Automated Deployment of Hierarchical Components

Tomas Kučera, Petr Hnětynka, and Jan Kofroň

Abstract. Deployment of distributed component-based systems is quite important stage in the system's life-cycle since it may significantly influence its overall performance and utilization of computers and the network. Thus, deployment of the system has to be carefully planned. There exist algorithms for deployment of component-based system; however they allow deployment of systems with a single level of component composition; hierarchical systems have to be flattened before deployment. However, such a flattening is not possible for component frameworks where hierarchical components exist also at run-time. In this paper, we present an algorithm for automated deployment planning of hierarchical component systems. The algorithm incorporates component demands and machine resources in order to maximize performance of deployed applications. We also present an implementation of the algorithm for the SOFA 2 component framework.

1 Introduction

Component-based development (CBD) [15] is currently well understood and widely used technique for development of software systems in all kinds of domains ranging from embedded to enterprise ones. Using it, systems are built by composition of well-defined reusable pieces of software blocks, i.e. components. There exist a number of component frameworks, which differ in understanding what a component is, how they can be composed, deployed, etc. Each of the frameworks defines its *component model*, which is a definition of components and all related abstractions. Nevertheless, a common consensus is that a component is a black-box entity

Tomas Kučera · Petr Hnětynka · Jan Kofroň
Charles University, Faculty of Mathematics and Physics
Department of Distributed and Dependable Systems
Malostranske namesti 25, Prague 1, 118 00, Czech Republic
e-mail: {hnetynka,kofron}@d3s.mff.cuni.cz

R. Lee (Ed.): Software Engineering, Artificial Intelligence, Networking, SCI 443, pp. 117–128.
springerlink.com

with explicitly defined provided and required services and behavior. Composition is done via binding the component interfaces together.

From the view of composition, component models can be divided into *flat* and *hierarchical* ones. A flat model allows composition only on a single level while a hierarchical one allows it on multiple levels of nesting, i.e. components can be composed of other subcomponents.

Many component frameworks allow for transparently distributed applications (e.g. SOFA 2 [5], Fractal [4]). It means that during development it does not matter where particular component will be deployed at run-time. The glue code for component interconnections is automatically generated before execution based on the deployment decisions – allocation particular components onto particular computers in the network. Importantly, components of a single system can be deployed differently every execution and thus allowing for optimal utilization of available computer resources based on their current actual load.

The deployment of component-based system is standardized in the OMG Deployment and Configuration of Component-based Distributed Applications Specification (OMG D&C) [11]. The specification defines all necessary meta-models for deployment, however it does not prescribe any algorithm for actual deployment. Even more, the specification assumes that hierarchical components exist only at design time, while during deployment, the system is flattened. Nevertheless, such flattening cannot be applied in those component frameworks in which hierarchical components exist during their whole life-cycle, i.e. even at run-time (e.g. aforementioned SOFA 2 and Fractal).

In this paper we present an algorithm for automated deployment of hierarchical component systems, in which hierarchical components exist at run-time. The algorithm tries to find an optimal deployment with respect to the performance of the deployed application and utilization of the computers. The structure of the paper is as follows. Section 2 presents a basis for our work and requirements imposed on the deployment algorithm. Section 3 describes the algorithm. Section 4 presents an implementation of the proposed algorithm while Section 5 discusses related work, and Section 6 concludes the paper.

2 Background

An optimal deployment of a component system is typically understood as an assignment of the system's components to particular computers (deployment nodes) such that the system meets some criteria (typically that it has maximized performance). To find out the optimal deployment is a complex problem. A naive solution is to recursively try all possible assignments but such an algorithm has an exponential complexity. There exist several works offering algorithms for finding optimal deployment automatically, typically by applying heuristics to overcome complexity of the problem (a downside is that obtaining the optimal deployment is not guarantied) [8, 6, 14]. Nevertheless all of them work for flat component models only.

As a basis for our algorithm deploying hierarchical components, we have chosen the algorithm in [14] as it seems to offer best results and also it is the most recent one. Now we will briefly describe it.

2.1 Original Algorithm

The algorithm attempts to maximize performance of the system. As a metric for evaluating the performance, it uses CPU time and disk usage time required by a component per visit (a call of a provided method). Additionally, the algorithm assumes that computers are interconnected via a high bandwidth network (i.e. it is intended primarily for deployment at a local area network) and also it assumes that each component of a system can be deployed to any computer. The algorithm assumes the following definitions: (a) *component makespan* ($make_C$) is the value of component's CPU or disk demands per visit (whichever is greater), (b) *machine makespan* ($make_M$) is the value of machine's total CPU or disk execution times (whichever is greater) calculated using data from the deployed components, and (c) *system makespan* ($make_S$) is the maximum value of $make_M$ in the network. To reduce the complexity, the algorithm works without backtracking. In general, it selects an ordering of components and then places components one by one onto the computers in the network. In the paper with the original algorithm [14], there are proposed several heuristics for selecting the order of components and also for selecting a target computer for the particular component. Based on the evaluation, the authors selected the best heuristic for each of them. They are as follows. As a computer for deployment of a particular component the one with minimal $make_M$ is selected. The ordering of components is not precomputed but evaluated on the fly – as a next component for deployment, the one having the highest demands per execution in the dimension of the current highest makespan resource (CPU or disk) is chosen. With the second heuristic, the problem of selecting the first component to be deployed arises. It is solved by letting the heuristic start with each of the components as the first component and thus generating n different orderings in total. At the end the best ordering is chosen, i.e. the ordering with the smallest $make_S$.

2.2 Additional Requirements

The algorithm above does not deal with hierarchical components and thus cannot be directly used in the frameworks such as SOFA 2 and Fractal. Additionally, it assumes that components have only a single input point, which does not hold for most of the contemporary component frameworks where components can have multiple interfaces, each with several methods. Finally, the algorithm assumes that each component can deployed to any computer in the network, which is typically not true; components can require particular services available and/or they can require that several interconnected components are deployed at the same computer.

Thus, the additional requirements laid on the algorithm for automated deployment of component-based systems (apart from being fast and finding an optimal solution that maximizes the performance) are support for: (1) hierarchical components, (2) multiple component interfaces and multiple methods per interface, and (3) additional deployment location constraints.

3 Automated Deployment of Hierarchical Components

As already mentioned, we use the algorithm briefly described in Sect. 2.1 as a basis and extend it to also support the requirements defined in Sect. 2.2.

To support hierarchical components and location constraints, our algorithm does not work directly with components but with *deployment units*. A deployment unit is either: (1) a primitive component, or (2) a composite component (without its sub-components), or (3) a group of interconnected primitive components that have to be deployed on the same computer plus all composite components that participate on delegation of method calls between these primitive components.

In SOFA 2, Fractal, and other frameworks with hierarchical composition, composite components do not contain any functional code; they only delegate method calls on their interfaces to interfaces of their sub-components. If a composite component and its sub-components with a delegated interface are deployed to different computers, this delegation consumes additional resources (unlike when deployed to the same computer). Thus, it is desirable to deploy such components to a single computer.

In the original algorithm, the resource demands are simplified (CPU and disk) and specified 'per visit'. However, in the case a component interface has multiple methods, it is not sufficient, since a call to each method can result in different demands. Thus resource demands have to be specified per method; for the deployment algorithm, total resource demands of each component have to be computed from the – see Sect. 3.2. Additionally, our algorithm deals with additional *component requirements*, such as presence of a particular service on the computer (e.g. version of the Java platform or ability to show graphical UI). A difference between resource demands and component requirements is that the demands "consumes" particular capabilities of a computer (i.e. CPU usage, disk usage, etc.) while component requirements does not "consume" a capability (i.e. for example in the case of ability to show GUI it does not matter whether there is deployed single component or five).

For describing the component requirements of components and capabilities of computers, we use the corresponding part of the OMG Deployment and Configuration of Component-based Distributed Applications Specification (OMG D&C) specification. Requirements and capabilities are specified by the following attributes: name, value, and kind. The kind classifies the requirement/capability and based on it, the requirements are matched against capabilities. There are following kinds: attribute, maximum, minimum, and capacity.

3.1 Deployment Algorithm

Now, we can describe the deployment algorithm we propose in this paper (Alg. 1). The algorithm assumes that each component has already its total resource demands described.

Algorithm 1: Deployment planning

Input: Set of components
Result: All components are deployed

deplUnits ← `GetDeploymentUnits(`*input set of components*`);`
compositeComponents ← `ExtractCompositeComponents(`deplUnits`);`
orderedDeplUnits ← Get order of deplUnits using Algorithm 3;
Deploy orderedDeplUnits using Algorithm 2;
Order compositeComponents according to a nesting level such that components with higher nesting level are put ahead;
Deploy compositeComponents using Algorithm 4;

The algorithm works in several steps. First, deployment units are identified. Resource demands and requirements are calculated from all of the components in the unit. Then, a set of the units composed of composite components only is extracted from the complete set of the units. Next, the deployment units in deplUnits are ordered using Alg. 3 and deployed using Alg. 2. Finally, the composite components in compositeComponents order based on the decreasing level of nesting in the system architecture are subsequently deployed via Alg. 4. The chosen ordering ensures that whenever a component is to be deployed, all of its sub-components have been already deployed.

Algorithms 2 and 3 are just minor modifications of the algorithms from [14].

Algorithm 2: Deployment of deployment units in the given ordering

Input: List of deployment units in the given ordering
Result: All deployment units are deployed

foreach *Deployment unit* U_i **do**
  foreach *Computer* D_j **do**
    Mock-deploy the deployment unit U_i on the computer D_j;
    Note the value of $make_M$;
    Cancel the mock-deployment;
  end
  Choose the computer D with the minimum value of $make_M$;
  Deploy the deployment unit U_i on the computer D and update the system;
end

The original algorithm always produces a solution. However, since we suppose additional restrictions and requirements put on deployment, our algorithm does not guarantee to always produce a solution (in the cases when no solution exists).

Algorithm 3: Selecting the ordering of deployment units for deployment

Input: Set of deployment units
Output: Ordering of the deployment units

foreach *Deployment unit* U_{start} **do**
  $U_{current} \leftarrow U_{start}$;
  while *exists a deployment unit that is not deployed* **do**
    Deploy $U_{current}$ using Algorithm 2;
    Find the most loaded resource in the system;
    $U_{current} \leftarrow$ Deployment unit with the highest demand in the dimension of the most loaded resource;
  end
  Save the ordering along with the resulting system makespan;
  Reset the deployment;
end
Output the ordering which results in the least system makespan;

Algorithm 4 deploys composite components. First it attempts to deploy a composite component to the same computer as sub-components with delegated interfaces are deployed. If it is not possible, Alg. 2 is employed.

Algorithm 4: Deployment of composite components in the given ordering

Input: List of composite components in the given ordering
Result: All composite components are deployed

foreach *Composite component* C_i **do**
  computers $\leftarrow$ Get set of computers where sub-components of C_i should be deployed;
  foreach *Computer* D_j *in* computers **do**
    Deploy the composite component C_i on the computer D_j and update the system;
    if *the deployment is successful* **then**
      break;
    end
  end
  if *the composite component* C_i *is NOT yet deployed* **then**
    $U \leftarrow$ Create a deployment unit from the composite component C_i;
    Deploy U using Algorithm 2;
  end
end

3.2 Resource Demands

As aforementioned, resource demands for our algorithm are defined per method. As resource demands, the time necessary for execution of a method and disk usage of a method (i.e. the same resources as in the original algorithm) are taken into account. However, specifying resource demands per method is not sufficient since

most contemporary component frameworks allows *active components*, i.e. those that feature their own threads. Thus resource demands have to defined per active thread, too.

Also, a question is which units should be used for specifying resource demands. Intuitively, they can be specified in time units (e.g. seconds), which for disk usage are fine, but for CPU usage it would cause problems, since it depends on the speed of the particular CPU used. A better solution (inspired by [3]) is to use general units such as the number of CPU operations per second and to calculate the actual time from them and from the description of the particular hardware.

For automated deployment, resource demands specified per method/thread are still not enough and resource demands have to be calculated per whole components. However, it cannot be done as a simple summarization of all method/thread demands as it depends on the actual usage of the component in a system (i.e. a single component can have different demands when it is used in different systems). To overcome this we need to know behavior of all of the components. By behavior we mean which methods are called (and how many times) by a component on its required interfaces as a reaction to a received call on its provided interface. And similarly for active threads, i.e. which methods are called by a component on its required interfaces during execution of a thread.

A suitable formalism for such a behavior description are *behavior protocols* [12]. They capture a component behavior in the sense of received calls and reactions to these received calls, however they do not contain information on the frequency particular methods are called. Thus we have created an extension of the behavior protocols to capture such information. We have extended definition of the *alternative* operator (which allows specification that from a list of methods a single one is called) by numbers capturing probability of a method call (from the given list). Also, we have extended definition of the *repetition* operator (which allows specification that a method is called in a cycle) by the most probable number of repetitions. Both of these extensions are optional – an alternative without specified probabilities presumes that the probability of calls are equally divided, and a repetition without a number is the same as a repetition with 1.

With the total component resource demands computed, the automated deployment algorithm has enough information and can be applied.

4 Implementation for SOFA 2

To evaluate it, we have implemented the algorithm for the SOFA 2 component framework [1]. The SOFA 2 deployment environment consists of a set of distributed containers and components deployed to them. The whole deployment infrastructure is quite similar to the OMG D&C specification; the components requirements and container capabilities are described exactly according to the specification.

[1] The implementation of the SOFA 2 framework is available at `http://sofa.ow2.org/`. The complete framework is open-source distributed under the LGPL license.

The only downside of using our proposed deployment algorithm is that behavior with the probabilities has to exist for all of the components to be deployed. And to specify it manually can be quite complicated even for authors of the particular component as the probabilities can depend for example on sizes of method parameters, etc. On the other hand, behavior can be "observed" from the executed system. Thus we have implemented a "logging mode", in which a system can be executed and all information about method calls among components are logged. After the system execution finishes, logged calls are automatically processed and the behavior protocols of the particular components updated with the probabilities. The "logging mode" was easily implemented thanks to the SOFA 2 extensible component structure by aspects applied at deployment time [10].

Now in SOFA 2, a user can decide whether he/she either deploys components of a system manually or uses the automated deployment. In the second case, he/she can just launch the system immediately or can review the created deployment and then launch it (Figure 1 shows a tool for reviewing and launching the created deployment).

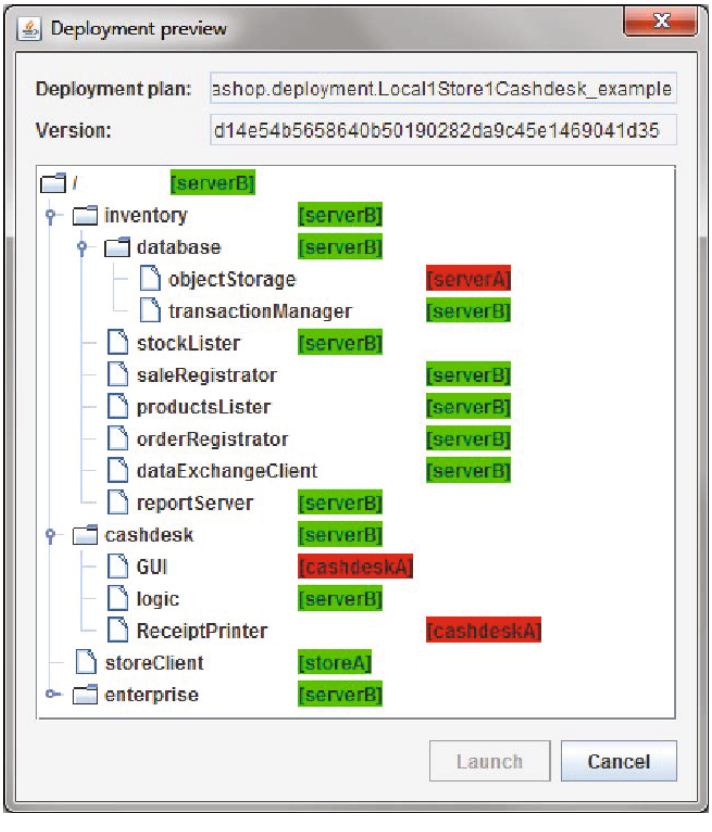

Fig. 1 SOFA 2 deployment tool

We tested our deployment algorithm and tools on several non-trivial examples, the most interesting one has been the CoCoME system [13], which is a real-life-like component application intended for comparison of component models and frameworks. The application models a system for a chain of stores with cash-desks, storage, etc. (the tool in Figure 1 shows the deployment). We performed a number of deployments with differently set of available computers with different setting. The algorithm always produced expected (i.e. optimal) deployment.

5 Related Work

As far as we know there is no actual implementation of an automated deployment algorithm for hierarchical component systems.

The already mentioned Fractal component model, which is similar to SOFA 2 and in which composite components at run-time also exist, does not directly address deployment at all. However, there exists the Fractal Deployment Framework (FDF)[2]. FDF defines the necessary deployment infrastructure and tools. Also it introduces a high-level description language for specifying deployment (i.e. assigning components to computers, etc.) according to the infrastructure which deploys applications. Nevertheless, a description in this language has to be prepared manually by a user; there is no automated creation of it.

FDF could be rather straightforwardly extended with our automated deployment, since there also exists an implementation of the behavior protocols for Fractal and "logging mode" can be implemented via component aspects, too.

There exist several implementations of automated deployment for flat component models (or technologies close to components, e.g. services – several of them already mentioned in Sect. 2). In [6], the authors construct a deployment planner for composition of web services, which are treated as software components. The composition of web services is done by *Reo circuits* [1]. A specification of the distributed environment is given by a description of computers and their capabilities. The capabilities are meant to be software capabilities (e.g. which implementations of the Reo channels the computer can support). Hardware capabilities such as CPU and disk speed, memory size, etc. are regarded as not important (the authors claim they focus on software abstraction only). However, the offered deployment planning is thus rather limited, especially in a heterogeneous deployment environment where each computer can have different capabilities.

Another interesting approach of automated deployment is used in the *Sekitei* planner [8]. The *Sekitei* planner uses an AI (artificial intelligence) planning algorithms. It is implemented as a pluggable module for Java component-based frameworks and used in the *Smock* framework [7] that serves as a run-time environment of the *Partitionable Services Framework* [7]. In the framework, services can be

[2] `http://fdf.gforge.inria.fr/`

composed of several components. Also, the framework allows transparent migration and replication of the components. The main purpose of the migration is to bring services closer to a client. In comparison to our method, the Sekitei planner solves more general problem than we consider. During the deployment planning of an application, the planner also decides, which particular set of components (from compatible ones) will be deployed (in our method the set of components to be deployed is given). Depending on the network and capabilities of the computers and network connections the planner may introduce some auxiliary components – either new ones or already available components could be reused. As the planner also considers capabilities of network interconnections, it can be used in a non-local network environment also. A downside is that due to its generality, the planning can consume a considerable amount of time.

Another solution for automated deployment is used in the *ProActive* framework [2]. ProActive is a Java open source framework for parallel, distributed, grid, and cloud computing. The framework is divided into several parts; the most important one from the view of this paper is *Scheduling*. ProActive Scheduling provides a framework for a job definition and execution. A job consists of tasks (which can be, e.g. Java or native applications, scripts) and dependencies among the tasks. The Scheduler then assigns tasks to the resources, i.e. Java virtual machines, that are managed by a Resource manager. Information for the Resource manager are supplied be an agent (a program implemented for a particular operating system allowing to launch the Java virtual machine and provide information about utilization). The Resource manager allows dynamic addition and removal of the resources. The scheduling algorithm simply deploys tasks of a job one by one (based on the task dependencies) on available resources provided by the resource manager. One resource may process only one task at a time – until the task is processed the resource is unavailable. This differs from our approach where more components may be deployed to the same container.

ProActive also defines a hierarchical component model for developing applications, which is in fact the Fractal component model. The deployment descriptor assigning components to containers has to be prepared manually.

A work related to the proposed approach from the specification point of view is the Palladio Component System (PCM) [9]. Here, finite state machines enriched by probabilities of transitions and information on resource consumption are used to compute the extra-functional properties, such as the response time of a service (provided method) and resource consumption. Resource consumption (CPU, disk, time, memory) can be either specified by a constant or a distribution function depending on an input parameter (e.g., the size of the array to be processed by the service). The information is, however, not used for computing a suitable deployment; PCM does not focus on particular runtime configurations, the hierarchical component models exist just at the design time. Although the information contained in PCM models could be used in our approach, they are too detailed and using them as an input for our deployment algorithm would make the user specify a lot of unused information.

6 Conclusion

In this paper, we have proposed an algorithm for automated deployment for component models with hierarchical components, where hierarchical components exist also at runtime. The proposed algorithm is based on a deployment algorithm [14] for flat component systems. The algorithm considers requirements of individual components to be deployed and also provided capabilities of the deployment environment. To be fast enough, the algorithm uses several heuristics.

To evaluate the algorithm we have implemented it for the SOFA 2 component framework. Application can be executed in a logging mode, which collects information about usage of individual components and then the information is used by the deployer tool for automated deployment.

Currently we are continuing with the implementation of additional tools in order to make deployment and general usage of the SOFA 2 framework more user-friendly. Also we are working on dynamic migration of running components to allow load-balancing based on components' resources consumption and provided capabilities of the run-time environment.

Acknowledgements. This work was partially supported by the Grant Agency of the Czech Republic project P202/11/0312 and partially by the Charles University grant SVV-2012-265312.

References

1. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical Structures in Computer Science 14(3), 329–366 (2004), doi:10.1017/S0960129504004153
2. Baduel, L., Baude, F., Caromel, D., Contes, A., Huet, F., Morel, M., Quilici, R.: Programming, composing, deploying for the grid. In: Cunha, J.C., Rana, O.F. (eds.) Grid Computing: Software Environments and Tools, pp. 205–229. Springer, London (2006), doi:10.1007/1-84628-339-6_9
3. Becker, S., Koziolek, H., Reussner, R.: Model-Based performance prediction with the Palladio component model. In: Proceedings of WOSP 2007, Buenes Aires, Argentina, pp. 54–65. ACM (2007), doi:10.1145/1216993.1217006
4. Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.-B.: The FRACTAL component model and its support in Java. Software: Practice and Experience 36(11-12), 1257–1284 (2006), doi:10.1002/spe.767
5. Bures, T., Hnetynka, P., Plasil, F.: SOFA 2.0: Balancing advanced features in a hierarchical component model. In: Proceedings of SERA 2006, Seattle, USA, pp. 40–48. IEEE CS (2006), doi:10.1109/SERA.2006.62
6. Heydarnoori, A., Mavaddat, F., Arbab, F.: Towards an automated deployment planner for composition of web services as software components. ENTCS 160, 239–253 (2006), doi:10.1016/j.entcs.2006.05.026

7. Ivan, A.-A., Harman, J., Allen, M., Karamcheti, V.: Partitionable services: A framework for seamlessly adapting distributed applications to heterogeneous environments. In: Proceedings of HPDC-11, Edinburgh, UK, pp. 103–112. IEEE CS (2002), doi:10.1109/HPDC.2002.1029908
8. Kichkaylo, T., Ivan, A., Karamcheti, V.: Constrained component deployment in wide-area networks using AI planning techniques. In: Proceedings of IPDPS 2003, Nice, France. IEEE CS (2003), doi:10.1109/IPDPS.2003.1213075
9. Koziolek, H., Becker, S., Happe, J., Reussner, R.: Evaluating Performance of Software Architecture Models with the Palladio Component Model. In: Model-Driven Software Development: Integrating Quality Assurance, pp. 95–118. IDEA Group Inc. (2008)
10. Mencl, V., Bures, T.: Microcomponent-based component controllers: a foundation for component aspects. In: Proceedings of APSEC 2005, Taipei, Taiwan, pp. 729–738. IEEE CS (2005), doi:10.1109/APSEC.2005.78
11. OMG: Deployment and configuration of component-based distributed applications specification. OMG document formal/2006-04-02 (2006)
12. Plasil, F., Visnovsky, S.: Behavior protocols for software components. IEEE Transactions on Software Engineering 28(11), 1056–1076 (2002), doi:10.1109/TSE.2002.1049404
13. Rausch, A., Reussner, R., Mirandola, R., Plášil, F. (eds.): The Common Component Modeling Example. LNCS, vol. 5153. Springer, Heidelberg (2008), doi:10.1007/978-3-540-85289-6
14. Sharma, V.S., Jalote, P.: Deploying Software Components for Performance. In: Chaudron, M.R.V., Ren, X.-M., Reussner, R. (eds.) CBSE 2008. LNCS, vol. 5282, pp. 32–47. Springer, Heidelberg (2008)
15. Szyperski, C.: Component software: beyond object-oriented programming, 2nd edn. Addison-Wesley, Boston (2002)