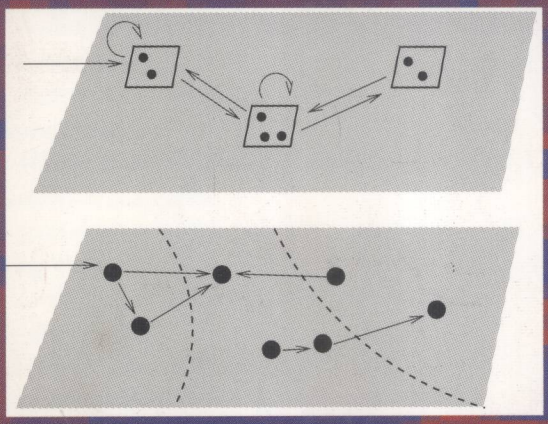
State-of-the-Art Survey

LNCS 4111

Frank S. de Boer
Marcello M. Bonsangue
Susanne Graf
Willem-Paul de Roever (Eds.)

Formal Methods for Components and Objects

**4th International Symposium, FMCO 2005
Amsterdam, The Netherlands, November 2005
Revised Lectures**



Springer

Frank S. de Boer Marcello M. Bonsangue
Susanne Graf Willem-Paul de Roever (Eds.)

Formal Methods for Components and Objects

4th International Symposium, FMCO 2005
Amsterdam, The Netherlands, November 1-4, 2005
Revised Lectures

Springer

Volume Editors

Frank S. de Boer
Centre for Mathematics and Computer Science, CWI
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
E-mail: F.S.de.Boer@cwi.nl

Marcello M. Bonsangue
Leiden University
Leiden Instiute of Advanced Computer Science
P. O. Box 9512, 2300 RA Leiden, The Netherlands
E-mail: marcello@liacs.nl

Susanne Graf
VERIMAG
2 Avenue de Vignate, Centre Equitation, 38610 Grenoble-Giéres, France
E-mail: Susanne.Graf@imag.fr

Willem-Paul de Roever
University of Kiel
Institute of Computer Science and Applied Mathematics
Hermann-Rodewald-Str. 3, 24118 Kiel, Germany
E-mail: wpr@informatik.uni-kiel.de

Library of Congress Control Number: 2006930291

CR Subject Classification (1998): D.2, D.3, F.3, D.4

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-36749-7 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-36749-9 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11804192 06/3142 5 4 3 2 1 0

Lecture Notes in Computer Science 4111

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

Preface

Large and complex software systems provide the necessary infrastructure in all industries today. In order to construct such large systems in a systematic manner, the focus in the development methodologies has switched in the last two decades from functional issues to structural issues: both data and functions are encapsulated into software units which are integrated into large systems by means of various techniques supporting reusability and modifiability. This encapsulation principle is essential to both the object-oriented and the more recent component-based software engineering paradigms.

Formal methods have been applied successfully to the verification of medium-sized programs in protocol and hardware design. However, their application to the development of large systems requires more emphasis on specification, modeling and validation techniques supporting the concepts of reusability and modifiability and their implementation in new extensions of existing programming languages like Java.

The new format of FMCO 2005 consisted of invited keynote lectures and tutorial lectures selected through a corresponding open call. The latter provide a tutorial perspective on recent developments. In contrast to existing conferences, about half of the program consisted of invited keynote lectures by top researchers sharing their interest in the application or development of formal methods for large-scale software systems (object or component oriented). FMCO does not focus on specific aspects of the use of formal methods, but rather it aims at a systematic and comprehensive account of the expanding body of knowledge on modern software systems.

This volume contains the contributions submitted after the symposium by both invited and selected lecturers. The proceedings of FMCO 2002, FMCO 2003, and FMCO 2004 have already been published as volumes 2852, 3188, and 3657 of Springer's *Lecture Notes in Computer Science*. We believe that these proceedings provide a unique combination of ideas on software engineering and formal methods which reflect the expanding body of knowledge on modern software systems.

June 2006

F.S. de Boer
M.M. Bonsangue
S. Graf
W.-P. de Roever

Organization

The FMCO symposia are organized in the context of the project Mobi-J, a project founded by a bilateral research program of The Dutch Organization for Scientific Research (NWO) and the Central Public Funding Organization for Academic Research in Germany (DFG). The partners of the Mobi-J projects are: the Centrum voor Wiskunde en Informatica, the Leiden Institute of Advanced Computer Science, and the Christian-Albrechts-Universität Kiel.

This project aims at the development of a programming environment which supports component-based design and verification of Java programs annotated with assertions. The overall approach is based on an extension of the Java language with a notion of component that provides for the encapsulation of its internal processing of data and composition in a network by means of mobile asynchronous channels.

Sponsoring Institutions

The Dutch Organization for Scientific Research (NWO)
The Royal Netherlands Academy of Arts and Sciences (KNAW)
The Dutch Institute for Programming research and Algorithmics (IPA)
The Centrum voor Wiskunde en Informatica (CWI), The Netherlands
The Leiden Institute of Advanced Computer Science (LIACS), The Netherlands

Lecture Notes in Computer Science

For information about Vols. 1–4011

please contact your bookseller or Springer

Vol. 4127: E. Damiani, P. Liu (Eds.), Data and Applications Security XX. X, 319 pages. 2006.

Vol. 4121: A. Biere, C.P. Gomes (Eds.), Theory and Applications of Satisfiability Testing - SAT 2006. XII, 438 pages. 2006.

Vol. 4112: D.Z. Chen, D. T. Lee (Eds.), Computing and Combinatorics. XIV, 528 pages. 2006.

Vol. 4111: F.S. de Boer, M.M. Bonsangue, S. Graf, W.-P. de Roever (Eds.), Formal Methods for Components and Objects. VIII, 429 pages. 2006.

Vol. 4108: J.M. Borwein, W.M. Farmer (Eds.), Mathematical Knowledge Management. VIII, 295 pages. 2006. (Sublibrary LNAI).

Vol. 4106: T.R. Roth-Berghofer, M.H. Göker, H. A. Güvenir (Eds.), Advances in Case-Based Reasoning. XIV, 566 pages. 2006. (Sublibrary LNAI).

Vol. 4104: T. Kunz, S.S. Ravi (Eds.), Ad-Hoc, Mobile, and Wireless Networks. XII, 474 pages. 2006.

Vol. 4099: Q. Yang, G. Webb (Eds.), PRICAI 2006: Trends in Artificial Intelligence. XXVIII, 1263 pages. 2006. (Sublibrary LNAI).

Vol. 4098: F. Pfenning (Ed.), Term Rewriting and Applications. XIII, 415 pages. 2006.

Vol. 4097: X. Zhou, O. Sokolsky, L. Yan, E.-S. Jung, Z. Shao, Y. Mu, D.C. Lee, D. Kim, Y.-S. Jeong, C.-Z. Xu (Eds.), Emerging Directions in Embedded and Ubiquitous Computing. XXVII, 1034 pages. 2006.

Vol. 4096: E. Sha, S.-K. Han, C.-Z. Xu, M.H. Kim, L.T. Yang, B. Xiao (Eds.), Embedded and Ubiquitous Computing. XXIV, 1170 pages. 2006.

Vol. 4094: O. H. Ibarra, H.-C. Yen (Eds.), Implementation and Application of Automata. XIII, 291 pages. 2006.

Vol. 4093: X. Li, O.R. Zaiane, Z. Li (Eds.), Advanced Data Mining and Applications. XXI, 1110 pages. 2006. (Sublibrary LNAI).

Vol. 4092: J. Lang, F. Lin, J. Wang (Eds.), Knowledge Science, Engineering and Management. XV, 664 pages. 2006. (Sublibrary LNAI).

Vol. 4090: S. Spaccapietra, K. Aberer, P. Cudré-Mauroux (Eds.), Journal on Data Semantics VI. XI, 211 pages. 2006.

Vol. 4088: Z.-Z. Shi, R. Sadananda (Eds.), Agent Computing and Multi-Agent Systems. XVII, 827 pages. 2006. (Sublibrary LNAI).

Vol. 4079: S. Etalle, M. Truszczyński (Eds.), Logic Programming. XIV, 474 pages. 2006.

Vol. 4077: M.-S. Kim, K. Shimada (Eds.), Advances in Geometric Modeling and Processing. XVI, 696 pages. 2006.

Vol. 4076: F. Hess, S. Pauli, M. Pohst (Eds.), Algorithmic Number Theory. X, 599 pages. 2006.

Vol. 4075: U. Leser, F. Naumann, B. Eckman (Eds.), Data Integration in the Life Sciences. XI, 298 pages. 2006. (Sublibrary LNBI).

Vol. 4074: M. Burmester, A. Yasinsac (Eds.), Secure Mobile Ad-hoc Networks and Sensors. X, 193 pages. 2006.

Vol. 4073: A. Butz, B. Fisher, A. Krüger, P. Olivier (Eds.), Smart Graphics. XI, 263 pages. 2006.

Vol. 4072: M. Harders, G. Székely (Eds.), Biomedical Simulation. XI, 216 pages. 2006.

Vol. 4071: H. Sundaram, M. Naphade, J.R. Smith, Y. Rui (Eds.), Image and Video Retrieval. XII, 547 pages. 2006.

Vol. 4070: C. Priami, X. Hu, Y. Pan, T.Y. Lin (Eds.), Transactions on Computational Systems Biology V. IX, 129 pages. 2006. (Sublibrary LNBI).

Vol. 4069: F.J. Perales, R.B. Fisher (Eds.), Articulated Motion and Deformable Objects. XV, 526 pages. 2006.

Vol. 4068: H. Schärfe, P. Hitzler, P. Øhrstrøm (Eds.), Conceptual Structures: Inspiration and Application. XI, 455 pages. 2006. (Sublibrary LNAI).

Vol. 4067: D. Thomas (Ed.), ECOOP 2006 – Object-Oriented Programming. XIV, 527 pages. 2006.

Vol. 4066: A. Rensink, J. Warmer (Eds.), Model Driven Architecture – Foundations and Applications. XII, 392 pages. 2006.

Vol. 4065: P. Perner (Ed.), Advances in Data Mining. XI, 592 pages. 2006. (Sublibrary LNAI).

Vol. 4064: R. Büschkes, P. Laskov (Eds.), Detection of Intrusions and Malware & Vulnerability Assessment. X, 195 pages. 2006.

Vol. 4063: I. Gorton, G.T. Heineman, I. Crnkovic, H.W. Schmidt, J.A. Stafford, C.A. Szyperski, K. Wallnau (Eds.), Component-Based Software Engineering. XI, 394 pages. 2006.

Vol. 4062: G. Wang, J.F. Peters, A. Skowron, Y. Yao (Eds.), Rough Sets and Knowledge Technology. XX, 810 pages. 2006. (Sublibrary LNAI).

Vol. 4061: K. Miesenberger, J. Klaus, W. Zagler, A. Karshmer (Eds.), Computers Helping People with Special Needs. XXIX, 1356 pages. 2006.

Vol. 4060: K. Futatsugi, J.-P. Jouannaud, J. Meseguer (Eds.), Algebra, Meaning and Computation. XXXVIII, 643 pages. 2006.

Vol. 4059: L. Arge, R. Freivalds (Eds.), Algorithm Theory – SWAT 2006. XII, 436 pages. 2006.

Vol. 4058: L.M. Batten, R. Safavi-Naini (Eds.), Information Security and Privacy. XII, 446 pages. 2006.

Vol. 4057: J.P. W. Pluim, B. Likar, F.A. Gerritsen (Eds.), Biomedical Image Registration. XII, 324 pages. 2006.

Vol. 4056: P. Flocchini, L. Gąsieniec (Eds.), Structural Information and Communication Complexity. X, 357 pages. 2006.

Vol. 4055: J. Lee, J. Shim, S.-g. Lee, C. Bussler, S. Shim (Eds.), Data Engineering Issues in E-Commerce and Services. IX, 290 pages. 2006.

Vol. 4054: A. Horváth, M. Telek (Eds.), Formal Methods and Stochastic Models for Performance Evaluation. VIII, 239 pages. 2006.

Vol. 4053: M. Ikeda, K.D. Ashley, T.-W. Chan (Eds.), Intelligent Tutoring Systems. XXVI, 821 pages. 2006.

Vol. 4052: M. Bugliesi, B. Preneel, V. Sassone, I. Wegener (Eds.), Automata, Languages and Programming, Part II. XXIV, 603 pages. 2006.

Vol. 4051: M. Bugliesi, B. Preneel, V. Sassone, I. Wegener (Eds.), Automata, Languages and Programming, Part I. XXIII, 729 pages. 2006.

Vol. 4049: S. Parsons, N. Maudet, P. Moraitis, I. Rahwan (Eds.), Argumentation in Multi-Agent Systems. XIV, 313 pages. 2006. (Sublibrary LNAI).

Vol. 4048: L. Goble, J.-J.C.. Meyer (Eds.), Deontic Logic and Artificial Normative Systems. X, 273 pages. 2006. (Sublibrary LNAI).

Vol. 4047: M. Robshaw (Ed.), Fast Software Encryption. XI, 434 pages. 2006.

Vol. 4046: S.M. Astley, M. Brady, C. Rose, R. Zwiggelaar (Eds.), Digital Mammography. XVI, 654 pages. 2006.

Vol. 4045: D. Barker-Plummer, R. Cox, N. Swoboda (Eds.), Diagrammatic Representation and Inference. XII, 301 pages. 2006. (Sublibrary LNAI).

Vol. 4044: P. Abrahamsson, M. Marchesi, G. Succi (Eds.), Extreme Programming and Agile Processes in Software Engineering. XII, 230 pages. 2006.

Vol. 4043: A.S. Atzeni, A. Lioy (Eds.), Public Key Infrastructure. XI, 261 pages. 2006.

Vol. 4042: D. Bell, J. Hong (Eds.), Flexible and Efficient Information Handling. XVI, 296 pages. 2006.

Vol. 4041: S.-W. Cheng, C.K. Poon (Eds.), Algorithmic Aspects in Information and Management. XI, 395 pages. 2006.

Vol. 4040: R. Reulke, U. Eckardt, B. Flach, U. Knauer, K. Polthier (Eds.), Combinatorial Image Analysis. XII, 482 pages. 2006.

Vol. 4039: M. Morisio (Ed.), Reuse of Off-the-Shelf Components. XIII, 444 pages. 2006.

Vol. 4038: P. Ciancarini, H. Wiklicky (Eds.), Coordination Models and Languages. VIII, 299 pages. 2006.

Vol. 4037: R. Gorrieri, H. Wehrheim (Eds.), Formal Methods for Open Object-Based Distributed Systems. XVII, 474 pages. 2006.

Vol. 4036: O. H. Ibarra, Z. Dang (Eds.), Developments in Language Theory. XII, 456 pages. 2006.

Vol. 4035: T. Nishita, Q. Peng, H.-P. Seidel (Eds.), Advances in Computer Graphics. XX, 771 pages. 2006.

Vol. 4034: J. Münch, M. Vierimaa (Eds.), Product-Focused Software Process Improvement. XVII, 474 pages. 2006.

Vol. 4033: B. Stiller, P. Reichl, B. Tuffin (Eds.), Performability Has its Price. X, 103 pages. 2006.

Vol. 4032: O. Etzion, T. Kuflik, A. Motro (Eds.), Next Generation Information Technologies and Systems. XIII, 365 pages. 2006.

Vol. 4031: M. Ali, R. Dapoigny (Eds.), Advances in Applied Artificial Intelligence. XXIII, 1353 pages. 2006. (Sublibrary LNAI).

Vol. 4029: L. Rutkowski, R. Tadeusiewicz, L.A. Zadeh, J. Zurada (Eds.), Artificial Intelligence and Soft Computing – ICAISC 2006. XXI, 1235 pages. 2006. (Sublibrary LNAI).

Vol. 4028: J. Kohlas, B. Meyer, A. Schiper (Eds.), Dependable Systems: Software, Computing, Networks. XII, 295 pages. 2006.

Vol. 4027: H.L. Larsen, G. Pasi, D. Ortiz-Arroyo, T. Andreasen, H. Christiansen (Eds.), Flexible Query Answering Systems. XVIII, 714 pages. 2006. (Sublibrary LNAI).

Vol. 4026: P.B. Gibbons, T. Abdelzaher, J. Aspnes, R. Rao (Eds.), Distributed Computing in Sensor Systems. XIV, 566 pages. 2006.

Vol. 4025: F. Eliassen, A. Montresor (Eds.), Distributed Applications and Interoperable Systems. XI, 355 pages. 2006.

Vol. 4024: S. Donatelli, P. S. Thiagarajan (Eds.), Petri Nets and Other Models of Concurrency - ICATPN 2006. XI, 441 pages. 2006.

Vol. 4021: E. André, L. Dybkjær, W. Minker, H. Neumann, M. Weber (Eds.), Perception and Interactive Technologies. XI, 217 pages. 2006. (Sublibrary LNAI).

Vol. 4020: A. Bredenfeld, A. Jacoff, I. Noda, Y. Takahashi (Eds.), RoboCup 2005: Robot Soccer World Cup IX. XVII, 727 pages. 2006. (Sublibrary LNAI).

Vol. 4019: M. Johnson, V. Vene (Eds.), Algebraic Methodology and Software Technology. XI, 389 pages. 2006.

Vol. 4018: V. Wade, H. Ashman, B. Smyth (Eds.), Adaptive Hypermedia and Adaptive Web-Based Systems. XVI, 474 pages. 2006.

Vol. 4017: S. Vassiliadis, S. Wong, T.D. Hämäläinen (Eds.), Embedded Computer Systems: Architectures, Modeling, and Simulation. XV, 492 pages. 2006.

Vol. 4016: J.X. Yu, M. Kitsuregawa, H.V. Leong (Eds.), Advances in Web-Age Information Management. XVII, 606 pages. 2006.

Vol. 4014: T. Uustalu (Ed.), Mathematics of Program Construction. X, 455 pages. 2006.

Vol. 4013: L. Lamontagne, M. Marchand (Eds.), Advances in Artificial Intelligence. XIII, 564 pages. 2006. (Sublibrary LNAI).

Vol. 4012: T. Washio, A. Sakurai, K. Nakajima, H. Takeda, S. Tojo, M. Yokoo (Eds.), New Frontiers in Artificial Intelligence. XIII, 484 pages. 2006. (Sublibrary LNAI).

¥479.00元

Table of Contents

Component and Service Oriented Computing

System Design

Tools

Algebraic Methods

Model Checking

Assertional Methods

Quantitative Analysis

A Software Component Model and Its Preliminary Formalisation

Kung-Kiu Lau[1], Mario Ornaghi[2], and Zheng Wang[1]

[1] School of Computer Science, the University of Manchester
Manchester M13 9PL, United Kingdom
{kung-kiu, zw}@cs.man.ac.uk
[2] Dipartimento di Scienze dell'Informazione,
Universita' degli studi di Milano
Via Comelico 39/41, 20135 Milano, Italy
ornaghi@dsi.unimi.it

Abstract. A software component model should define what components are, and how they can be composed. That is, it should define a theory of components and their composition. Current software component models tend to use objects or port-connector type architectural units as components, with method calls and port-to-port connections as composition mechanisms. However, these models do not provide a proper composition theory, in particular for key underlying concepts such as encapsulation and compositionality. In this paper, we outline our notion of these concepts, and give a preliminary formalisation of a software component model that embodies these concepts.

1 Introduction

The context of this work is Component-based Software Engineering, rather than Component-based Systems. In the latter, the focus is on system properties, and components are typically state machines. Key concerns are issues related to communication, concurrency, processes, protocols, etc. Properties of interest are temporal, non-functional properties such as deadlock-freedom, safety, liveness, etc. In the former, the focus is on software components and middleware for composing them. Usually a software component model, e.g. Enterprise JavaBeans (EJB) [21], provides the underlying framework.

A software component model should define (i) what components are, i.e. their syntax and semantics; and (ii) how to compose components, i.e. the semantics of their composition. Current component models tend to use objects or port-connector type architectural units as components, with method calls and port-to-port connections as composition mechanisms. However, these models do not define a proper theory for composition.

We believe that encapsulation and compositionality are key concepts for such a theory. In this paper, we explain these notions, and their role in a composition theory. Using these concepts, we present a software component model, together with a preliminary formalisation.

F.S. de Boer et al. (Eds.): FMCO 2005, LNCS 4111, pp. 1–21, 2006.

2 Current Component Models

Currently, so-called component models, e.g. EJB and CCM (CORBA Component Model) [24], do not follow a standard terminology or semantics. There are different definitions of what a component is [6], and most of these are not set in the context of a component model. In particular, they do not define composition properly.

For example, a widely used definition of components is the following, due to Szyperski [28]:

> "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

A different definition is the following by Meyer [20]:

> "A component is a software element (modular unit) satisfying the following conditions:
> 1. It can be used by other software elements, its 'clients'.
> 2. It possesses an official usage description, which is sufficient for a client author to use it.
> 3. It is not tied to any fixed set of clients."

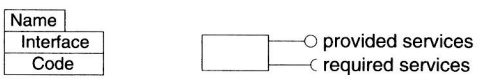
Both these definitions do not mention a component model, in particular how composition is defined.

The following definition given in Heineman and Councill [12] mentions a component model:

> "A [component is a] software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard."

but it does not define one.

Nevertheless, there is a commonly accepted abstract view of what a component is, viz. a software unit that contains (i) code for performing services, and (ii) an interface for accessing these services (Fig. 1(a)). To provide its services, a component



(a) (b)

Fig. 1. A software component

may require some services. So a component is often depicted as in Fig. 1(b), e.g. in CCM and UML2.0 [23].

In current software component models, components are typically objects as in object-oriented languages, and port-connector type architectural units, with method calls and ADL (architecture description languages [26]) connectors as composition mechanisms respectively.

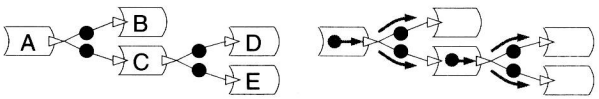
A complete survey of these models is beyond the scope of this paper. It can be found in [17].

3 Our Component Model

In our component model, *components* encapsulate *computation* (and data),[1] and *composition operators* encapsulate *control*. Our components are constructed from (i) computation units and (ii) connectors. A computation unit performs computation within itself, and does not invoke computation in another unit. Connectors are used to build components from computation units, and also as composition operators to compose components into composite components.

3.1 Exogenous Connectors

Our connectors are *exogenous connectors* [16]. The distinguishing characteristic of exogenous connectors is that they encapsulate control. In traditional ADLs, components



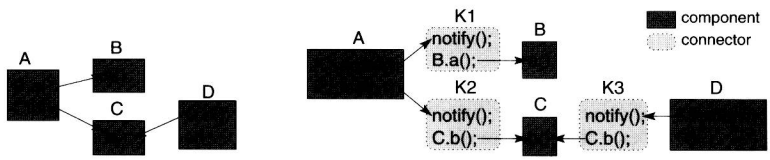
(a) Components and connectors (b) Control flow

Fig. 2. Traditional ADLs

are supposed to represent *computation*, and connectors *interaction* between components [19] (Fig. 2 (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 Fig. 2 (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. Direct message



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

Fig. 3. Connection by message passing

passing corresponds to direct method calls, as exemplified by objects calling methods in other objects (Fig. 3 (a)), using method or event delegation, or remote procedure call (RPC). Software component models that adopt direct message passing schemes as

[1] For lack of space, we will not discuss data in this paper.

composition operators are EJB, CCM, COM [5], UML2.0 [23] and KobrA [3]. 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 we use a connector that when notified by the former invokes a method in the latter (Fig. 3 (b)). Besides ADLs, other software component models that adopt indirect message passing schemes are JavaBeans [27], Koala [30], SOFA [25], PECOS [22], PIN [14] and Fractal [8].

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

A categorical semantics of connectors is proposed in [9], where coordination is modelled through signature morphisms. There is a clear separation between computation, occurring in components, and coordination, performed by connectors. However, shared actions may propagate control from one component to others.

By contrast, in exogenous connection, control originates in and flows from connectors, leaving components to encapsulate only computation. This is illustrated by Fig. 4.

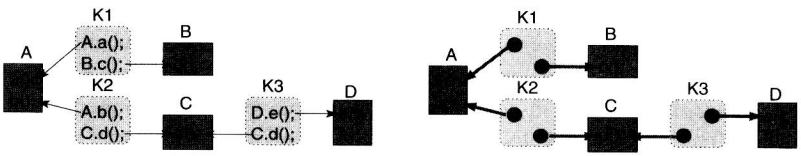


Fig. 4. Connection by exogenous connectors

In Fig. 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 Fig. 4 (b), in clear contrast to Fig. 2 (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.

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.

3.2 Components

Our view of a component is that it is not simply a part of the whole system. Rather it is something very different from traditional software units such as code fragments,

functions, procedures, subroutines, modules, classes/objects, programs, packages, etc, and equally different from more modern units like DLLs and services.

We define a component as follows:

Definition 1. A *software component* is a software unit with the following defining characteristics: (i) encapsulation and (ii) compositionality.

A component should encapsulate both *data* and *computation*. A component C encapsulates data by making its data private. C encapsulates computation by making sure that its computation happens entirely within itself.

An object can encapsulate data, but it does not encapsulate computation, since objects can call methods in other objects (Fig. 5(a)).

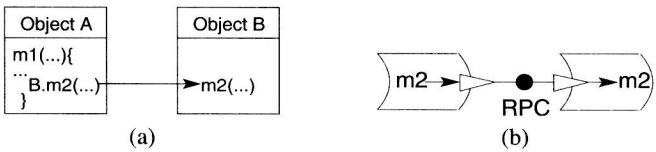


Fig. 5. Objects and architectural units

Port-connector type components, as in e.g. ADLs, UML2.0 and Koala, can encapsulate data. However, they usually do not encapsulate computation, since components can call methods in other components by remote procedure call (RPC), albeit only indirectly via connectors (and ports) (Fig. 5(b)).

Components should be *compositional*, i.e. the composition of two components C_1 and C_2 should yield another component C_3, which in turn should also have the defining characteristics of encapsulation and compositionality. Thus compositionality implies that composition preserves or propagates encapsulation.[2]

Classes and objects are not compositional. They can only be 'composed' by method calls, and such a 'composition' does not yield another class or object. Indeed, method calls break encapsulation. Port-connector type components can be composed, but they are not compositional if they do not have (computation) encapsulation.

Encapsulation entails that access to components must be provided by *interfaces*. Classes and objects do not have interfaces. Access to (the methods of) objects, if permitted, is direct, not via interfaces. So-called 'interfaces' in object-oriented languages like Java are themselves classes or objects, so are not interfaces to components. Port-connector type components use their ports as their interfaces.

Our components are constructed from *computation units* and *exogenous connectors*. A computation unit performs just computation within itself and does not invoke computation in another unit. It can be thought of as a class or object with methods, except that these methods do not call methods in other units. Thus it encapsulates computation.

Exogenous connectors encapsulate control, as we have seen in the previous section. The type hierarchy of these connectors in our component model is as follows. At the

[2] Compositionality in terms of other (non-functional) properties of sub-components is an open issue, which we do not address here.

lowest level, level 1, because components are not allowed to call methods in other components, we need an exogenous *invocation connector*. This is a *unary* operator that takes a computation unit, invokes one of its methods, and receives the result of the invocation. At the next level of the type hierarchy, 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 at level 2, we need *n-ary* connectors for connecting invocation connectors, and at level 3, we need *n-ary* connectors for connecting these connectors, and so on. In other words, we need a hierarchy of connectors of different arities and types. We have defined and implemented such a hierarchy in [16]. Apart from invocation connectors at level 1, our hierarchy includes *pipe* connectors, for sequencing, and *selector* connectors, for branching, at levels $n \geq 2$. These connectors are called *composition connectors* for the obvious reason. Level-1 connectors are invocation connectors, and level-2 composition connectors connect only invocation connectors, but composition connectors at higher levels are polymorphic since they can connect different kinds of connectors at different levels (and with different arities).

We distinguish between (i) *atomic* components and (ii) *composite* components.

Definition 2. An *atomic component* C is a pair $\langle i, u \rangle$ where u is a computation unit, and i is an invocation connector that invokes u's methods. i provides an interface to the component C.

A *composite component* CC is a tuple $\langle k, C_1, C_2, \ldots C_j \rangle$, for some j, where k is a j-ary connector at level $n \geq 2$, and each $C_i, i = 1, \ldots, j$, is either an atomic component or a composite component. k is called a *composition connector*. It provides an interface to the component CC.

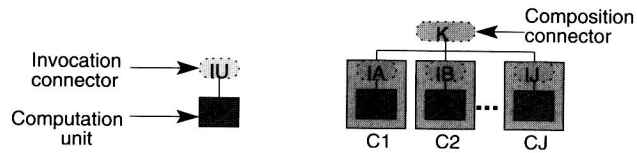


Fig. 6. Atomic and composite components

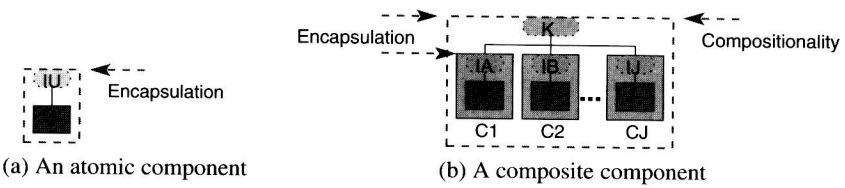


Fig. 7. Encapsulation and compositionality

Figure 6 illustrates atomic and composite components. Clearly, an atomic component encapsulates computation, since a computation unit does so, and an invocation connector invokes only methods in the unit (Fig 7(a)). It is easy to see that a composite component also encapsulates computation (Fig 7(b)).

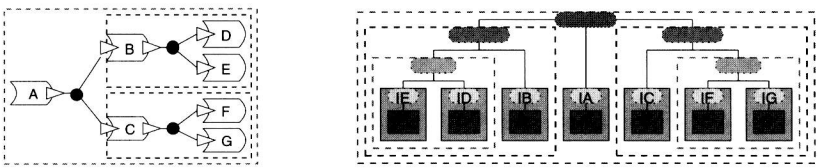
3.3 Composition Operators

To construct systems or composite components, we need composition operators that preserve encapsulation and compositionality. Such composition operators should work only on interfaces, in view of encapsulation.

Glue code is certainly not suitable as composition operators. Neither are object method calls or ADL connectors, as used in current component models. Indeed, these models do not have proper composition operators, in our view, since they do not have the concepts of encapsulation and compositionality.

As in Definition 2, we use exogenous connectors at level $n \geq 2$ as composition operators. These operators are compositional and therefor preserve and propagate encapsulation. As shown in Fig 7(b), a composite component has encapsulation, as a result of encapsulation in its constituent components. Furthermore, the composite component is also compositional. Thus, any component, be it atomic or composite, has a top-most connector that provides an interface, and it can be composed with any other component using a suitable composition operator.

This self-similarity of a composite component is a direct consequence of component encapsulation and compositionality, and provides the basis for a compositional way of constructing systems from components. Fig. 8(b) illustrates self-similarity of a com-



(a) Acme (b) Exogenous connection

Fig. 8. Self-similarity of a composite component

posite component in a system composed from atomic and composite components. Each dotted box indicates a composite component. Note in particular that the composite at the top level is the entire system itself. Most importantly, every composite component is similar to all its sub-components.

The system in Fig. 8(b) corresponds to the Acme [11] architecture in Fig. 8(a). By comparison, in the Acme system, the composites are different from those in Fig. 8(b). For instance, (D,E) is a composite in (b) but not in (a). Also, in (a) the top-level composite is not similar to the composite (B,D,E) at the next level down. The latter has an interface, but the former does not.

In general, self-similarity provides a compositional approach to system construction, and this is an advance over hierarchical approaches such as ADLs which are not compositional, strictly speaking.

3.4 The Bank Example

Having defined our component model, we illustrate its use in the construction of a simple bank system. The bank system has just one ATM that serves two banks (Bank1 and Bank2) as shown in Fig. 9.