Towards Encapsulating Data in Component-Based Software Systems

Kung-Kiu Lau and Faris M. Taweel

School of Computer Science,
The University of Manchester,
Manchester, M13 9PL, UK
{kung-kiu, faris.taweel}@cs.manchester.ac.uk

Abstract. A component-based system consists of components linked by connectors. Data can reside in components and/or in external data stores. Operations on data, such as access, update and transfer are carried out during computations performed by components. Typically, in current component models, control, computation and data are mixed up in the components, while control and data are both communicated by the connectors. As a result, such systems are tightly coupled, making reasoning difficult. In this paper we propose an approach for encapsulating data by separating it from control and computation.

1 Introduction

A software system consists of three elements: *control*, *computation*, and *data*. The system's behaviour is the result of the interaction between these elements. Therefore, the latter determines whether the system has desirable properties such as loose coupling and ease of analysis and reasoning. It is reasonable to expect that it is advantageous to encapsulate these elements, and separate them from one another, since this should make reasoning more tractable. For example, some recent research in stimulus reactive systems has focused on separating control flow from data flow [9, 5].

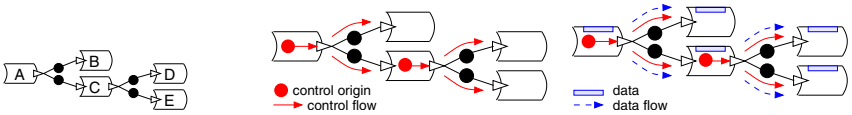
Paradoxically perhaps, for component-based software systems, it is not any easier to achieve such separation of concerns. A component-based system consists of components linked by connectors, as exemplified by software architectures [17]. In such a system, data can reside either in components or in external databases (which are often also regarded as components). Operations on data, such as access, update and transfer are carried out during computations performed by components. Typically, in current component models, control, computation and data are mixed up in the components, while control and data are both communicated by the connectors. As a result, such systems are tightly coupled, making reasoning difficult. More seriously, this impedes component reuse, which is a key objective for component-based development.

In this paper we introduce an approach for encapsulating data by separating it from control and computation, in component-based systems. Our approach is based on our earlier work on encapsulating control and computation in component-based systems [10].

I. Gorton et al. (Eds.): CBSE 2006, LNCS 4063, pp. 376–384, 2006.

2 Data in Current Component Models

In current component models, computation, control and data are intermixed (Fig. 1). Components initiate control and perform computation (Fig. 1(b)). Connectors provide communication between components for both control and data (Fig. 1(c)). Some



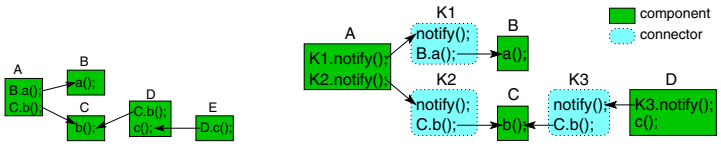
(a) Components and connectors (b) Control flow (c) Data flow

Fig. 1. Current component models

components may act as (external) databases, but for simplicity we will treat them all as components.

In these models, data exists in components, and during the computation performed by a component, data can be accessed from other components, updated, and transferred to other components. Component models with external data stores provide special abstractions modelling these sources. For example, EJB [4, 15] provides the *entity bean* that connects to an external database; .NET [14] provides the *Dataset component*, and CCM [16] provides *persistent components*. .NET takes an extra step by providing a database connector called the .NET Data Adaptor Framework [14]. However, in these models, data is not separated from computation or control.

In general, connection schemes in current component models use message passing, and fall into two main categories:[1]. (*i*) connection by direct message passing; and (*ii*)



(a) Direct message passing (b) Indirect message passing

Fig. 2. Connection by message passing

connection by indirect message passing. Direct message passing corresponds to direct method calls, as exemplified by objects calling methods in other objects (Fig. 2 (a)), using method or event delegation, or remote procedure call (RPC). Component models that adopt direct message passing schemes as composition operators are EJB, CCM, COM [2] etc.

Indirect message passing corresponds to coordination (e.g. RPC) via connectors, as exemplified by ADLs. Here, connectors are typically glue code or scripts that pass

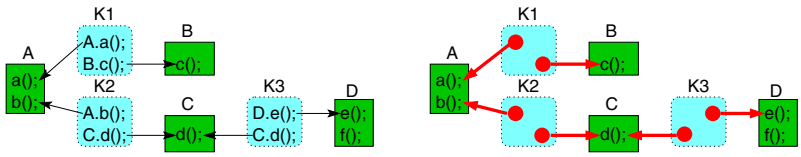
[1] For a survey, see [11].

messages between components indirectly. A connector, when notified by a component invokes a method in another component (Fig. 2 (b)). Besides ADLs, other component models that adopt indirect message passing schemes are JavaBeans [8], Koala [19], SOFA [1] etc.

In connection schemes by message passing, direct or indirect, control originates in and flows from components (Fig. 1(b)). This is clearly the case in both Fig. 2(a) and (b). Furthermore, data resides in components, and as components perform computation (invoked by message passing), data flows between components in tandem with control flow (message passing) between them (Fig. 1(c)). This is the case in both Fig. 2(a) and (b), although data is not shown explicitly. Clearly in current component models, neither control nor data is encapsulated.

3 Encapsulating Computation and Control

We are developing a component model [10] in which components encapsulate *computation* (unlike objects or port-connector type components, Fig. 2), and connectors encapsulate *control* (unlike current component models, Fig. 1). In our model, components do not call methods in other components, and control originates in and flows from connectors, leaving components to encapsulate only computation. This is illustrated by Fig. 3. We call our connectors *exogenous connectors*. Fig. 3 (a) shows an example



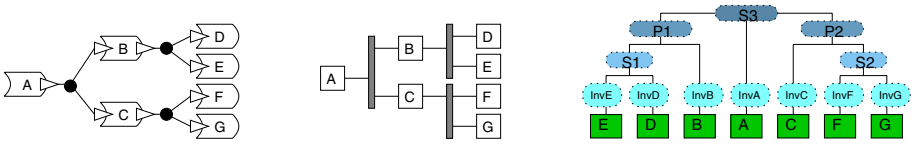
(a) Example (b) Control flow

Fig. 3. Connection by exogenous connectors

of exogenous connection. Here components do not call methods in other components. Instead, all method calls are initiated and coordinated by exogenous connectors. The latter thus encapsulate control, as is clearly illustrated by Fig. 3 (b), in contrast to Fig. 1 (b). Exogenous connectors thus encapsulate control, i.e. they *initiate* and *coordinate* control.

Exogenous connectors are hierarchical in nature, with a type hierarchy [10]. At the bottom of the hierarchy, because components are not allowed to call methods in other components, we have an exogenous *method invocation connector*. This is a *unary* operator that takes a component, invokes one of its methods, and receives the result of the invocation. To structure the control flow in a set of components or a system, at the next level of the type hierarchy, we have other connectors for sequencing exogenous method calls to different components. So we have n*-ary* connectors for connecting invocation connectors, and n*-ary* connectors for connecting these connectors, and so on. As well as invocation connectors, we have defined and implemented *pipe* connectors, for sequencing, and *selector* connectors, for branching.

Example 1. Consider a system whose architecture can be described in the Acme [6] and C2 [18] ADLs by the architectures in Fig. 4 (a) and (b) respectively. Using exogenous connectors in our component model, the corresponding architecture is that shown in Fig. 4 (c). In the latter, the lowest level of connectors are unary invocation connectors



(a) Acme (b) C2 (c) Exogenous connection

Fig. 4. Corresponding architectures

that connect to single components, viz., *InvA*, *InvB*, . . . ; the second-level connectors are binary selector connectors *S1* and *S2* and connect pairs of invocation connectors; and the connectors at levels 3 and 4 are of variable arities and types: *P1* and *P2* are pipes, while *S3* is a selector.

4 Encapsulating Data

In our component model, a system consists of a hierarchy of exogenous connectors sitting atop a flat layer of components, as illustrated in Fig. 4(c). The components encapsulate computation while the connectors encapsulate control (Fig. 5). Since it is the

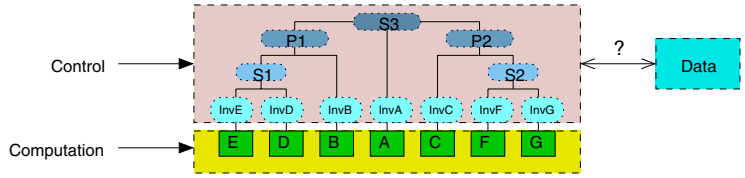


Fig. 5. Encapsulating data

only model we know of that has these two kinds of encapsulation, we believe our component model is a good basis for attempting to encapsulate data as well. So the question we want to address in this paper is how we can encapsulate data in our component model (Fig. 5).

In our component model, components already encapsulate *local data*, like objects in object-oriented languages. However, for *system* or *global state* we have not provided any encapsulation. To clearly illustrate the challenge we face, consider what happens if data flow follows control flow, as in current component models (Fig. 1).

Example 2. The architecture in Fig. 4(c) can be used to represent a bank system, with A being an *ATM*, B and C two bank consortia, each containing two bank branches,

E and D, and F and G respectively. At level 1, each component has its own invocation connector. The ATM (A) can display a menu, accept a customer card, read its information, accept customer requests such as choice of operations (deposit, withdrawal, check balance, etc.).

At level 2, the selector connector $S1$ selects the customer's bank branch (from *E* and *D*), prior to invoking that branch's methods requested by the customer. Similarly, the selector connector *S2* chooses between *F* and *G*, prior to invoking their methods requested by the customer. To pass values from one bank consortium (B or C) to one of its bank branches, we need a pipe connector; so at level 3, we have two pipe connectors *P1* and *P2*, for consortia *B* and *C* respectively. At level 4, the selector connector *S3* selects the customer's bank consortium from consortia *B* and *C*, after receiving customer requests and card information from the ATM (A). Thus, the bank system's operational cycle is initiated by passing the customer requests and card information from the ATM (A) to the connector *S3*.[2]

We will show what happens if data flow follows control flow. Initially, data comes with a customer request to the ATM. This data comprises customer details on his card (customer identification, account number, bank consortium identification code, etc.), the customer's choice of transaction (say withdrawal), and the amount involved.

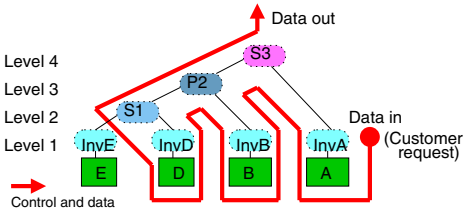


Fig. 6. Control flow and data flow of the bank system

This data flows with control until it reaches the customer's bank branch where it is used for processing the requested transaction. Fig. 6 traces the data (and control) flow for an example where the customer bank branch is D, belonging to consortium B. At *S3*, the bank consortium code is used to determine the next connector, in this case *P2*. All the data, except the bank consortium code, is then passed on to *P2*. Using the customer account number, *P2* invokes *B* to identify the customer bank branch, which is D in this case. So the bank branch code together with the rest of data is passed to $S1$. Here, the bank branch D is selected and the customer's request, account number and amount are passed to D, and the customer's request is processed.

The main observation here is that, apart from the bank consortium identification code, customer data traverses a long route, with the control flow, to its final destination. This property is not unique to exogenous connectors, but also pertains to any component model with explicit connectors where data flow follows control flow, e.g. Fig. 4(a) and (b). It clearly raises performance and space concerns.

[2] A pipe selector should be used to connect A and *S3*, but for simplicity, we omit it here.

As clearly illustrated by Example 2, rather than leaving global data to flow with control, it is desirable to separate the two. To this end, it is necessary to store data in a shared *data space* (a repository), and to allow access to such data via *data connectors* (Fig. 5) at suitable points in the control flow. Furthermore, for data access operations to be generic against the data space, the type of data stored in data spaces must be *generic* with respect to the programming environment. A generic data type for a programming environment is a type that subsumes all its other data types. We call this type the type *Universal* to be independent from different names provided by different programming languages.

The data space is a transient data store that every system must have to maintain its global state. In a data space, data is created and accessed indirectly via data connectors. A data element stored in a data space has many properties including a unique string *reference*; a *state* indicating whether it is transient or persistent; and a set of *access permissions* specifying which exogenous connector in the system can read (r), write (w), persist (p) and execute (x) this data element. Any data element is a collection element that can contain executable string code such as SQL data manipulation statements. To execute these elements, they must have the execute (x) permission.

Data connectors link data elements stored in (global) data spaces to exogenous connectors as well as to other data connectors. In effect, they *set*, *get* and *execute* data elements in the data space; and encapsulate data access coordination and data flow in a system. They are hierarchical in nature. At the bottom of the hierarchy, because exogenous connectors only encapsulate control they cannot access (global) data directly, we need a *data accessor connector*. This is a unary data connector that can link to only one data element in the data space; and store, return and execute that specific data element. In order to structure flow and access coordination for a set of data elements, at the next level of the hierarchy, we have other connectors for sequencing access and execution of data elements. Accordingly, we have n*-ary* data connectors for connecting accessor data connectors, and n*-ary* data connectors for connecting these connectors, and so on.

Data connectors and exogenous connectors are both connectors However, they form two separate kinds of hierarchies that can be connected at design time at specific points, where data values are required by exogenous connectors.

Example 3. To explain our approach for encapsulating data, we consider Example 2 again. In particular, we consider the case in (Fig. 6). Here the data flow can be partitioned into three paths: (i) customer's request and card information passing through *InvA, S3, P2* and *InvB*; (ii) data passing through *InvB, P2, S1* and *InvD*; and (iii) data passing information back to the ATM (A), where the returned data can be, for example, a bank statement.

To prevent data from flowing along these paths in the hierarchy of exogenous connectors, we store the customer's request and card information in the system's data space, as three data elements (bank consortium code, bank branch code and transaction data), and use data connectors to link these data elements to exogenous connectors in which they are needed (Fig 7). As a result, data flows separately from control. At the start of the system's operation, data is exported to the data space by *InvA*, then at *S3* only the first element (consortium code) is retrieved to evaluate the next pipe connector. At the connector *P2*, control is passed to *InvB* which in turn uses the second element (branch

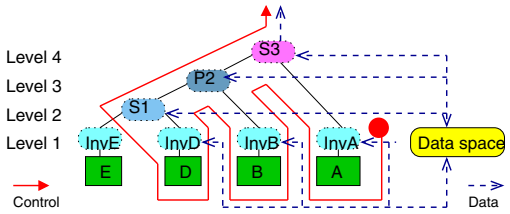


Fig. 7. Data connectors for the bank system

code) to determine the customer's bank branch. When control reaches $S1$, the bank branch identity is used to direct control to *InvD*. *InvD* retrieves from the data space the third data element (transaction data), and passes it by value to *D*, to process the requested service. *InvD* stores a report in the data space that is ultimately accessed by *S3* and displayed to the customer. Thus, instead of data following control, we have separated data from control.

This example shows only the basic idea of data connectors, but not their hierarchies, i.e. data connectors that compose other data connectors. So, while this example may give the impression that data connectors are just a means of handling global data, the whole picture about our data connectors will only become complete when composition operations among them are addressed.

Nevertheless, Example 3 does illustrate how in our approach data no longer flows with control. In fact, what flows in exogenous connectors is just a unique signal identifying the required service. Data is encapsulated in data spaces; and its flow and access coordination in data connectors. This results in not only reduced coupling between components (and even connectors), but also enables data structuring as well as compositional operations on data.

5 Evaluation

In present approaches to software connectors, control and data are mixed up in connectors [13]. Some approaches even allow for the parameterisation of computation for connectors [12]. Current component models are no exceptions. Either they provide no explicit connectors (EJB, CCM, COM, etc), or have connectors that mix data and control (Koala, SOFA, ADLs, etc). Furthermore, an example of a basic set of connectors that can be composed systematically is yet to be seen. As connectors encapsulate interactions among components, data and control flowing together does not make the analysis of a system any easier. It has been demonstrated that separation of data and control makes the analysis of systems' properties easier [9, 5]. Compared to these approaches, the novelty in our work lies in the total separation and hence encapsulation of control, computation, and data. Our motivation is to make component-based development more amenable to analysis, reasoning, and above all reuse. Our data connectors lead to loose coupling between components, and therefore contribute to our objective.

Compared to other component models, our data connectors are unique in that they encapsulate data completely. Compared to data processing languages, the hierarchy of

our data connectors is also unique in that they provide a structured way of accessing and processing data (by composition). As a result, our work should provide a useful basis for developing data-intensive applications which can benefit from component-based solutions.

Using our data connectors, we can distinguish between global data, i.e. data shared by all the components in the whole system, and data that is shared by a specific subset of components in the system. This is not just a distinction between global data and local data as in current programming languages, e.g. object-oriented languages, but it is a distinction between local data for individual components and shared data between a composite component, or a sub-system, made up of sub-components. It is thus a finer distinction, and more importantly, it allows us to encapsulate data at different levels of granularity, i.e. at the level of individual components, at the level of composite components, or at the level of the whole system. For database applications, we believe such encapsulation provides a new dimension.

Our data space is similar to tuple spaces in generative communication and coordination languages [7, 3]. But we are different in using explicit connectors to data spaces.

6 Conclusion

In this paper we have proposed a way to encapsulate data in a component model that already encapsulated control and computation. The resulting model is, to the best of our knowledge, the first model with encapsulation of control computation *data*. However, the work is preliminary, and serves to demonstrate the feasibility of our idea. More practical evaluation will be needed, and in future work, we intend to do such evaluation.

Nevertheless, compared to related work, viz. component models with explicit connectors, our work seems unique in that global data is really separated from control and computation. In fact, using our connectors we can differentiate between global data, i.e. data global to the whole system, and data shared by specific components.

We also have operations for structuring and composing data, making our approach potentially very useful for data-intensive applications, which also benefit from component-based solutions.

References

1. Dusan Balek. Connectors in software architectures, 2002.
2. Don Box. *Essential COM*. Addison-Wesley, Harlow, 1998.
3. Nicholas Carriero and David Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, 1989.
4. Linda G. DeMichiel, editor. *Enterprise JavaBeans Specification, Version 2.1*. Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A, November 12 2003.
5. Berndt Farwer and Mauricio Varea. Object-based control/data-flow analysis. Technical Report DSSE-TR-2005-1, University of Southampton, Department of Electronics and Computer Science, Highfield, Southampton SO17 1BJ, United Kingdom, March 2005.
6. David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

7. David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.
8. Graham Hamilton, editor. *JavaBeans*. Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A, August 8 1997.
9. Ouassila Labbani1, Jean-Luc Dekeyser1, and Pierre Boulet1. Mode-automata based methodology for scade. In Manfred Morari and Lothar Thiele, editors, *Hybrid Systems: Computation and Control, 8th International Workshop, HSCC 2005*, volume LNCS 3414, pages 386–401, Berlin Heidelberg, March 9-11 2005. Springer-Verlag GmbH.
10. K.-K. Lau, Perla I. Velasco, and Zheng Wang. Exogenous connectors for components. In *Eighth International SIGSOFT Symposium on Component-based Software Engineering (CBSE'05)*, May 2005.
11. Kung-Kiu Lau and Zheng Wang. A survey of software component models. Survey CSPP-30, The University of Manchester, Manchester, UK, April 2005.
12. Antnia Lopes, Michel Wermelinger, and Jos Luiz Fiadeiro. Higher-order architectural connectors. *ACM Trans. Softw. Eng. Methodol.*, 12(1):64–104, 2003.
13. Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a taxonomy of software connectors. In *ICSE*, pages 178–187, 2000.
14. Microsoft. *Data access development overview: within the Microsoft Enterprise Development Platform*. Microsoft Enterprise Development Strategy Series. Microsoft, http://msdn.microsoft.com/netframework/technologyinfo/entstrategy/default.aspx, March 2005.
15. Richard Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, Farnham ; Sebastopol, Calif., 4th ed. edition, 2004.
16. OMG. *CORBA Component Model, V3.0*. Object Management Group, http://www.omg.org/docs/formal/02-06-69.pdf, 2002.
17. Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
18. Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead Jr., Jason E. Robbins, Kari A. Nies, Peyman Oreizy, and Deborah L. Dubrow. A component- and message-based architectural style for GUI software. *Software Engineering*, 22(6):390–406, 1996.
19. Rob C. van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, 2000.