Improving Software Reliability in Data-centered Software Systems by Enforcing Composition Time Constraints

Lieven Desmet, Frank Piessens, Wouter Joosen, Pierre Verbaeten
DistriNet research group, Department of Computer Science
K.U.Leuven, Celestijnenlaan 200A, B-3001 Leuven, Belgium
Lieven.Desmet@cs.kuleuven.ac.be

Abstract

Specifying and enforcing constraints and invariants such as architectural constraints and data typing, strongly enhances the safety and reliability of the software system. Next to design and development constraints, the composition of software systems in component-based software also introduces composition time constraints and dependencies. In data-centered software systems, for example, the software composer implicitly creates dataflow dependencies between software components. Describing composition time constraints and enforcing these constraints at deploy time or at run-time strongly improves the safety and reliability of the software. In this paper, we present an approach for expressing and enforcing dataflow dependencies in data-centered software systems, and conclude with a validation of the approach in a servlet-based case study.

1. Introduction

Nowadays, software systems become more and more mission critical. Software failures can cause a lot of loss and damage as business processes strongly depend on the availability and the correct functioning of software systems.

Besides, software systems evolve towards modularly composed applications, in which existing software components are reused within new compositions [13]. Recent software systems even enable dynamic reconfiguration of the running application [6].

Furthermore, building manageable, large-scale software systems introduces several development constraints and invariants, such as architectural constraints and data typing. Specifying and enforcing constraints and invariants has already proven to greatly enhance the safety and reliability of the software system (e.g. type systems [10, 1] and ADL's[3]).

Within component-based software development, also composition time constraints are introduced by the software composer. For example, indirect interactions between software components, such as implicit invocation in event-based communication or indirect data sharing through data-centered repositories, create composition time dependencies between software components.

In our opinion, keeping such constraints and dependencies implicit within a software system (as is often is done in state of the art systems today), increases the risk for software failure due to violated constraints or broken dependencies, especially in a dynamically reconfigurable system. Therefore, we promote making composition time constraints and dependencies explicit within the software system, in order to improve the safety and reliability.

In the following section one specific composition time constraint is introduced, namely dataflow dependencies. Next, a simple servlet-based case study is presented in section 3, in order to demonstrate the safety enhancement of the composition by making the dataflow dependencies explicit (section 4). Section 5 briefly summarizes the status of our work and overviews future investigations.

2. Data flow dependencies

In the data-centered architectural style [11], a system consists of a central data structure (representing the state of the system) and a set of separate components interacting with the central data store. This architectural style is quite commonly used in several component models. Sometimes, further refinements are introduced to this style. However, in practice, most of the refinements or constraints are only modeled implicitly and are not checked at run-time.

The components of a data-centered software system describe a *required* and *provided* dataset, specifying the set of data that a component fetches from or puts onto the shared repository.

A correct composed data-centered application is a collection of separate components and a shared repository, with respect to functional data dependencies: every required data item of a component is provided by another compo-

nent by means of the shared repository. In figure 1, different dataflows within a data-centered application are explicitly shown, while the actual component interactions (control flow) are abstracted.

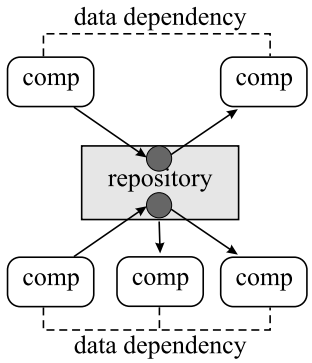

Figure 1. Functional dataflow dependencies

Besides functional dataflow dependencies, also non-functional requirements on the dataflows may exist. In other words extra constraints on the dataflows through the shared repository may be expressed. These constraints can address for instance the authenticity of the dataflow, confidentiality or synchronization.

Also more general constraints, such as splitting up the shared repository in several disjunct logical repositories, or protecting the repository against name clashes between several dataflows are possible extra composition requirements in data-centered applications.

In our opinion, all data dependencies should be explicitly modeled in data-centered application to improve safety and reliability of the composition. Furthermore, those data dependencies should be clearly separated from the core application functionality in order to improve reusability, adaptability and manageability.

3. Case study

In this section, the dataflow dependencies of a small servlet-based e-commerce application are thoroughly analyzed. Firstly, servlet-based technology is briefly summarized. Next, the functionality of the example case is examined, in order to study the dataflow dependencies within the case.

3.1. Servlet-based webcontainers

The Java Servlet technology is part of the J2EE specification and provides a mechanism for extending the functionality of a Web server and for accessing existing business systems [7, 5]. Servlet-based webcontainers offer infrastructural support for using servlets.

The core functionality of the container is to handle incoming webrequests and to use (chains of) servlets for processing the requests. In general, servlets are pure functional units of the web tier, and extra-functional properties such as load-balancing and security can be added to the webcontainer. The web deployment descriptor (`web.xml`) of a webcontainer contains the deployment information of the web application, including extra-functional properties and a list of servlets with their corresponding url mapping.

Within a web application, servlets are loosely coupled with each other and support for dispatching between servlets is provided by the webcontainer. However, servlets can communicate anonymously by means of a shared data repository. In fact, five instances of shared repositories are provided to the servlet: a data repository associated with the dynamic webpage (1), with the webrequest (2), with the user session (3), the webcontext (4) and the application (5). Hence, servlet-based applications are data-centered compositions, and the application composer must pay special attention to the dataflow dependencies.

3.2. A small e-commerce site

Figure 2 illustrates a simple e-commerce web application. Three different services are identified within the application: adding a product item to the personal shopping basket, the payment of the shopping order and searching through the website. Each box represents a functional task implemented as a servlet, and the services are pipe-and-filter compositions of several independent tasks.

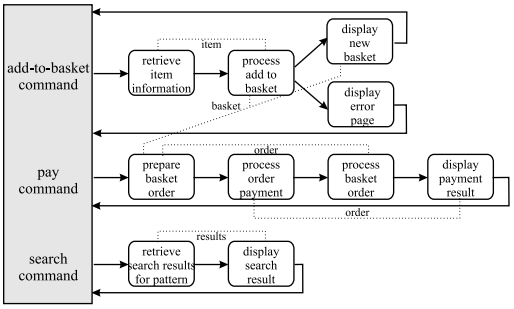

Figure 2. A small e-commerce web application

Adding a product item to the shopping basket starts with retrieving the necessary item information from the data back-end *(retrieve item information)*, next the item is added

to the shopping basket *(process add to basket)*, and depending on the success or failure of adding the product to the basket, the new basket *(display new basket)* or an error page *(display error page)* is displayed to the end user.

Similarly, the payment service is decomposed into a first servlet constructing an order out of the shopping basket entries *(prepare basket order)* and a second servlet *(process order payment)*, taking care of the payment (e.g. the submission of creditcard information). Next the order is logically processed at the server *(process basket order)* and the payment result is displayed to the end user *(display payment result)*.

In processing search requests, a third-party servlet retrieves pages conform the given search pattern *(retrieve search results for pattern)*. Next, the search results are displayed *(display search results)*.

3.3. Data flow dependencies

Although the servlets are implemented as independent components, dataflow dependencies exists between these servlets. The dataflow dependencies are depicted in figure 2 by dotted lines.

For each website user, a personal shopping basket is saved at server-side, in the session scope of the shared repository. The personal basket is created at a user's first visit. It is used by three different servlets, entitled in the figure as 'process add to basket', 'display new basket' and 'prepare basket order'. Furthermore, in order to prevent race conditions, the composed application needs also extra synchronization support for the shopping basket (figure 3(a)). A simple synchronization scenario might be: a lock to the basket is defined, which must be obtained by the 'process add to basket' 'prepare basket order' servlet before execution and is this lock is released by the 'display new basket' and 'prepare basket order servlet after execution.

Within the payment service, two dataflow dependencies are present. Between the first and third servlet, the payment order (constructed from the current shopping basket) is shared for further processing. The second and fourth servlet share the result of the payment transaction. Since the servlets are developed separately (e.g. the payment component is typically a third-party component), a conflict exists in the naming of the shared data. Therefore, extra support is needed to prevent the name clash (figure 3(b)).

Other dataflow dependencies are the sharing of the retrieved item information in the add-to-basket service chain, and of the search results in the search service.

Moreover, two additional general constraints are expressed on the dataflow of this example. Firstly, although there is only one shared repository (with five scope levels), the three services are grouped into two separate parts, each requiring a (logically) separate shared repository (figure 3(c)). Secondly, since the third-party search servlet can't be trusted completely, all unspecified access from the search servlet to the shared repository must be prevented (figure 3(d)).

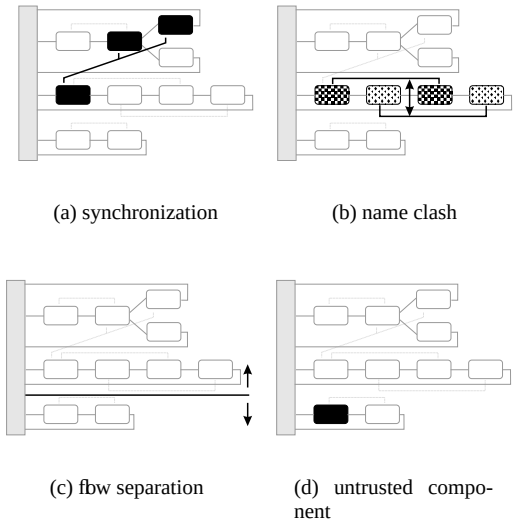

Figure 3. Data flow dependency constraints

4. Explicit support

As stated in section 1, the reliability of an application can be enhanced by explicitly describing and enforcing composition time constraints and dependencies, that currently exists implicitly. Therefore, in order to introduce specific support for implicit data dependencies, we have identified the following approach in which three requirements can be defined. Firstly, functional components within the application need an explicit specification extension of the required and provided data items for each component. Secondly, composing an application requires a declarative policy of the functional and non-functional dataflow dependencies between the components. Finally, the enforcement of this declarative policy is needed, either at deployment time or at run-time.

4.1. Extended specification

Traditionally, an operation is syntactically and semantically specified based on the operation's name and its input and output [2]. In order to express data dependencies, this specification must be extended with extra information about the data items that are provided or required from the shared repository by the component's operation.

Extending the specification with repository interactions can be achieved either manually by the component's de-

signer or implementor, or generated by tools that are based on the component's implementation.

In servlet-based web application, the web descriptor could be extended to provide the needed extra specification of the servlets. The necessary specification for this case study is illustrated in figure 4: the servlet definition is extended with a provided and required data set, described by the name of the shared data item, the data type and the sharing scope.

```
<servlet>
   <servlet-name>itemRetriever</servlet-name>
   <servlet-class>be.comp.RetrieveItem</servlet-class>
   <data-provision>
      <data-name>item</data-name>
      <data-class>be.comp.ItemBean</data-class>
      <scope>request</scope>
   </data-provision>
</servlet>
<servlet>
   <servlet-name>basketDisplayer</servlet-name>
   <servlet-class>be.comp.DisplayBasket</servlet-class>
   <data-requirement>
      <data-name>basket</data-name>
      <data-class>be.comp.ItemBasket</data-class>
   </data-requirement>
</servlet>
```

Figure 4. Extended specification

4.2. Declarative policy

The composition of an application with a shared repository requires more than defining the functional components within the application and the corresponding control flow. The application composer also needs to define the dataflow by means of functional data dependencies and the extra constraints on the dependencies.

Moreover, to enhance adaptability and manageability, the dataflow should be described in a separate, declarative policy.

The description of the dataflow policy for the case study as an extension to the web descriptor is presented in figure 5. A dependency is described by listing the providers and consumers of the data, extended with extra-functional characteristics. In this example, a representation of the simple synchronization scenario is included.

4.3. Policy enforcement

To enforce the dataflow policy at run-time, additional support is needed for controlling the access to the shared repository. As such, the shared repository can be extended with an enforcement engine. Alternatively, a wrapper with a built-in enforcement engine can encapsulate all access to the shared repository.

```
<dependency>
   <provider>
      <servlet-name>ecommerceInitializer</servlet-name>
      <data-name>basket</data-name>
   </provider>
   <consumer>
      <servlet-name>addToBasket</servlet-name>
      <data-name>basket</data-name>
   </consumer>
   <consumer>
      <servlet-name>basketDisplayer</servlet-name>
      <data-name>basket</data-name>
   </consumer>
   <consumer>
      <servlet-name>orderPreparator</servlet-name>
      <data-name>basket</data-name>
   </consumer>
   <synchronization>
      <lock-obtainer>addToBasket</lock-obtainer>
      <lock-obtainer>orderPreparator</lock-obtainer>
      <lock-releaser>basketDisplayer<lock-releaser>
      <lock-releaser>orderPreparator<lock-releaser>
   </synchronization>
</dependency>
```

Figure 5. Declarative policy

The enforcement engine decides whether a component is allowed or denied access to a data item on the shared repository (functional data dependency). In addition, the enforcement engine ensures the non-functional constraints on the considered dataflow.

In order to correctly enforce the dataflow policy, the enforcement machine may also need some additional state information of the running application.

In the case study presented in section 3, a wrapper is used to intercept access to the shared repository in every scope. Furthermore, the enforcement engine firstly lacked the ability to retrieve the identity of the calling servlet, although this identity was clearly needed for proper access control to the shared repository. Various solutions were found to provide this additional information to the enforcement engine, such as using stack walks, or explicitly recording the active servlet for every thread within the webcontainer.

In the presented approach, both functional dataflow dependencies and the extra constraints are clearly separated from core functionality and existing specification of the different components. Furthermore, the data dependency concern is easily adaptable through the use of the declarative policy, and together with the enforcement engine a high cohesion and low coupling is achieved. Finally, by explicitly expressing implicit dependencies and enforcing those dependencies, the robustness and reliability of the composed application is enhanced.

5. Summary

Currently, the approach is validated in a simple servlet-based case study by means of an early stage prototype implementation. Hereby, the specification and policy language

used, is still in development. The goal of making implicit composition assumptions explicit and therefore enhancing the safety and reliability of the composition is reached, especially in dynamically reconfigurable systems.

Also other efforts, such as the Struts framework [12], aim at providing extra support for building reliable and reconfigurable servlet-based applications (e.g. declarative composition description). To our knowledge however, no project currently support describing and enforcing dataflow dependencies, and these efforts are hence complementary with the presented work.

Next to the presented case study, the approach of making dataflow dependencies explicit has also been validated in the component-based protocol stack framework DiPS. In DiPS [4, 9, 8] functional components are chained into a pipe-and-filter structure and components can share data anonymously along the pipe by means of a shared repository. Similar results were achieved within this case study.

Future work will expand the current approach to other implicit constraints and dependencies. Target tracks are the study of dataflow dependencies in Message Driven Beans, and implicit invocations in event-based systems.

Furthermore, the integration of specifying composition time constraints into existing composition and deployment tools needs further investigation. Composition tools can easily be extended to suggest most of the functional dependencies to the software composer, in order to facilitate the composer's effort of specifying the declarative policy. However, tool support able to suggest extra-functional constraints to the software composer remains an open question.

References

[1] J. Aldrich. *Using Types to Enforce Architectural Structure*. PhD thesis, University of Washington, August 2003.

[2] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. In *ieee-software*, pages 38–45, june 1999.

[3] P. Clements. A survey of architecture description languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*, March 1996.

[4] K. DistriNet Research Group. DiPS home page. http://www.cs.kuleuven.ac.be/cwis/research/-distrinet/projects/DIPS/.

[5] J. Hunter and W. Crawford. *Java Servlet Programming*. O'Reilly, second edition, April 2001.

[6] N. Janssens, S. Michiels, T. Mahieu, and P. Verbaeten. Towards Hot-Swappable System Software: The DiPS/CuPS Component Framework. In *Proceedings - The Seventh International Workshop on Component Oriented Programming*, 2002.

[7] Java servlet technology. http://java.sun.com/products/servlet/.

[8] S. Michiels, L. Desmet, N. Janssens, T. Mahieu, and P. Verbaeten. Self-adapting concurrency: The DMonA architecture. In D. Garlan, J. Kramer, and A. Wolf, editors, *Proceedings of the First Workshop on Self-Healing Systems (WOSS'02)*, pages 43–48, Charleston, SC, USA, 2002. ACM SIGSOFT, ACM press.

[9] S. Michiels, F. Matthijs, D. Walravens, and P. Verbaeten. Position summary: DiPS: A Unifying Approach for Developing System Software. In A. D. Williams, editor, *Proceedings - The Eigth Workshop on Hot Topics in Operating Systems (HoTOS-VIII*, page 175. University of Karlsruhe, IEEE Computer Society Press, 2001.

[10] A. Myers. JFlow: Practical mostly-static information fbw control. In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[11] M. Shaw and D. Garlan. *Software Architecture - Perspectives on an emerging discipline*. Prentice-Hall, 1996.

[12] The struts framework. http://jakarta.apache.org/struts/.

[13] C. Szyperski, D. Gruntz, and S. Murer. *Component Software - Beyond Object-Oriented Programming*. Addison Wesley Professional, 2nd edition, November 2002.