Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 163 (2006) 57–69
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

A Component Model for Separation of Control Flow from Computation in Component-Based Systems

Kung-Kiu Lau, Vladyslav Ukis, Perla Velasco and Zheng Wang

School of Computer Science, The University of Manchester
Manchester M13 9PL, United Kingdom
Email: kung-kiu, vukis, velascop, zw@cs.man.ac.uk

Abstract

Today's component models as well as architectural description languages (ADLs) compose components either using direct or indirect method calls. When using direct method calls, components carry out computation, originate control to and perform communication between each other. When using indirect message calls, components are connected using connectors encapsulating communication between them. The components in these (ADL) systems are supposed to do computation only. However, in this paper we show that components in ADLs not only perform communication as intended but also originate control towards connectors resulting again in a mixture of control and computation inside components. To separate control from computation in component-based systems we have been developing a new component model aimed at separation of control from computation [15]. In this paper we show how it can be used to build modular and maintainable systems and argue that our component model has its place in Model-driven architecture.

Keywords: Component Model, Indirect Message Calls, Separation of Control Concerns from Computation.

1 Introduction

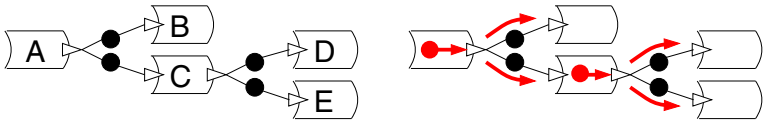
In component-based software development [25], composition is a central issue. Architecture description languages (ADLs) [24] provide connectors as composition operators. However, traditional ADLs do not separate computation (components) from interaction (connectors) as cleanly as intended, thus mixing two semantically different concerns and complicating architectural reasoning. Components not only perform computation, but also initiate control, which is then passed by the connectors to other components. To separate computation from control and to make compositional reasoning more tractable, we believe it is necessary to improve encapsulation of computation (components) as well as control (connectors). Therefore we have been developing a component model with component composition operators called exogenous connectors for component composition. These connectors provide

1571-0661/$ – see front matter
doi:10.1016/j.entcs.2006.07.003

composition mechanisms different from those in existing component models (including ADLs) [16,17], in that they completely capture control, leaving components to encapsulate only computation. In this paper, we present how we separate computation from control and join them together in a system as well as point out the properties of the resulting systems.

2 Separation of Control from Computation

A component model defines components and composition operators to connect them. In our component model components do solely computational tasks. To compose components together in a system we have special composition operators, exogenous connectors, whose distinguishing characteristic is that they encapsulate control in the system. By having this, we can fully separate control from computation in a component-based system. This is in contrast to traditional ADLs,



(a) Components and connectors (b) Control flow

Fig. 1. Traditional ADLs.

where components are supposed to represent *computation*, and connectors *interaction* between components [18] (Figure 1 (a)). Actually, however, components represent computation as well as *control*, since control originates in components, and is passed on by connectors to other components. This is illustrated by Figure 1 (b), where the origin of control is denoted by a dot in a component, and the flow of control is denoted by arrows emanating from the dot and arrows following connectors.

In this situation, components are not truly independent, i.e. they are tightly coupled, albeit only indirectly via their ports.

In general, component connection schemes in current component models (including ADLs) use message passing, and fall into two main categories: (i) connection by direct message passing; and (ii) connection by indirect message passing. Di-

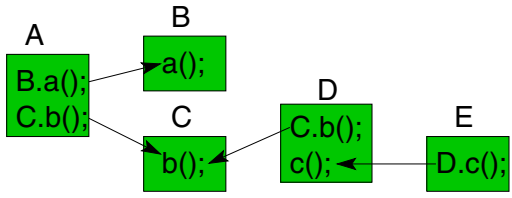


Fig. 2. Connection by direct message passing.

rect message passing corresponds to direct method calls, as exemplified by objects calling methods in other objects (Figure 2), using method or event delegation, or remote procedure call (RPC). Software component models that adopt direct message

passing schemes as composition operators are Enterprise JavaBeans [10], CORBA Component Model [22], COM [3], UML2.0 [21] and KobrA [2]. In these models, there is no explicit code for connectors, since messages are 'hard-wired' into the components, and so connectors are not separate entities.

Indirect message passing corresponds to coordination (e.g. RPC) via connectors, as exemplified by ADLs. Here, connectors are separate entities that are defined explicitly. Typically they are glue code or scripts that pass messages between components indirectly. To connect a component to another component a connector

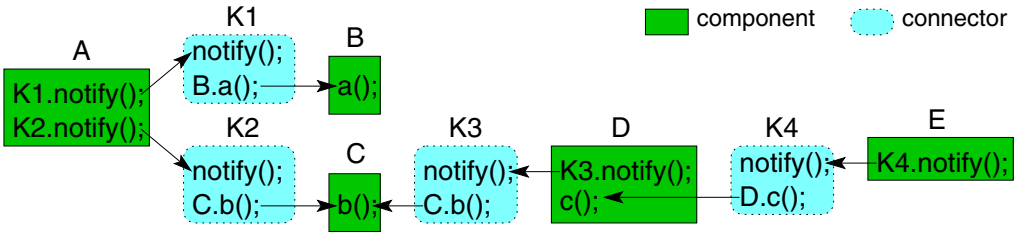


Fig. 3. Connection by indirect message passing.

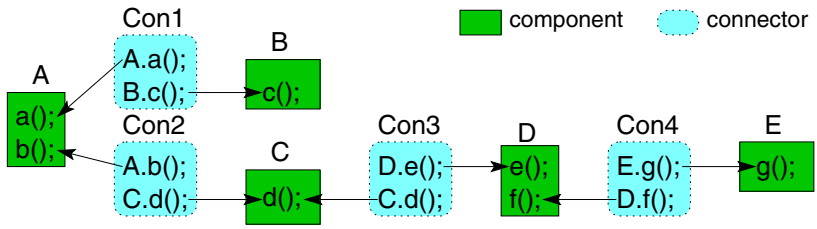
is used that when notified by the former invokes a method in the latter (Figure 3). Besides ADLs, other software component models that adopt indirect message passing schemes are JavaBeans [5], Koala [27], SOFA [23], PECOS [20], PIN [12] and Fractal [4].

In connection schemes by message passing, direct or indirect, control originates in and flows from components, as in Figure 1 (b). This is clearly the case in both Figure 2 and Figure 3.

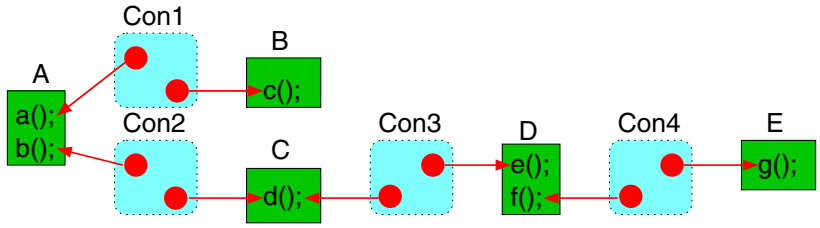
By contrast, in exogenous connection, control originates in and flows from connectors, leaving components to encapsulate only computation. This is illustrated by Figure 4. In Figure 4 (a), components do not call methods in other components. Instead, all method calls are initiated and coordinated by exogenous connectors. The latter's distinguishing feature of control encapsulation is clearly illustrated by Figure 4 (b), in clear contrast to Figure 1 (b).

Exogenous connectors thus encapsulate control (and data), i.e. they *initiate* and *coordinate* control (and data). With exogenous connection, components are truly independent and decoupled resulting in a system with separated control and computation.

Exogenous connection [15] is not provided by any existing software component models (including ADLs). However, exogenous connection has been defined as exogenous coordination in coordination languages for concurrent computation [1]. Also, in object-oriented programming, the courier pattern [7] uses the idea of exogenous connection whereby a courier object links a producer-consumer pair of objects by calling the *produce* method in the producer object and then calling the *consume* method in the consumer object with the result of the *produce* method. The courier pattern doesn't define a hierarchy, though.



(a) Example



(b) Control flow

Fig. 4. Connection by exogenous connectors.

2.1 Hierarchy of Control

The concept of exogenous connection entails a type hierarchy of exogenous connectors. Because they encapsulate all the control in a system, such connectors have to connect to one another (as well as components) in order to build up a complete control structure for the system. For this to be possible, there must be a type hierarchy for these connectors. Therefore such a hierarchy must be defined for any component model that is based on exogenous connection. In this section we describe the connector type hierarchy for our component model.

In our component model,[1] components are units of computation linked by exogenous connectors. A component is a unit of software with (i) an *interface* that specifies the services it provides (i.e. its methods) and the services it requires, and the dependencies between the two sets of services; and (ii) *code* that implements the provided services. In essence it is similar to Szyperski's definition [25]. However, our components do not invoke methods or services in other components. Rather, they only perform their provided services (methods) when they are invoked from outside, by connectors. Thus our components encapsulate computation only.

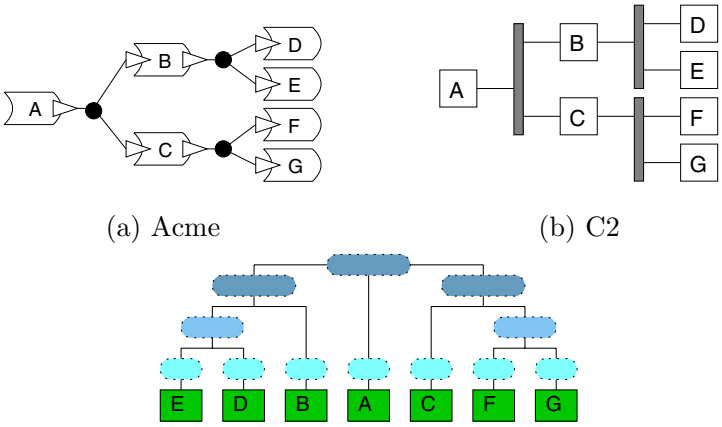
Connectors are composition operators that compose components into systems. They are exogenous, i.e. they initiate and coordinate method calls in components, and handle their results. Thus they determine control flow and data flow, i.e. they encapsulate communication in general, and control in particular.

In the connector type hierarchy for our component model, components are obviously a basic type. Because components are not allowed to call methods in other components, we need an exogenous *method invocation connector*. This is a *unary* operator that takes a component, invokes one of its methods, and receives the result

[1] We do not give a full description; it is not necessary here.

of the invocation.

To structure the control and data flow in a set of components or a system, we need other connectors for sequencing exogenous method calls to different components. So we need *n-ary* connectors for connecting invocation connectors, and *n-ary* connectors for connecting these connectors, and so on. In other words, we need a hierarchy of connectors of different arities and types.



(a) Acme (b) C2

(c) Exogenous connection

Fig. 5. Corresponding architectures.

For example, consider a system whose architecture can be described in the Acme [8] and C2 [26] ADLs by the architectures in Figure 5 (a) and (b) respectively. Using exogenous connectors in our component model, the corresponding architecture is that shown in Figure 5 (c). In the latter, the lowest level of connectors are unary invocation connectors that connect to single components; the second-level connectors are binary and connect pairs of invocation connectors; and the connectors at levels 3 and 4 are of variable arities and types. Note that at the top level, there is only one connector.

In general, connectors at any level other than the first can be of variable arities; connectors at any level higher than two can be of variable arities *and* types; and we can define any number of levels of connectors. Connectors at level n for any $n > 1$ can be defined in terms of connectors at levels 1 to $(n-1)$, according to the following type hierarchy:

Basic types	Component, Result;		
Connector types	$L1 \equiv$ Invocation	$\equiv$	Component $\longrightarrow$ Result;
	$L2$	$\equiv$	$L1 \times \ldots \times L1 \longrightarrow$ Result;
	$L3$	$\equiv$	$L \times \ldots \times L \longrightarrow$ Result
			where L is either $L1$ or $L2$;
	...		

Thus level-one and level-two connectors are not polymorphic since they can connect

only to invocation connectors, but connectors at higher levels are. They can connect to any kind of connectors.

More formally, for an arbitrary number n of levels, the connector type hierarchy can be defined in terms of dependent types and polymorphism as follows:

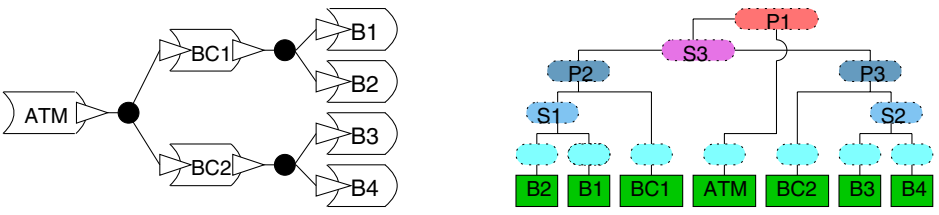
$$
\begin{aligned}
L1 &\equiv \text{Component} \longrightarrow \text{Result};\\
L2 &\equiv L1 \times \ldots \times L1 \longrightarrow \text{Result};\\
\text{For } 2 < i \leq n, \quad Li &\equiv L(j_1) \times \ldots \times L(j_m) \longrightarrow \text{Result, for some } m\\
&\quad \text{where } j_k \in \{1, ..., (i-1)\} \text{ for } 1 \leq k \leq m,\\
&\quad \text{and } L(i) = \begin{cases} L1 \ , \ i = 1 \\ L2 \ , \ i = 2 \\ \vdots \\ Ln \ , \ i = n. \end{cases}
\end{aligned}
$$

2.2 Component Composition

Just as exogenous connection entails a connector type hierarchy, so the latter in turn entails a strictly hierarchical way of constructing systems by composing components. As illustrated by Figure 5 (c), in such a system, components form a flat layer, and the entire control structure (of connectors) sits on top of this. Beyond level 1, the precise choice of connectors, the number of levels of connectors, and the connection structure, depend on the relationship between the behaviour of the individual components and the behaviour that the whole system is supposed to achieve. Whatever the control structure, however, it is strictly hierarchical, which means that there is always only one connector at the top level. This is the connector that initiates control flow in the whole system.

2.2.1 The Bank Example

Consider a bank system, whose architecture is described in Acme in Figure 6 (a). The system has just one ATM that serves two bank consortia ($BC1$ and $BC2$),



(a) Acme (b) Exogenous connection

Fig. 6. Architecture of the bank example.

each with two bank branches ($B1$ and $B2$, $B3$ and $B4$ respectively). The ATM

passes customer requests together with customer details to the customer's bank consortium, which in turn passes them on to the customer's bank branch. The bank branches provide the usual services of withdrawal, deposit, balance check, etc.

The Bank System's architecture in Figure 6 (b) is a refinement of Figure 5 (c). At level 1, each component has an invocation connector. At level 2, there is a selector connector $S1$ that is used to select the customer's bank branch from banks $B1$ and $B2$, prior to invoking that branch's methods requested by the customer. Similarly, there is a level-2 selector connector $S2$ for choosing between $B3$ and $B4$, prior to invoking their methods requested by the customer. To pass values from one bank consortium to one of its banks we need a pipe connector; at level 3, we have two pipe connectors $P2$ and $P3$, for $BC1$ and $BC2$ respectively. At level 4, $S3$ is a selector connector that selects the customer's bank consortium from consortia $BC1$ and $BC2$. Finally, at level 5, the top level, the pipe connector $P1$ initiates the bank system's operational cycle by passing customer requests and card information to the ATM, invoking the ATM's methods, and then passing the resulting value to connector $S3$.

3 Joining Control and Computation

In addition to their hierarchical nature, exogenous connectors can also be implemented in a generic manner. That is, application-*independent* templates for these connectors can be created, which can be reused for different applications by creating application-specific instances. These generic exogenous connectors can be deposited in a repository and retrieved on demand for each application. Furthermore, for any specific application with an exogenous control or connection structure, the generic connectors can be instantiated, on the fly, into the instances in the latter's connection structure. This means that it is possible to generate the control flow of a system dynamically and automatically from its architecture.

To illustrate this, consider the connection structure of the Bank example in Figure 6 (b). The system contains three pipe connectors and three selector connectors (as well as seven invocation connectors). Each of these connectors hosts different connector types (and in different numbers). For example, the pipe $P1$ hosts a selector $S3$ and an invocation connector $I4$ for the component ATM, whereas the pipe $P2$ hosts a selector $S1$ and an invocation connector $I3$ for the component $BC1$. Although the two pipes are doing completely different things, they have been constructed from the same template. The template is generic enough to embody different instances. So, $P1$ is an instance of the pipe template that hosts the selector $S3$ and the invocation connector $I4$, and $P2$ is an instance that hosts the selector $S1$ and the invocation connector $I3$.

The same applies to selector and invocation connectors (and indeed to any connector). A selector connector template can take any number of any connectors, and an invocation connector template can call any method on any component.

Thus we can automate the process of control flow construction for any system with an exogenous connection structure by instantiating connector templates into

instances in the latter. Indeed, we have implemented a generic container [14] for joining control and computation, which can construct, on the fly, the control flow for any exogenous connection structure expressed as an XML description. For system developers the process of system construction is reduced to the provision of components (encapsulating computation only) and a description of the system's connection structure. From these, the generic container automatically generates the run-time system. [2]

Figure 7 illustrates this using the bank example.

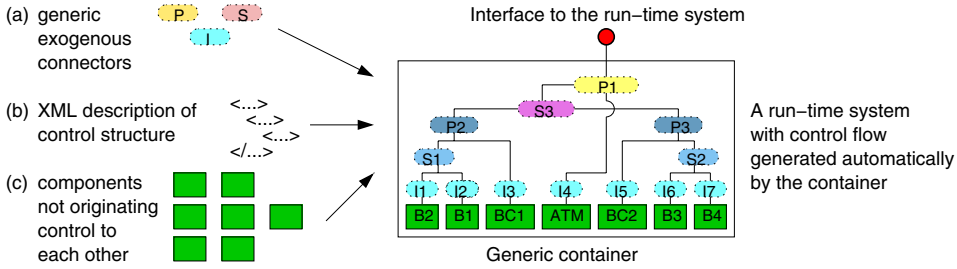


Fig. 7. Automated system construction using a container.

The top-level connector is exposed by the generic container to provide a user interface to the system. As in the classic Model View Controller pattern [6], the system can have several user interfaces to the same business logic.

Finally, in this example, we need and use only three connector types. Other systems may require more, and these can be defined and used in the same way as in this example. [3]

4 MDA-like system construction

The system construction introduced above is MDA-like. Figure 8 shows our various models at the various levels of abstractions. We start by constructing an

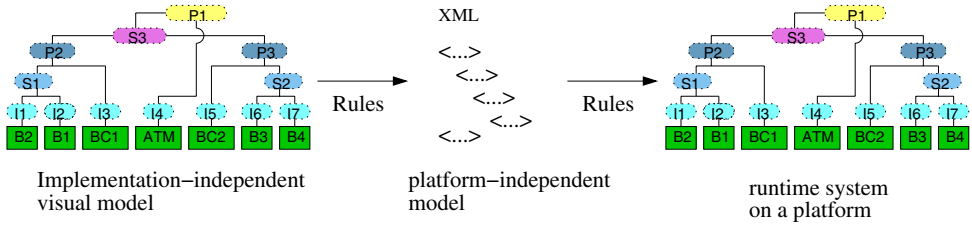


Fig. 8. MDA-like system construction

implementation-independent visual model of a system. This is similar to ADL diagrams. Subsequently, we use a set of rules to transform the visual model into an

[2] In [14] we show that our container is different from containers in existing component models like EJB (Enterprise JavaBeans) and CCM (CORBA Component Model) in that the latter only execute control flow already fixed in and between the components; they do not generate control flow automatically.

[3] We have built an Automated Train Protection System (ATP) using exogenous connectors and the generic container. In that system we could reuse the pipe, selector and invocation connector from the Bank Example. Furthermore, we introduced a sequencer connector and an ATP-specific connector. Further applications of our component model to different domains are being performed.

XML-based one. The XML-based model is not at the level of abstraction of the visual model any more as it contains more details about the system. However, it is platform-independent. Finally, we transform the platform-independent model of the system into a running system using a set of rules. The running system contains instances of components and connectors and is platform-dependent. The generic container builds up the system following the XML model. Note, that by contrast to ADLs we do not generate code from our model but use a generic container to build up and instantiate the system on the fly.

Note that since the four-level metamodel hierarchy in MDA is relative [19], we do not attempt here to put our models from Figure 8 into specific levels M0-M3 in MDA.

5 Properties of Systems

Systems built using our component model are easy to manage because they are modular and maintainable. And this, in turn, is due to separation of two concerns in these systems: control flow and computation.

5.1 Modularity

As we have seen in the bank example, the top-level connector in a system with exogenous connectors provides an interface to the system. Similarly, any connector in the system provides an interface to the subsystem of which it is the top-most connector. Thus a system with exogenous connectors is modular, and any part of the system is an independent subsystem. Such subsystems can be tested or reused separately.

Figure 9 shows two subsystems in the bank example. *Subsystem1* represents the

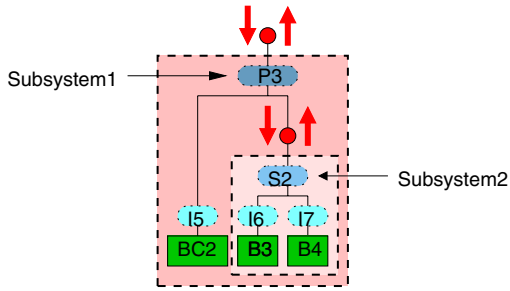


Fig. 9. Subsystems.

subsystem that takes the customer information passed on by the *ATM* component, and executes the action requested by the customer. To do this task, *Subsystem1* uses the functionality of *Subsystem2*, the subsystem that actually carries out the action.

Subsystem1 and *Subsystem2* have a specific function each that can be tested and verified independently. Moreover, the subsystems can also be reused independently of each other. As the figure shows, each subsystem has an interface which provides

an entry and exit point for control and data. All required resources, such as data accessed by the subsystem, are either contained within the subsystem or explicitly identified as input or output to the subsystem via its interface. Thus each subsystem can be reused as an independent unit.

5.2 Maintainability

As a corollary of its modularity, a system based on our component model is maintainable. Not only can a subsystem be tested and reused separately, as we have seen, but also a subsystem can be easily added to or removed from a connection structure.

Consider the scenario of adding a new subsystem to the bank example, for example a new consortium $BC3$ with banks $B5$, $B6$ and $B7$, as shown in Figure 10. In a traditional port-based architecture (e.g. Figure 10 (a)), because components are tightly coupled and connectors embedded into them, this addition will require some modifications in the code of the existing components. It is necessary not only to define the required ports in ATM and $BC3$, but also to add the code in ATM to direct the control flow in the system to $BC3$ whenever $BC3$ is the consortium that the customer's bank belongs to. By contrast, using exogenous connectors (Figure 10 (b)), existing components do not need to be modified. It is only necessary to redefine the connector $S3$, by adding a new condition and its corresponding action, i.e. if the customer's bank is in $BC3$, then execute the subsystem with $P4$ as its interface.

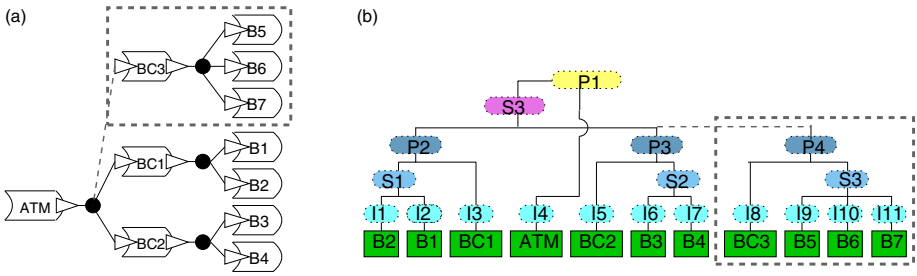


Fig. 10. Adding a new bank consortium.

With the container in Figure 7 for systems in our component model, typical maintenance tasks such as replacing, adding or removing subsystems will only involve changing the XML description of the system's connection structure.

6 Conclusion

In this paper we have briefly presented our component model and a qualitative analysis of its potential advantages over traditional ADLs.

We believe that the overall benefit of using exogenous connectors is that they separate control from computation in component-based systems. Exogenous connectors make components truly independent and therefore more reusable in different architectures, because they take control out of components totally, leaving the latter

to perform purely computation. Exogenous connectors make hierarchical system design possible, due to their own strictly hierarchical nature. They also make system construction easier by enabling automated control flow generation from a system's architecture. Systems based on our component model are easier to manage because they are modular and maintainable. All these advantages mean that using our component model should result in not only reduced time to market, but also reduced software production and maintenance costs.

We think that our component model can find its specific place in the Model-driven architecture as systems in our component model, systems are constructed by model transformation beginning with a visual model through the XML model towards the runtime system. The runtime system is constructed following a 'construction plan', which is the XML description of exogenous connection of the system, used by the generic container. The system description is platform-independent, which is one of the key properties in MDA.

However, in this paper we have not presented a quantitative analysis. Our work on the component model is only beginning, and we do not have any substantial experimental data to report yet. Nevertheless, we firmly believe that our component model holds great promise, not only because of the aforementioned advantages but also because of its potential to provide a unique bridge between traditional ADLs and component-based software development. The former is top-down, has a well-developed theory, but has not proved very practical; the latter is bottom-up, has no firm theoretical foundations as yet, but has a lot of practical support by way of tools and middleware. Constructing an architecture by putting an exogenous connection structure on top of pre-existing components mixes software architecture with component-based software development in a mutually beneficial manner. Thus, our component model has the potential to combine the best of both worlds, and as future work, we plan to gather quantitative information on the performance of exogenous connectors in practical component-based software development.

In terms of technical work, we also need to extend our component model to concurrency, as well as layered architectures. Furthermore, we are working on Deployment Contracts for software components, which are metadata [13] about components' runtime behavior.

The work on coordination contracts presented in [9] suggests an approach to facilitate evolution of software systems. The idea is to coordinate classes by using a special language for expressing coordination rules among them. Code for coordination contracts along with coordinated classes is compiled together to yield the complete code for the system. In our approach we operate on binaries. That is, the generic container takes components as well as connectors as binaries. It then puts them together on the fly resulting in their increased reuse potential. Code reuse is not preferable as the generated code for the system has to be maintained just for that system thus complicating system's evolution. Our connectors are reusable entities whereas coordination contracts from [9] are not intended to be reused. The coordination contracts approach does not tackle software architecture issues.

The idea of hyperspaces introduced in [11] aims at identifying slices in the pro-

gram relating to a concern. Once the hyperslices are identified, weaving is done. In this approach, again, after the code for the system has been woven, the system cannot be changed without changing the code. Our approach allows system changes to be performed by only changing the XML description of the control structure in the system thus offering more flexibility in system maintenance. Furthermore, we can reuse our components and connectors. By contrast, the idea of hyperslices does not promote reuse.

In terms of dynamic composition and reconfiguration, our approach seems to hold great potential. Using the generic container for constructing the system on the fly gives the opportunity to govern dynamic architectural system changes as well as reconfigurations by the container as well. In other words, the generic container container can be extended to take an XML description of additional connectors and components and connect them to a running system on the fly. Such changes are however difficult to perform in a stateful system.

The number of connectors in a system constructed using exogenous connectors might be bigger than in a system built using direct or indirect method calls. This has its nature in the flexibility offered by our architectures. Although, it can be argued that the generic container takes charge of the composition releaving the system developer from manual composition, the footprint of the system gets larger with the increased number of connectors involved.

References

[1] F. Arbab. The IWIM model for coordination of concurrent activities. In P. Ciancarini and C. Hankin, editors, *Lecture Notes in Computer Science 1061*, pages 34–56. Springer-Verlag, 1996.

[2] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, and J. Zettel. *Component-based Product Line Engineering with UML*. Addison-Wesley, 2001.

[3] D. Box. *Essential COM*. Addison-Wesley, 1998.

[4] E. Bruneton, T. Coupaye, and J.B. Stefani. The Fractal component model. Technical Report Draft-Version 2.0-3, The ObjectWeb Consortium, 2004.

[5] R. Englander. *Developing Java Beans*. O'Reilly & Associates, 1997.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissades. *Design Patterns – Elements of Reusable Object-Oriented Design*. Addison-Wesley, 1994.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. The courier pattern. *Dr. Dobb's Journal*, Feburary 1996.

[8] D. Garlan, R.T. Monroe, and D. Wile. Acme: Architectural description of component-based systems. In G.T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[9] Joao Gouveia, Georgios Koutsoukos, Michel Wermelinger, Lus Andrade, and Jose Luiz Fiadeiro. Developing and evolving java applications using coordination contracts. In *OOPSLA '02: Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 28–29, New York, NY, USA, 2002. ACM Press.

[10] R.M. Haefel. *Enterprise JavaBeans*. O'Reilly & Associates, 3rd edition, 2001.

[11] William Harrison, Harold Ossher, Stanley Sutton, and Peri Tarr. Concern modeling in the concern manipulation environment. In *MACS '05: Proceedings of the 2005 workshop on Modeling and analysis of concerns in software*, pages 1–5, New York, NY, USA, 2005. ACM Press.

[12] J. Ivers, N. Sinha, and K.C Wallnau. A Basis for Composition Language CL. Technical Report CMU/SEI-2002-TN-026, CMU SEI, 2002.

[13] K.-K. Lau and V. Ukis. Component metadata in component-based software development: A survey. Preprint CSPP-34, School of Computer Science, The University of Manchester, October 2005.

[14] K.-K. Lau and V. Ukis. A container for automatic system control flow generation using exogenous connectors. Preprint CSPP-31, School of Computer Science, The University of Manchester, August 2005.

[15] K.-K. Lau, P. Velasco Elizondo, and Z. Wang. Exogenous connectors for software components. In *Proc. 8th Int. SIGSOFT Symp. on Component-based Software Engineering, LNCS 3489*, pages 90–106, 2005.

[16] K.-K. Lau and Z. Wang. A survey of software component models. Pre-print CSPP-30, School of Computer Science, The University of Manchester, April 2005.

[17] K.-K. Lau and Z. Wang. A taxonomy of software component models. In *Proc. 31st Euromicro Conference*. IEEE Computer Society Press, 2005.

[18] N.R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *Proc. 22nd International Conference on Software Engineering*, pages 178–187. ACM Press, 2000.

[19] Stephen J. Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. *MDA Distilled*. Addison-Wesley, March 2004.

[20] O. Nierstrasz, G. Arévalo, S. Ducasse, R. Wuyts, A. Black, P. Müller, C. Zeidler, T. Genssler, and R. van den Born. A component model for field devices. In *Proc. 1st Int. IFIP/ACM Working Conference on Component Deployment*, pages 200–209. ACM Press, 2002.

[21] OMG. *UML 2.0 Superstructure Specification*. http://www.omg.org/cgi-bin/doc?ptc/2003-08-02.

[22] OMG. *CORBA Component Model, V3.0*, 2002. http://www.omg.org/technology/documents/formal/components.htm.

[23] F. Plasil, D. Balek, and R. Janecek. SOFA/DCUP: Architecture for component trading and dynamic updating. In *Proc. ICCDS98*, pages 43–52. IEEE Press, 1998.

[24] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[25] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 2002.

[26] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A component- and message-based architectural style for GUI software. *Software Engineering*, 22(6):390–406, 1996.

[27] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *IEEE Computer*, pages 78–85, March 2000.