and may contain one or more placeholders for services. The composition templates can be instantiated by filling in the placeholders with different services. When services are added to service composition templates, the SCE checks whether the protocols of the services are compatible with the protocol of the service composition template.
- A third repository contains different *crosscutting concerns* corresponding to management concerns such as billing schemes. A crosscutting concern can be connected to services and composition templates visually or through a pointcut language.

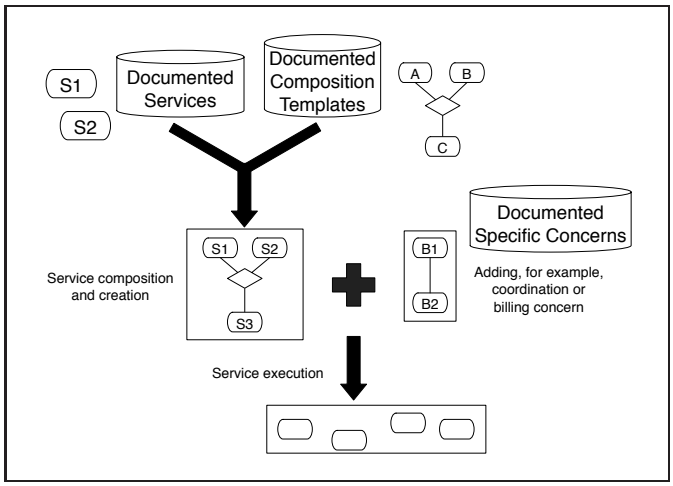

Fig. 1. SCE architecture

3.2 Aspects and Padus

The SCE supports the modularization of crosscutting concerns through the Padus [7] language. Padus is our aspect-oriented process language based on WS-BPEL. A detailed explanation of Padus is outside the scope of this paper. We only introduce and explain the features of Padus relevant for the SCE and for defining concern-specific languages.

Padus is an XML-based language, and introduces two main concepts: *aspects* and *aspect deployments*. An aspect is a reusable description of a crosscutting concern, and contains one or more pointcuts and advice. A pointcut selects interesting points in the execution of the target WS-BPEL process (called joinpoints), and exposes target objects to the advice. The pointcut language of Padus is a logic language based on Prolog, and is thus very expressive [17]. The complete target WS-BPEL process is reified as a collection of facts that can be queried by the pointcut. The advice language is WS-BPEL, extended with some AOSD-specific constructs. The Padus technology is based on a traditional static weaver that processes the target WS-BPEL processes and generates new WS-BPEL processes containing the advice code as specified in our visual environment. The main advantage of this approach is the compatibility with existing infrastructure, as the output can be deployed on any WS-BPEL-compatible engine.

3.3 SCE GUI

Figure 2 provides a screenshot of the SCE's interface. The editor view (in the middle of the screen) is used to edit compositions, and consists of two main parts: a large drawing canvas, and a smaller palette. The palette contains some selection and connection tools, and shows the available services, composition templates and aspects as they are loaded from the library. By double-clicking on an entity, the configured editor for that entity is launched. For instance, a graphical BPMN-based editor is launched for a composition

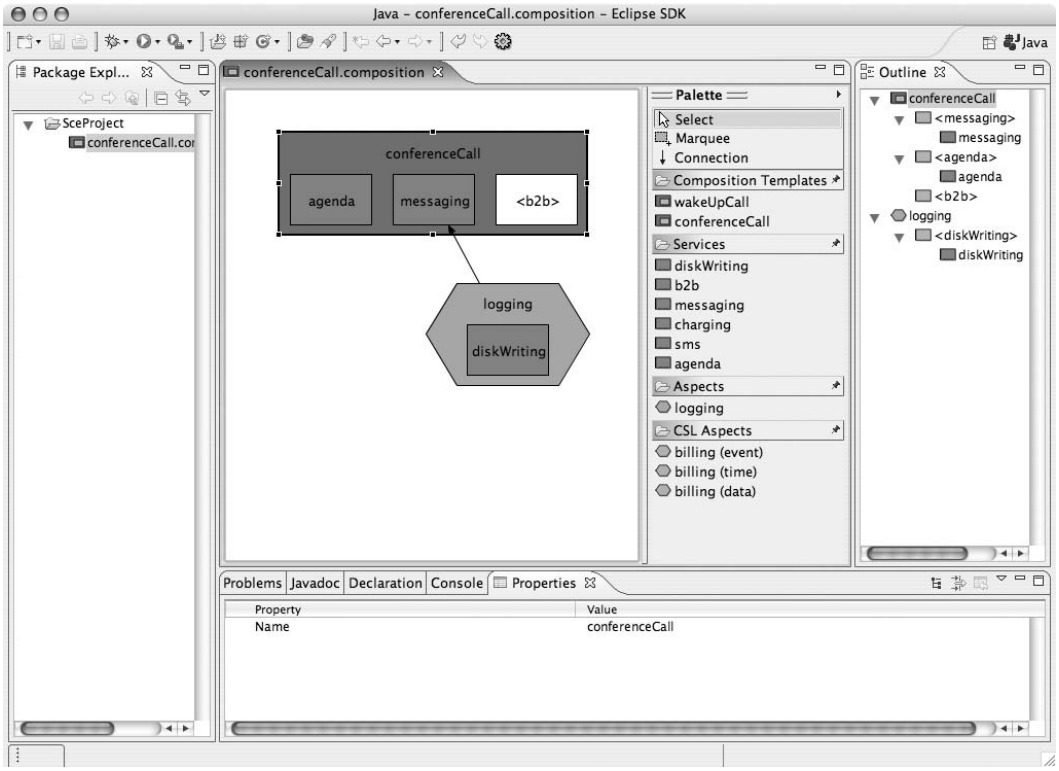

Fig. 2. Screenshot of the SCE's interface

template. The changes made through the visual editor are taken into account for the composition at hand. As such, a composition template can be adapted on the fly and at a higher level of abstraction than WS-BPEL code.

The outline view (at the right of the screen) shows a tree-based overview of the state of the composition, and the properties view (at the bottom of the screen) shows the properties of the element that is currently selected in the editor view or in the outline view.

3.4 Composition

In order to create a composition in the SCE, it suffices to drag a composition template on the composition canvas and fill all the placeholders with concrete services. Aspects can be connected to services, meaning that they will only be applied to these concrete services, or to a complete composition template, in order to apply them to all the services that take part in this composition.

The composition shown in Figure 2 contains a composition template called "conferenceCall" with three placeholders. Two services called "agenda" and "messaging" have been added to the composition template's placeholders, while one placeholder is still empty. This placeholder should be filled in before the composition can work, for instance using the "b2b" service available in the library. The composition also contains an aspect called "logging", which is connected to the "messaging" service. A service called "diskWriting" has

been added to the aspect's only placeholder. The result of this composition would be that the conference call application works using the selected services, and that a logging aspect, which invokes the disk writing service, is deployed to the messaging service in order to log the messaging actions selected by the aspect's pointcut.

3.5 Verification

An important requirement of the SCE is that it guides users in creating correct compositions without requiring in-depth technical knowledge. The SCE accomplishes this by verifying whether compositions are correct while they are created: when a service is dragged onto a placeholder, the SCE checks whether the service's protocol is compatible with the composition template's protocol. If the service turns out to be incompatible, a report is generated that provides mismatch feedback to the user. Compatibility checking based on protocols rather than plain APIs is possible because every service is explicitly documented with a protocol specification expressed in WS-BPEL.

In literature, a wealth of research exists on the topic of protocol verification [9,21,33,27,38]. Our verification engine is based on the PacoSuite approach [37], which introduces algorithms based on automata theory to perform protocol verification. In order to provide protocol verification in the SCE, the WS-BPEL specifications of each service, aspect and composition template are translated into deterministic finite automata (DFA). By applying the algorithms introduced by the PacoSuite approach, the SCE can decide whether the service's protocol is compatible with the composition template's protocol.

In case a certain service is not compatible with a certain composition template placeholder, the user has two options: Either select another service for that placeholder, or edit the composition template on the fly as described in Section 3.3.

3.6 Code Generation and Deployment

When the composition is complete and verified, the user may choose to generate the resulting composition and deploy it on a WS-BPEL engine. This will start the code generation process, which will bind the unbound partner links in the composition templates. An aspect deployment is automatically generated for the aspects contained in the composition. The Padus weaver is then employed to weave the aspects into the resulting WS-BPEL processes based on the aspect deployment specification.

A resulting composition can also be imported back into the library as a new service. The generated WS-BPEL process then serves as documentation for the new service. Apart from specifying a name and some other properties, this process is also automated.

The SCE also includes a built-in WS-BPEL engine that can be used to immediately execute a resulting composition. This feature is meant to be able

to quickly assess the result rather than to be the real deployment target. We are currently working on improving the integration of this engine, so that it can be used as a debugger for compositions by providing feedback directly to the SCE.

4 Concern-Specific Languages

Aspect-oriented principles are supported by the SCE through the use of the Padus aspect-oriented programming language. One or more aspects describe a concern, and are written in Padus. If a user of the SCE wants to express aspects, the only possibility is to specify these aspects in Padus, which requires in-depth knowledge of the Padus language. This is in contradiction with our research objective, which states that the SCE should allow the description of management concerns in an intuitive, concise and declarative manner. Therefore, we need the ability to visually specify concerns on a higher level of abstraction.

The earliest aspect-oriented programming languages are each developed for a particular crosscutting concern; we name these *concern-specific languages* (CSLs). Examples of these early CSLs are COOL [19,20], a language for expressing the aspect of synchronization for programs written in Java, and RIDL [19,20], a language for expressing the aspect of data serializability in distributed environments. A recent concern-specific language is KALA [16]. KALA is a powerful aspect language for describing the use of advanced transaction models by an application, which also allows new models to be defined if needed. However, since a new language has to be devised for each concern, construction of concern-specific languages was quickly deemed too costly. Instead, more approaches opt for general-purpose aspect languages, such as AspectJ [18].

Our objective is to enable the definition of concern-specific languages on top of the Padus technology and integrated in the SCE. The implementation of a crosscutting concern is thus defined in a specific CSL, built on Padus. To apply the concern to the service process or service composition process, a user can select the relevant aspects, add them to the service process and concretize them.

The remainder of this section contains an example of a concern-specific language: Billing. First, we define this language, and next, we show how this language is integrated in the SCE and how it can be used in a concrete service composition.

4.1 The Need for a Billing Language

Billing is a concern that occurs in many systems. It can be as simple as deducting a fixed fee from a client's account after the execution of an operation, but it can also require complicated schemes based on the client's location, the

```
1  <concern language="billing" type="time" name="billcall">
2
3      <!-- specify when billing should occur: -->
4      <start when="invoking(Service, Port, 'connect', User)" />
5      <end when="invoking(Service, Port2, 'disconnect', User)" />
6
7      <!-- specify what should be charged: -->
8      <advice>
9          <begin> <charge type="setup" context="User" /> </begin>
10         <success> <charge type="time" context="User, $Time" /> </success>
11         <fail> <!-- do nothing --> </fail>
12         <finally> <!-- do nothing --> </finally>
13     </advice>
14 </concern>
```

Listing 1: Billing example

client's account type, which operation was executed, how long it took, etc.

We recognize two important patterns in the billing concern. On the one hand there is the issue of *when* billing starts and ends. On the other hand there is the issue of *what* should be charged. In our approach we separate these two parts. Our dedicated Billing language selects the points in a process execution where billing starts and ends, and allows us to add extra behavior at each of these points. Typically, we pass the information about the operation and associated timestamps to a dedicated charging service. This service keeps a complete log of all charged events. At a later time, a program may collect these logs and create bills for the customers, possibly affected by business rules. This is the issue of what should be charged, and can greatly vary on the context of the events. Therefore, the Billing CSL exposes the context of the process events to a large extent.

4.2 Definition of Billing

The Billing language allows expressing billing concerns in dedicated XML-based modules, which are specified separate from the main functionality of service compositions. Listing 1 provides an example of such a module.

The main element of a Billing module is the `concern` element. Its attributes specify the language and the type of the module. In our example, line 1 specifies that the module is specified using the `billing` language, and that its type is `time`. Modules that are expressed using another concern-specific language would also contain a `concern` element, but its `language` attribute would indicate that another language is used, and that its contents are thus different than those of a Billing module.

There are three types of Billing modules: *event-based* modules are used to perform billing based on events that occur during the execution of a service (e.g., when a text message has been sent), *time-based* modules are used to perform billing based on the time that has passed between two events (e.g., between the start and the end of a telephone call), and *data-based* modules are used to perform billing based on the volume of data that has been exchanged

between two services.

The children of the `concern` element specify *when* billing should occur, and *what* should be charged. Because our example is a time-based module, it specifies both when billing should start (using the `start` element in line 4) and when billing should end (using the `end` element in line 5). The `when` attributes of the `start` and `end` elements are Padus pointcuts that select certain points in the execution of a service.

Each module specifies what should be charged in the `advice` element. This element has four children: the `begin` element specifies what should be done when the concern is activated, the `success` element specifies what should be done when the concern terminates successfully, the `fail` element specifies what should be done when an exception is thrown while the concern is active, and the `finally` element specifies what should be done when the concern terminates, regardless of whether it terminates successfully or not.

Each of these four elements may contain regular WS-BPEL code in order to perform the charging. Alternatively, one may use the Billing language's dedicated `charge` element in order to perform the charging without writing WS-BPEL code. In our example, line 9 sends a message to the charging service which specifies that the user has started a connection, and line 10 sends a message to the charging service which specifies that the user has ended a connection with a certain duration. In the advice code, variables that were bound in the Padus pointcut may be used. Additionally, we expose some context of the process by means of the `$Time` variable. In the example, the duration of the call is retrieved from the concern's context using the `$Time` variable and passed to the charging process.

In order to perform the actual charging, each Billing module contains an implicit partner link that refers to a charging service. This partner link will be employed when the `charge` element is used in the module's advice, and it can be linked to a concrete service using the SCE's interface (see Section 4.3). If one wants to use another partner link or more than one partner link, this can be specified in the optional `using` child of the `concern` element.

4.3 Visualization of Billing

In Section 3, we illustrate how the SCE can be used to create new compositions, by dragging a composition template on the canvas, filling its placeholders with services, and adding an aspect to a service. This aspect is retrieved from a repository of crosscutting concerns, and is implemented using the Padus language. Using the SCE, it is straightforward to change when such aspects are applicable or which services are used by the aspect. However, changing what the aspect actually does (i.e., changing the aspect's advice) requires in-depth knowledge of the Padus language. Therefore, the SCE also allows adding concern-specific aspects, which allow specifying an advice without in-depth knowledge of Padus.

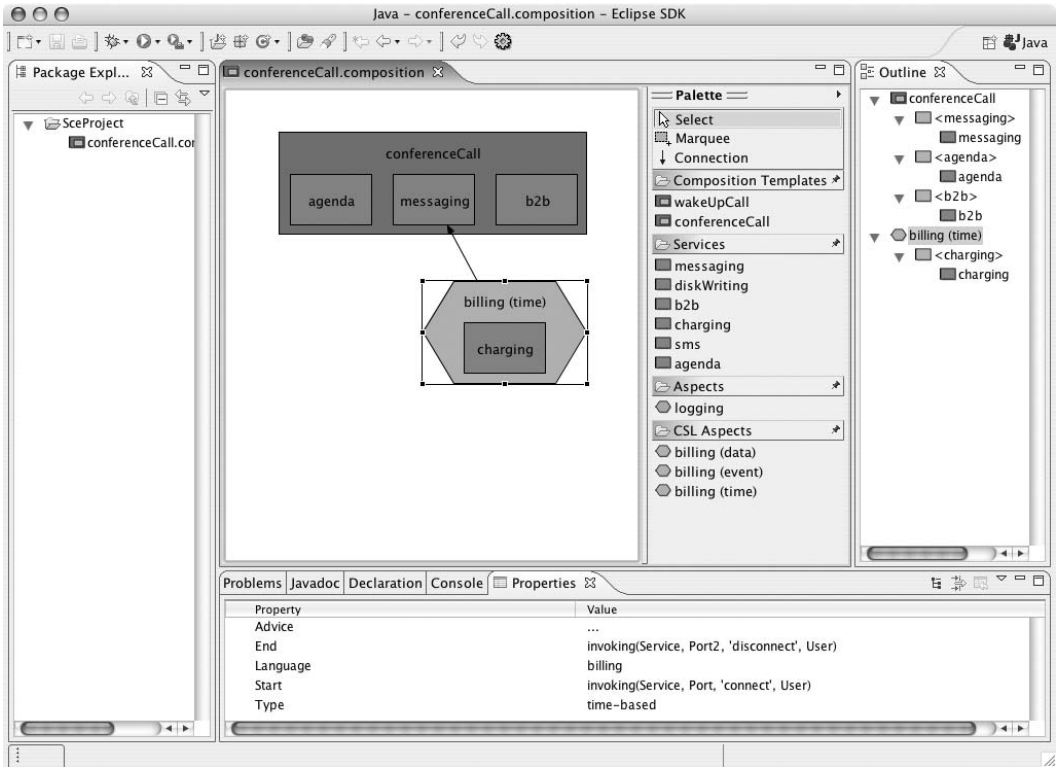

Fig. 3. A composition that contains a concern-specific aspect

Figure 3 provides an example of a composition that contains such a concern-specific aspect. The example is the same as the one in Figure 2, but the logging aspect has been replaced by a concern-specific billing aspect.

The palette contains a library of templates for concern-specific aspects, which may be instantiated by dragging them on the canvas. The palette in the example contains three such templates, i.e., "billing (time)", "billing (event)" and "billing (data)", which correspond to the three types of billing that are identified above. Each Billing aspect has at least one placeholder, which allows binding the implicit partner link that was mentioned above to a concrete web service.

When a concern-specific aspect is selected in the editor view, its properties appear in the properties view. A time-based billing aspect, for example, has five properties: "language", "type", "start", "end", and "advice". The first two properties simply show the language and the type of the aspect, respectively. The other properties, however, can be changed in order to define between which two points in the execution of the composition billing should occur, and how this billing should be achieved. Based on this information, most of the information for the corresponding Billing module (such as the one in Listing 1) is generated by the SCE.

When the composition in the editor view is complete, any concern-specific aspects are translated to Padus aspects, which are then applied to the appropriate services and/or composition templates similar to regular compositions.

5 Related Work

Several visual component composition environments already exist in the context of component-based software development (CBSD). CBSD advocates reusable and loosely-coupled components in order to realize flexible plug-and-play component composition of off-the-shelf components [31]. The main problem in CBSD is that wiring components together requires writing glue-code manually in order to resolve syntactic and semantic incompatibilities. A visual component composition environment allows to visually compose the components and supports the (semi-)automatic generation of glue-code that implements the composition. Current practice component composition environments, such as VisualAge for Java from IBM, JBuilder from Borland and BeanBuilder from Sun already allow some form of automatic glue-code generation from a given component composition. The main difference with our approach, apart from the focus on components instead of web services, is that they do not support a reusable encapsulation of composition logic. Furthermore, there is no support for verifying whether a certain composition is possible apart from syntactically checking messages and arguments. Another disadvantage is that they do not support modularizing crosscutting concerns.

Documenting components with protocol documentation is already well investigated in literature. Campbell and Habermann [9] introduced the idea of augmenting interface descriptions with sequence constraints already in 1974. More recent work includes the Rapide system [21] or the PROCOL system [33]. In the research area of component based software development, several component composition environments are available that lift the abstraction level for component composition. Yellin and Strom [38], Reussner's CoCoNut project [27] and PacoSuite [37] for example also employ automata to document components. PacoSuite is one of the most advanced component composition environments and supports higher-level component composition based on sequence charts. The main advantage with respect to the other work on protocol verification is that PacoSuite supports multi-party connectors, whereas other approaches typically only support binary connectors. The PacoSuite approach is, however, domain dependent, and is only targeted at the simple JavaBeans component model.

BPMN is a graphical notation for specifying workflows, and aims to become the de facto graphical standard similar to WS-BPEL for workflow languages. BPMN allows for a higher-level graphical notation for processes in comparison to WS-BPEL, and is in fact complementary to our approach. A BPMN-based editor that is able to import/export WS-BPEL can for instance be used to edit the specification of a composition template. As soon as there is a standardized file format for BPMN, the SCE can also directly support BPMN for the documentation of services and composition templates, instead of or next to WS-BPEL.

Several approaches exist that focus on the construction of concern-specific

languages, also referred to as domain-specific languages. Note that these languages do not have facilities for encapsulating crosscutting concerns and are hence not aspect languages. Such approaches provide environments for the more efficient and scalable construction of languages fit to express concepts from a particular domain. Examples are Draco [23], GenVoca [5], Babel [8] and Intentional Programming [28].

Agarwal *et al.* [1] present a service creation environment based on end-to-end composition of web services, but this environment does not allow visual composition of web services nor separation of concerns using aspect-oriented techniques.

6 Conclusions and Future Work

In this paper, we present a high-level service creation environment for composing web services. Our approach supports the modularization of crosscutting concerns through Padus aspects. Padus aspects can be visually deployed onto composition templates or services. Furthermore, support for concern-specific languages on top of Padus is available.

On the abstraction scale we situate our SCE on the same level as BPMN and the Padus language. The SCE is an advanced tool for configuring service compositions and augmenting them with separated concerns, by means of Padus aspects or higher-level concern-specific languages.

Our work is still in an early phase and as such several improvements are possible:

- Our approach supports visually deploying aspects onto concrete services. The pointcuts still have to be defined programmatically in Padus. Describing pointcuts at a higher level of abstraction would be an important contribution to our work. We are experimenting with existing pointcut visualizations such as Theme/UML [11], Join Point Designation Diagrams [30,29] and AOSF [22] to solve this problem.
- It is possible that an aspect adapts the external protocol of an existing service (e.g., by adding an invocation) so that it becomes incompatible with the composition template's protocol. Currently, our tool is not able to cope with this problem. In order to solve this, we are planning to include the aspect protocol documentation and verification algorithms proposed by *composition adapters* [34].
- The support for integrating concern-specific languages is currently quite limited. Apart from a set of common tool (such as XML parsing and transformation tools) and a simple visualization template, defining and implementing a new concern-specific language still largely happens in an ad hoc manner. A more in-depth solution based on existing work (such as Babel) is subject to future work.
- The repository of available composition templates, services and aspects is

a custom solution and limited to local files. In the future, we plan to investigate support for the industrial standard for discovery of web services called UDDI [32].

Acknowledgments

This research is partly funded by Alcatel Belgium and the Institute for the Promotion of Innovation Through Science and Technology in Flanders (IWT-Vlaanderen) through the WIT-CASE project.

References

[1] Agarwal, V., K. Dasgupta, N. Karnik, A. Kumar, A. Kundu, S. Mittal and B. Srivastava, *A service creation environment based on end to end composition of web services*, in: *Proceedings of the 14th International World Wide Web Conference (WWW 2005)* (2005), pp. 128–137.

[2] Alonso, G., F. Casati, H. Kuno and V. Machiraju, editors, "Web Services: Concepts, Architectures and Applications," Springer-Verlag, Heidelberg, Germany, 2004.

[3] Andrews, T., F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic and S. Weerawarana, *Business Process Execution Language for Web Services, version 1.1* (2003).
URL http://www.ibm.com/developerworks/library/ws-bpel/

[4] Arsanjani, A., B. Hailpern, J. Martin and P. Tarr, *Web services: Promises and compromises*, Queue **1** (2003), pp. 48–58.

[5] Batory, D., V. Singhal, J. Thomas, S. Dasari, B. Geraci and M. Sirkin, *The GenVoca model of software-system generators*, IEEE Software **11** (1994), pp. 89–94.

[6] *BPWS4J.*
URL http://www.alphaworks.ibm.com/tech/bpws4j

[7] Braem, M., K. Verlaenen, N. Joncheere, W. Vanderperren, R. Van Der Straeten, E. Truyen, W. Joosen and V. Jonckers, *Isolating process-level concerns using Padus*, in: *Proceedings of the 4th International Conference on Business Process Management (BPM 2006)* (2006), (to appear).

[8] Brichau, J., "Integrative Composition of Program Generators," Ph.D. thesis, Programming Technology Lab (PROG), Vrije Universiteit Brussel, Brussels, Belgium (2005).

[9] Campbell, R. and A. Habermann, *The specification of process synchronisation by path expressions*, in: *Proceedings of an International Symposium on Operating Systems*, 1974, pp. 89–102.

[10] Charfi, A. and M. Mezini, *Aspect-oriented web service composition with AO4BPEL*, in: L.-J. Zhang, editor, *Proceedings of the 2nd European Conference on Web Services (ECOWS 2004)* (2004), pp. 168–182.

[11] Clarke, S. and E. Baniassad, "Aspect-Oriented Analysis and Design — The Theme Approach," Addison-Wesley, 2005.

[12] Cottenier, T. and T. Elrad, *Dynamic and decentralized service composition with Contextual Aspect-Sensitive Services*, in: *Proceedings of the 1st International Conference on Web Information Systems and Technologies (WEBIST 2005)*, Miami, FL, USA, 2005, pp. 56–63.

[13] D'Hondt, M. and V. Jonckers, *Hybrid aspects for weaving object-oriented functionality and rule-based knowledge*, in: K. Lieberherr, editor, *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)* (2004), pp. 132–140.

[14] Du, W. and A. Elmagarmid, *Workflow management: State of the art vs. state of the products*, Technical Report HPL-97-90, Hewlett-Packard Labs, Palo Alto, CA, USA (1997).

[15] *The Eclipse platform.*
URL http://www.eclipse.org/

[16] Fabry, J., "Modularizing Advanced Transaction Management — Tackling Tangled Aspect Code," Ph.D. thesis, Programming Technology Lab (PROG), Vrije Universiteit Brussel, Brussels, Belgium (2005).

[17] Gybels, K. and J. Brichau, *Arranging language features for pattern-based crosscuts*, in: M. Akşit, editor, *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)* (2003), pp. 60–69.

[18] Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. G. Griswold, *An overview of AspectJ*, in: J. L. Knudsen, editor, *Proc. ECOOP 2001, LNCS 2072* (2001), pp. 327–353.

[19] Lopes, C. V., "D: A Language Framework for Distributed Programming," Ph.D. thesis, College of Computer Science, Northeastern University (1997).

[20] Lopes, C. V. and G. Kiczales, *D: A language framework for distributed programming*, Technical Report SPL-97-010, Palo Alto Research Center (1997).

[21] Luckham, D., J. Kenney, L. Augustin, D. Vera, D. Bryan and W. Mann, *Specification and analysis of system architecture using Rapide*, IEEE Transactions on Software Engineering **21** (1995).

[22] Mahoney, M., A. Bader, T. Elrad and O. Aldawud, *Using aspects to abstract and modularize statecharts*, in: O. Aldawud, G. Booch, J. Gray, J. Kienzle, D. Stein, M. Kandé, F. Akkawi and T. Elrad, editors, *The 5th Aspect-Oriented Modeling Workshop In Conjunction with UML 2004*, 2004.

[23] Neighbors, J. M., *Draco: A method for engineering reusable software systems*, in: *Software Reusability — Concepts and Models*, ACM Press, New York, NY, USA, 1989 pp. 295–319.

[24] *Oracle BPEL Process Manager*.
URL
http://www.oracle.com/technology/products/ias/bpel/index.html

[25] Ossher, H. and P. Tarr, *Using subject-oriented programming to overcome common problems in object-oriented software development/evolution*, in: *Proc. 21st Int'l Conf. Software Engineering* (1999), pp. 687–688.

[26] Parnas, D. L., *On the criteria to be used in decomposing systems into modules*, Comm. ACM **15** (1972), pp. 1053–1058.

[27] Reussner, R. H., *Automatic component protocol adaptation with the CoCoNut tool suite*, Future Generation Computer Systems **19** (2003), pp. 627–639.

[28] Simonyi, C., *The death of programming languages, the birth of intentional programming*, Technical report, Microsoft, Inc. (1995).

[29] Stein, D., S. Hanenberg and R. Unland, *Query models*, in: *UML '04: Proceedings of the international conference on the Unified Modelling Language* (2004), pp. 98–112.

[30] Stein, D., S. Hanenberg and R. Unland, *Expressing different conceptual models of join point selections in aspect-oriented design*, in: *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development* (2006), pp. 15–26.

[31] Szyperski, C., "Component Software: Beyond Object-Oriented Programming," ACM Press and Addison-Wesley, New York, NY, USA, 1998.

[32] *UDDI*.
URL http://www.uddi.org/

[33] van den Bos, J. and C. Laffra, *PROCOL: A concurrent object-oriented language with protocols delegation and constraints*, Acta Informatica **28** (1991), pp. 511–538.

[34] Vanderperren, W., D. Suvee and V. Jonckers, *Combining AOSD and CBSD in PacoSuite through invasive composition adapters and JAsCo*, in: *Proceedings of the Net.ObjectDays 2003 International Conference*, 2004, pp. 35–50.

[35] Verheecke, B., W. Vanderperren and V. Jonckers, *Unraveling crosscutting concerns in web services middleware*, IEEE Software **23** (2006), pp. 42–50.

[36] White, S. A., *Business Process Modeling Notation (BPMN), version 1.0* (2004).
URL http://www.bpmn.org/

[37] Wydaeghe, B., "PacoSuite: Component Composition Based on Composition Patterns and Usage Scenarios," Ph.D. thesis, System & Software Engineering Lab (SSEL), Vrije Universiteit Brussel, Brussels, Belgium (2001).

[38] Yellin, D. M. and R. E. Strom, *Protocol specifications and component adaptors*, ACM Transactions on Programming Languages and Systems **19** (1997), pp. 292–333.