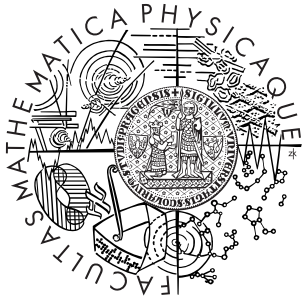
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Tomáš Kučera

Cloud computing using a hierarchical component system

Department of Software Engineering

Supervisor: RNDr. Petr Hnětynka, Ph.D.

Study Program: Computer Science, Software Systems

2010

I would like to thank Petr Hnětynka for ideas and comments during the preparation of the thesis.

I hereby declare that I have elaborated my master thesis on my own and listed all used references. I agree with making the thesis publicly available.

Prague, December 9, 2010 Tomáš Kučera

Contents

1 Introduction **6**
- 1.1 Goals and structure of the thesis . . . 7

2 Overview **8**
- 2.1 Cloud computing . . . 8
- 2.2 Component-based development . . . 9
  - 2.2.1 Component definition . . . 9
  - 2.2.2 Component model . . . 10
- 2.3 SOFA 2 . . . 10
  - 2.3.1 Application life-cycle . . . 11
  - 2.3.2 Runtime environment . . . 12
- 2.4 Behavior protocol . . . 12

3 Automated deployment planning **14**
- 3.1 Assumptions and requirements . . . 14
- 3.2 Possible approaches . . . 15
  - 3.2.1 AI planning techniques . . . 15
  - 3.2.2 Graph based approach . . . 16
  - 3.2.3 Heuristic based approach . . . 17
- 3.3 Selection of the algorithm . . . 19
- 3.4 Proposed algorithm . . . 21
  - 3.4.1 Resource demands of a component . . . 23
  - 3.4.2 Component's requirements and deployment dock's capabilities 27
- 3.5 Total resource demands of components . . . 28
  - 3.5.1 Discrete-time Markov chains . . . 30
  - 3.5.2 Automated observation of an application behavior . . . 31

4 SOFA 2 extensions **35**
- 4.1 Component description . . . 35
  - 4.1.1 Frame meta-class . . . 35
  - 4.1.2 Architecture meta-class . . . 37
- 4.2 Deployment dock description . . . 40
  - 4.2.1 Management of deployment dock's capabilities . . . 40

4.2.2 Management of resource usages . . . 41
4.2.3 Conversion of component's resource demands . . . 42
4.3 Behavior protocol extension . . . 43
4.4 Deployment planning . . . 44
4.5 Application launch and termination . . . 46
4.6 Implemented tools . . . 47
4.6.1 Cushion methodinfo . . . 47
4.6.2 Logging aspect . . . 48
4.6.3 Tool sofa-update-frames . . . 48
4.6.4 Cushion autodeploy . . . 49
4.7 Cloud Gate application . . . 50
4.7.1 Automated generation of a deployment plan . . . 52

5 Example SOFA 2 application **54**
5.1 Adding new information to components . . . 54
5.2 Test of the automated deployment planning . . . 55

6 Related work **58**
6.1 Julia . . . 58
6.2 Sekitei . . . 58
6.3 ProActive . . . 59

7 Conclusion and future work **60**

Bibliography **61**

A Contents of the CD-ROM **63**

Název práce: Cloud computing s využitím hierarchických komponentových systémů
Autor: Tomáš Kučera
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí práce: RNDr. Petr Hnětynka, Ph.D.
E-mail vedoucího: hnetynka@d3s.mff.cuni.cz

Abstrakt: Cloud computing je v současnosti populární výpočetní model, kde počítače propojené v síti (tzv. cloud) společně nabízí velký výpočetní výkon. SOFA 2 je hierarchický komponentový systém s distribuovaným běhovým prostředím, které je vhodné pro využití výhod cloud computingu. Aplikace jsou složené z komponent, každá komponenta může běžet na jiném stroji v síti. Rozmístění jednotlivých komponent ovlivňuje celkovou výkonnost aplikace a využití zdrojů v síti. Z tohoto důvodu musí být rozmístění komponent pečlivě naplánováno. Tato práce navrhuje algoritmus pro automatizované plánování rozmístění hierarchických komponentových aplikací a implementuje jej v SOFA 2 systému. Algoritmus pracuje s požadavky komponent a s dostupnými zdroji v síti za účelem zvýšení celkového výkonu aplikace. Dále tato práce navrhuje a implementuje rozšíření SOFA 2 systému, která umožňují využívat daný systém pro účely cloud computingu.

Klíčová slova: hierarchické komponenty, automatizované rozmístění, plánování

Title: Cloud computing using a hierarchical component system
Author: Tomáš Kučera
Department: Department of Software Engineering
Supervisor: RNDr. Petr Hnětynka, Ph.D.
Supervisor's e-mail address: hnetynka@d3s.mff.cuni.cz

Abstract: Cloud computing is nowadays a popular computing paradigm. Computers are interconnected via network and jointly offer a lot of computing performance. SOFA 2 is a hierarchical component system offering a distributed run-time environment; therefore, it is a suitable environment for cloud computing. Applications are composed from components; each component may run on different computer in the 'cloud'. The deployment of the components influences the overall performance of the application and the utilization of resources in the 'cloud'; therefore, it has to be planned carefully. In this theses, an algorithm for automated deployment planning of hierarchical component-based applications is proposed and further implemented in the SOFA 2 system. The algorithm incorporates components' demands and machines' resources in order to maximize performance of the deployed applications. The thesis also proposes and implements extensions that allow using the SOFA 2 component system as an actual cloud platform.

Keywords: hierarchical components, automated deployment, planning

Chapter 1

Introduction

Cloud computing is a word that can be heard nowadays all around you. It is a computing paradigm and its main idea is to provide scalable and often virtualized resources as a service over a net. It can be compared with an electricity grid; you do not have to have a generator to make your own electricity, you just use electricity from the grid and you pay only for what you consume. And now this behavior is also viable in computer world. Users may pay only for computer's performance they need and they do not have to have knowledge of, or control over the technology infrastructure used in the 'cloud'. This simplifies the life of the users as they do not need to buy their own hardware.

Because of the fact that the *Cloud computing* is becoming more popular these days it is necessary to have means for developing and launching applications in the 'cloud'. In application development branch component-based development is very convenient. Large applications are build from smaller components which can be utilized also in another applications. *SOFA 2* is an advanced hierarchical component system offering a distributed run-time environment; therefore, it is a suitable candidate for use in connection with the *Cloud computing*.

SOFA 2 system currently lacks support for easy launch of SOFA 2 applications in the 'cloud'. In particular deployment of such applications has to be prepared manually which is not convenient especially with regard to managing available resources in larger clouds. Thus an automated deployment planning is needed for easy launch of applications in the cloud.

Purpose of this thesis is to add a support of cloud computing to the *SOFA 2* system. In particular by adding support for automated generation of deployment plans of applications based on the current status of the cloud; and by creating appropriate user tools for these actions.

1.1 Goals and structure of the thesis

Main goal of this thesis is to add support for automated generation of deployment plans to the *SOFA 2* system. This can be achieved by firstly looking for algorithms that would solve this problem, comparing these algorithms and then selecting the suitable one. Secondly it is necessary to implement the selected algorithm and to create relevant supporting tools. Further it is required to simplify connection to the cloud and to improve and automate launching of SOFA 2 applications via new system tray application.

The structure of the thesis is as follows: Chapter 2 provides background information, Chapter 3 describes the issue of automated deployment planning and its solution, Chapter 4 mentions SOFA 2 extensions, Chapter 5 shows implemented deployment planning on an example, Chapter 6 focuses on related work and, finally, Chapter 7 concludes this work.

Chapter 2

Overview

This chapter provides general information that will help the reader to understand the rest of the thesis.

2.1 Cloud computing

As mentioned in Chapter 1, *Cloud computing* is an ubiquitous word. But what does it really mean? It is a computing paradigm and it can be defined as an updated version of utility computing, basically virtual servers available over the Internet or it can also be defined as anything outside your firewall. According to [15] the cloud computing consists of various components:

SaaS Software as a Service, a single application is delivered through the browser to thousands of customers. Example applications: Salesforce.com, Google Apps.

Utility computing storage and virtual servers can be accessed on demand over the network. Example providers: Amazon.com, IBM, SUN.

Web services in the cloud web service providers offer APIs that enable developers to exploit functionality over the Internet, rather than delivering full-blown applications (as it is in SaaS). Example: APIs offered by Google Maps.

Platform as a service the whole development environment is delivered as a service over the Internet rather than just one application. You can build your own applications that run on the provider's infrastructure and are delivered to your users via the Internet from the provider's servers. Example: Salesforce.com's Force.com, Google App Engine.

MSP Managed Service Providers, a managed service is basically an application exposed to IT rather than to end-users, such as a virus scanning service for e-mail. Example provider: SecureWorks, IBM.

All of these components have a single common feature – delivering IT as a utility (like electricity).

2.2 Component-based development

The main idea of the component-based development is to develop new applications from existing components. This assumption has consequences for the entire development process of the new application [8]:

- development process of component-based application is separated from development process of components. The components should have already been developed, tested and possibly used in other application at time when the component-based development process starts.
- new process of searching and validating existing components has to be added into development life-cycle. Less time is spend on implementation (we already have some components implemented) but more time has to be spend on proper selection and especially validation of the potential components.
- design and implementation of components is harder because one of the main purposes of these components is their possible reuse. Thus the design should be general enough and the implementation should count with all possible usages of the component.

Implemented components are usually stored with their metadata[1] in a repository, where these are accessible for further development of new applications.

Advantages of the component-based development stems mainly from reusing the old components, thus the development is faster and final application has less bugs. Another advantage of the component-based applications is their easier maintenance as the components can be easily changed or upgraded if necessary.

2.2.1 Component definition

There are more possibilities how to characterize and define a software component [3]. Classical definition is from Clementz Szyperski[2]:

> *"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 software component has precisely-defined behavior and interfaces (provided and required) for communication with other components. It can be seen as a black-box entity so it can be easily reused in different contexts without need of modifying internal structure of the component.

[1] additional description of the component, e.g. behavioral description, measured results of component performance, ...

[2] The definition was developed in the first Workshop on Component-Oriented Programming (WCOP'96) at ECOOP'96 in Linz.

As there is no general list of detailed features that the component has to specify, we should look at the concept of component only in the context of its respective *component model*.

2.2.2 Component model

Component model specifies rules and semantics for proper work with components. It specifies how the components are created, how they could be composed together (this depends on the model type, whether it is flat or hierarchical), what communication style is permitted between components or how the assembly and deployment of components is performed.

Implementation and runtime support for a particular component model is called *component system/platform*.

2.3 SOFA 2

SOFA 2 is an advanced hierarchical component system offering a distributed runtime environment [6, 5, 21]. The component model of *SOFA 2* system is specified as an EMF meta-model [9]. This brings simplification as the EMF technology offers tools for automated generation of a repository with standardized API for working with components.

In *SOFA 2*, a component can communicate with other components only via designed provided and required interfaces. It can be viewed as a black-box and also as a gray-box entity. Black-box view is performed by a *component frame*, which defines the type of the component. It specifies provided and required interfaces and it can also specify the component's behavior (e.g., via behavior protocol).

A gray-box view of a component is defined by a *component architecture*. The architecture implements particular frame and can be either *primitive* or *composite*. Primitive architecture contains direct implementation of a component (the 'business' code), whereas composite architecture contains sub-components specified as frames or architectures. Composite architecture then specifies bindings among its sub-components. It does not contain any 'business' code, it only delegates calls from its interfaces to interfaces of its sub-components.

A component may be used in different applications in different ways. Therefore, the component can be specified by additional parameters to ensure generality of the component. These parameters are set during deployment to tune up the component to its specific usage. Parameters can be specified either to a frame or to an architecture.

In addition to the software components the connectors are also defined as first-class entities. This enables effective connection of components according to their distribution. Various communication styles can be specified, such as *method invocation*, *message passing*, *streaming* or *distributed shared memory*.

SOFA 2 also supports advanced management of components' control functionality. This is based on a micro-component model, simple flat component model that enables easy construction and extension of the control part of components via micro-components composition. Composition of micro-components is stated in *component aspects*.

There is also a basic support for service-oriented architectures (SOA). Provided interface can be exported as a service (e.g. via Web Service) and required interface can be mapped on an externally available service.

2.3.1 Application life-cycle

Application life-cycle consists of these steps:

- component development
- application assembly
- application deployment and execution

All code and metadata produced in each step are stored in a repository. For the development and management of SOFA 2 applications the command line tool named *Cushion* can be used.

Application development

Application development usually consist from composition of ready components. A new architecture is created for the new application. Then either available sub-components are used from repository or new components are created.

Application assembly

As application architecture is usually described mainly by frames, particular architecture for each frame has to be selected in this step. The result (mapping from application's components to architectures) is stored in an *assembly descriptor*.

Application deployment and execution

In order to launch a SOFA application, a *deployment plan* has to be created. This plan contains assignment of each component of the application to a particular deployment dock (a place where the component runs). Also additional parameters for components are specified in the deployment plan.

When the deployment plan is ready, connectors are generated and stored in the repository. Now the application can be launched.

2.3.2 Runtime environment

A SOFA 2 runtime environment is called *SOFAnode*. It is a distributed runtime environment which can comprise a number of computers. It consists of a *repository*, a *global connector manager*, a *deployment dock registry* and a number of *deployment docks*.

The *repository* is the most important part of a SOFAnode as it contains implementation and metadata of components. It is used both in the development time and in the run time.

A *deployment dock* serves as a container for components. It provides necessary infrastructure for component instantiation and runtime management, such as starting, stopping and updating components. All necessary component's code and metadata are retrieved from the repository.

The *deployment dock registry* registers all running deployment docks in a SOFAnode, thus it is used to look for other deployment docks when someone needs to communicate with them.

The *global connector manager* is responsible for managing all units of connectors in a SOFAnode and for interconnection of the units that belong to one connector. Thus the interconnection of components can work properly.

2.4 Behavior protocol

Behavior protocol serves as a specification describing a behavior of a component [18]. The specification is similar to regular expressions. It describes activity on the component's interfaces. That means all possible sequences of method calls and returns on provided and required interfaces that the component can perform.

The simplest behavior protocol can be either calling of the method 'm' on the required interface 'itfA' (written as `!itfA.m`), acceptance of the method 'm' on the provided interface 'itfB' (written as `?itfB.m`) or the `NULL` symbol which means empty trace (no sequence of method calls on interfaces). A more complex behavior protocol can be constructed using operators specified in Table 2.1. If there are any actions to be done during acceptance of a method on a provided interface, they are specified inside curly braces (e.g. `?itfB.m{!itfA.m;!itfA.n}`).

In SOFA 2 a behavior protocol of a component can be specified to component's frame. Its main purpose is a possibility to check whether the component composition of an application is correct. That means that the methods of each component are invoked in a correct order and no dead-lock can occur due to an unexpected communication between components.

Operator	Description
`A ; B`	*sequencing*; the set of traces formed by concatenation of a trace generated by `A` and a trace generated by `B`
`A + B`	*alternative*; the set of traces which are generated either by `A` or by `B`
`A*`	*repetition*; equivalent to `NULL + A + (A;A) + (A;A;A) + ...` where `A` is repeated any finite number of times
`A | B`	*and-parallel*; an arbitrary interleaving of event tokens of traces generated by `A` and `B`
`A || B`	*or-parallel*; stands for `A + B + (A | B)`

Table 2.1: Operators for constructing behavior protocols. `A` and `B` stands for a protocol.

Chapter 3

Automated deployment planning

As mentioned in Section 1.1, the main goal of this work is to add support for automated generation of deployment plans to the SOFA 2 system. Currently all assignments of components to deployment docks in a deployment plan (see Section 2.3.1) has to be done manually. That is inconvenient especially when the application consists of many components and a lot of the deployment docks are running in the SOFAnode.

Following section covers possible approaches how to solve the deployment planning problem. In Section 3.3 one algorithm is selected and in Section 3.4 this algorithm is extended to accomplish our purposes.

3.1 Assumptions and requirements

Although SOFA 2 runtime environment is distributed and therefore it could be used also in wide-area network, it is assumed that the runtime environment will be used mainly in local high bandwidth networks. Every machine can run more then one deployment dock. High bandwidth network ensures that each deployment dock is connected with every other deployment dock and communication between them is fast.

An interface of a component can be marked with tag *local-communication*. This means that all components that are connected via this interface have to be deployed on one deployment dock.

Requirements on the deployment planning algorithm are following:

1. generate deployment plans that maximize performance of deployed applications
2. perform the deployment planning reasonably fast
3. support hierarchical components
4. support *local-communication* feature at interface specification

3.2 Possible approaches

In order to implement an automated deployment planning, suitable algorithm has to be found first. Possible approaches are described in the following sections.

3.2.1 AI planning techniques

Artificial Intelligence (AI) is a popular field in computer science. Therefore, planning problems in AI have been studied for a long time and thus their results could be used also in deployment planning. This idea was studied at New York University and the result of their investigation is a planner called *Sekitei* [13].

They propose a general planning model called as the *Component Placement Problem* (CPP) which is solved by the planner using AI planning algorithms. The component placement problem is defined by the following elements [13]:

Network topology The network topology is described by a set of nodes and links, each of them can have associated static and dynamic properties. The dynamic properties are non-negative real values, the static properties might be represented by Boolean values or real intervals.

Application framework The application is defined by sets of interface types and component types. Each component type specifies sets of provided and required interfaces. In addition, each interface is characterized by a set of component-specific properties that are defined as functions of other properties and have no semantics attached to them.

Component deployment behavior A component is deployed on a node only if the required interfaces are provided by other components on the node and if the node and link resources are sufficient. After deployment, the provided interfaces of the component become available on the node and the dynamic properties of the node are updated.

Link crossing behavior The link crossing behavior is described by interface specific functions. These functions describe how the interface properties are affected by the link properties when crossing the link, and how dynamic properties of the link are changed as a result of this operation.

Goal of the CPP In the simplest case, the goal is to put a component of a given type onto a given node. Other goals can include e.g. delivering a particular set of interfaces to a given node.

Solving the CPP problem with the Sekitei planner would be as follows: transform a framework-specific representation of the CPP into an AI-style planning problem; solve the problem with improved AI planning algorithm; convert the AI-style solution into a framework specific deployment plan.

In order to overcome scalability restrictions of the AI planning techniques some specific characteristics of CPP are exploited. The planning algorithm combines multiple AI planning techniques. It consists of four steps: regression phase, progression phase, plan extraction phase and symbolic execution. Each phase helps to reduce the search space. If a phase fails, the searching is returned to the previous phase. The first three phases guarantee correctness of the logical part of the plan whereas the fourth phase serves for checking resource conditions. Any arbitrary monotonic functions in resource preconditions are supported.

Inconvenient input for the planner is following: very strict resource constraints, multiple component types implementing the same interface, highly connected networks. With this input the planning can take a lot of time.

In [14] the CPP model was extended with cost functions and resource levels to target two goals: allowing the planner to find solutions in resource constrained situations and specifying preferences over possible plans.

Using resource levels, which have to be specified by a domain expert (possibly based on profiling results), the planner can find a solution also in resource constrained situations, because it does not allocate 'worst-case' network resources (considering the maximum possible utilization of the resource) but only minimal resources that will be needed. Thus the overall resource consumption is minimized. This functionality depends greatly on the actual specification of levels. Using multiple levels for each resource increases the size of the problem and negatively affects performance of the planner. However, it also permits identification of some resource conflicts at an earlier (and cheaper) phase of the search.

3.2.2 Graph based approach

In [11] authors construct a deployment planner for a composition of Web services as software components. The composition of Web services is done by *Reo circuits* [1], thus the specification of the Web application consists of the Web services, their requirements and constraints, and the Reo circuit used among them.

A specification of the distributed environment is given by a description of hosts and their capabilities. These capabilities are meant to be software capabilities (like which different implementations of Reo channels the host can support). Hardware parameters such as CPU and disk speed, memory size, etc. are regarded as not important due to the fact that they wish to focus on software abstraction and not hardware abstraction.

The deployment planning is defined as a constraint satisfaction problem. Resources available at different hosts should be optimally allocated and the requirements and constraints of the application should be accommodated. Moreover some quality of service (QoS) requirements of the resulting deployment plan are also considered.

The deployment problem is solved by a graph-based approach. There are made two graphs for this purpose: an *Application Graph*, which models a component-

based application as a graph of components connected by different channel types, and a *Target Environment Graph*, which describes the distributed environment as a graph of hosts connected by different channel types that can exist between every two hosts. The deployment planning of an application is then defined as a mapping of its application graph to its target environment graph, subject to maximizing the desired QoS requirements.

Currently only one QoS requirement is supported. Suppose that a charge applies for a usage of a particular host (when a component is deployed on this host). Then a deployment plan with a minimal cost has to be found. Such a deployment problem is regarded to be equivalent to the *Minimum Set Cover problem* in graph theory [11].

Definition 1 (Minimum Set Cover Problem)**.** Given a finite set U of n elements, a collection of subsets of U, $S = \{s_1, s_2, \ldots, s_k\}$ such that every element of U belongs to at least one s_i, and a cost function $c : S \to R$, the problem is to find a minimum cost sub-collection of S that covers all elements of U.

According to [11], the cost-effective deployment problem is converted to the minimum set cover problem in the following way:

- set $U = \{C_1, C_2, \ldots, C_n\}$, i.e., the components of the application are set as the elements of the universe
- set $S = \{CS_{H_1}, CS_{H_2}, \ldots, CS_{H_m}\}$ in which each CS_{H_i} corresponds to the host H_i, and it represents the set of components of the application that can be run on the host H_i
- define $c : S \to R$ such that it returns the cost of each host

The minimum set cover problem is a NP-hard problem so it cannot be solved in polynomial time. But there exist some greedy approximation algorithms that can find reasonably good answers in polynomial time. So the solution presented in [11] consists of these steps: first the cost-effective deployment problem is converted to the minimum set cover problem, then the minimum set cover problem is solved by using existing algorithms and finally the result is converted back to the deployment plan.

3.2.3 Heuristic based approach

In [20] the goal of their research is to find a deployment of software components that maximize performance of the application. Their solution is heuristic-based and incorporates the software architecture, component resource requirements, and the hardware specifications of the system.

The formal description of the deployment problem is following. Let $C_1, C_2, \ldots, C_n$ be n software components. Each component C_i is described by its per visit resource requirements: cpu_i, which denotes the component's average CPU processing time,

and $disk_i$, which denotes the average disk I/O requirements per visit of the component C_i.

Target environment consists of m machines $M_1, M_2, \ldots, M_m$. Each machine M_j is described by speed ratings of its CPU (r_{cj}) and disk (r_{dj}). These ratings express the multiplicative factors by which these devices are faster than those used to measure the per visit demands of the components.

The deployment problem involves finding an assignment of all components to particular machines such that the overall performance is maximized.

For the purpose of this study it is assumed that the machines are interconnected by high bandwidth network and that the communication time is small compared to the CPU and disk I/O times for each component visit. Therefore, the CPU and disk are considered as primary resources in the system.

An approach for deploying components is to pick one component at a time and deploy it on a particular machine. Deciding which machine the component should be deployed on is based on the resource utilizations of the machines and the resource demands of the component. Although the components are described by per visit resource demands, the total resource demands per user request or per job of each component have to be computed. A software component may be executed more than once or even less than once on an average for servicing a job. This depends on the software architecture. For calculating the visit counts of components, the software architecture can be modeled by *discrete-time Markov chain* (see Section 3.5.1).

The effectiveness of the final deployment depends on the order in which the components are selected for deployment as well as on how the deployment is actually done. Therefore, the problem of deploying components for maximizing performance is divided into two sub-problems:

- decide the best deployment for the components presented in given order
- select the order in which the components should be presented for deployment

Both sub-problems are solved by heuristics. Before explaining the heuristics, following terms are introduced:

component makespan $make_C$ is the value of component's CPU or disk demands per job, whichever is greater

machine makespan $make_M$ is the value of machine's total CPU or disk execution times (depends on the deployed components), whichever is greater

system makespan $make_S$ is the maximum value of $make_M$ from among all the machines in the system at the time of evaluation

The heuristics for deploying software components in the given ordering deploy the components one by one. Each component is processed only once, no backtracking is involved. First the heuristics tries to 'mock-deploy' a component on each machine

Algorithm 1: Deployment of components in the given ordering.

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

foreach *Component* C_i **do**
  foreach *Machine* M_j **do**
    Mock-deploy the component C_i on the machine M_j;
    Note the value of $make_M \leftarrow \max\{CPU\ exec.\ time, Disk\ I/O\ time\}$;
    Cancel the mock-deployment;
  end
  Choose the machine M with the minimum value of $make_M$;
  Deploy the component C_i on the machine M and update the system;
end

and records the resulting value of $make_M$. Then it picks the machine with the least value of $make_M$ and the component is deployed on that machine (see Algorithm 1).

Because of the fact that the first heuristics does not use backtracking, the order in which components will be deployed will affect the overall performance of the application. Thus an appropriate ordering of components for deployment is needed.

This is solved by the second heuristics which is dynamic in nature. It uses information from intermediate steps in the deployment process to select components and thus results in an on-the-fly ordering of components. The next component to deploy is selected based on the resource having the highest system makespan. Specifically the next selected component is the one having the highest demand per job in the dimension of the current system makespan resource.

Unfortunately there is a problem of choosing the first component. This 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 (the ordering which results in the least system makespan). Details are shown in Algorithm 2.

Based on the exhaustive testing the combination of these two heuristics perform very good results, outputting the best possible deployment or a deployment close to the best in more than 96% cases [20]. It is assumed that the component-based systems which need to be deployed are synchronous in nature. The performance impact of middleware or virtual machines were not considered.

3.3 Selection of the algorithm

In the previous section three possible approaches for solving automated deployment planning were presented. Here the approaches are compared and one algorithm is selected.

Algorithm 2: Selecting the ordering of components for deployment.

```
Input: Set of components
Output: Ordering of the components
foreach Component C_start do
    C_current ← C_start;
    while exists a component that is not deployed do
        Deploy C_current using Algorithm 1;
        Find the most loaded resource (CPU or Disk I/O) in the system;
        C_current ← Component 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;
```

The Sekitei system using AI planning techniques solves in fact more general problem than it is needed for SOFA 2. During the deployment of an application it decides on a particular set of components (from compatible ones) that will be deployed. Depending on the network and capabilities of the nodes/links the planner may introduce some auxiliary components for fulfilling application constraints, or already available components could be reused. The planner also works well in wide-area environments as it counts with the capabilities specified at links. Due to its generality and possibility of finding deployments also in resource constrained environments the planner could suffer from performance issues.

On the other hand the second approach is quite specific in its purpose. The hosts may specify only software capabilities (e.g., whether a software is available on the host) and no other types of capabilities or resource requirements are supported. That is quite limiting. Also the possibility of getting a deployment, which results in the lowest cost (when some amount has to be paid for the host usage), is not so interesting in SOFA 2 environment, as it is expected that the SOFA 2 environment will run mainly on local networks.

Finally the third approach is interesting as it allows finding a deployment that optimize the overall performance. It assumes that the host machines are connected with high bandwidth network and that the communication times between components are adequately small, because no constraints can be specified to the links. Using heuristics can suggest better performance when finding the deployment, unfortunately the found deployment may not be the best one.

The main purpose, why the automated deployment planning should be implemented into SOFA 2 environment, is to simplify the process of launching SOFA 2 applications, which are specified by assembly descriptors. The assembly descriptor

has strictly defined set of components that has to be deployed, thus such a generality of the first approach is not needed. The planning process should be also fast as the user usually does not want to wait for a long time before the application is actually launched. It is also assumed that the SOFA 2 environment will mainly run in private local high bandwidth networks. Thus the third approach, the heuristic based approach, has been chosen to implement the automated deployment planning in SOFA 2.

3.4 Proposed algorithm

Proposed solution of the deployment planning algorithm exploits the heuristic based algorithm (see Section 3.2.3). This algorithm does not cover all requirements specified in Section 3.1 and thus it has been extended in the following ways:

- added support for hierarchical software components
- added support for *local-communication* feature at interface specification

The original algorithm is intended for flat component systems, as it does not differentiate between primitive and composite components. In SOFA 2 a composite component does not contain any 'business' code, it only delegates method calls from its interfaces to interfaces of its sub-components. This delegation will consume more resources if the sub-components are not located on the same dock as the composite component. Thus, it is convenient to decide deployment of a composite component when all deployments of its sub-components are known. If it is possible the composite component should be deployed on a deployment dock, where some of its sub-components are deployed.

For the purpose of handling *local-communication* feature at interface specification new term *deployment unit* has been introduced. A *deployment unit* represents indivisible unit of deployment and it is represented by one of the following possibilities:

- a primitive component
- a composite component (without its sub-components)
- a group of primitive components that are interconnected via interfaces with *local-communication* feature together with a group of composite components that participate on delegation of method calls from those interfaces

In the original algorithm the description of a component's resource demands is quite simplified. Resource demands of a component are specified as 'per visit' of the component. But the component can be 'visited' via different methods on provided interfaces and each method can be called different number of times in different component applications; thus, the general 'per visit' resource demands are

hard to obtain. Therefore, the resource demands are specified for each method call from the component's provided interfaces to get more precise results (see Section 3.4.1). But for the purpose of deployment planning algorithm we need to know the total resource demands of each component per user request or per servicing a job. How these total resource demands are computed is described in Section 3.5.

Also the original algorithm does not cope with other component's requirements or machine's capabilities. It is assumed that each component can be launched on every machine and there are no upper bounds of available resources on the machines. But primitive components could require also some software requirements (e.g., a component may require some services to be available on a deployment dock); therefore, general specification of components' requirements and deployment docks' capabilities had to be included (see Section 3.4.2). A component can be deployed on a deployment dock only if the deployment dock has sufficient capabilities to serve component's requirements.

The process of automated deployment planning is shown in Algorithm 3. The algorithm assumes that each component is already described with total resource demands.

Algorithm 3: 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 5;
Deploy orderedDeplUnits using Algorithm 4;
Order compositeComponents according to a nesting level such that components with higher nesting level are put ahead;
Deploy compositeComponents using Algorithm 6;

First, the deployment units are created from all input components (function `GetDeploymentUnits()`). This function analyzes interconnections between components and according to the attribute *local-communication* it creates a set of deployment units. Requirements and resource demands of a deployment unit are deduced from requirements of all components that the deployment unit represents.

Further, it is necessary to divide the deployment units into two groups: the first group will contain deployment units representing only composite components and the second group that will contain the rest of deployment units, meaning the deployment units that represent only primitive components or group of components. This is done by the function `ExtractCompositeComponents()`; the first group is stored into compositeComponents list, the second group remains in deplUnits structure.

Next step covers deploying of deployment units in deplUnits. Firstly, the order of the deployment units is selected by Algorithm 5 and secondly the deployment units

are deployed in the given order using Algorithm 4.

At the end the composite components in compositeComponents list are ordered according to a nesting level such that the components with higher nesting levels are put ahead and then they are deployed using Algorithm 6. The order of composite components ensures that when a composite component is selected to be deployed all of its sub-components have been already deployed.

An expected result of the Algorithm 3 is that all components are deployed (in case that no error occurs).

Algorithms 4 and 5 are directly derived from Algorithms 1 and 2. Instead of deploying components on machines the deployment units are deployed on deployment docks.

Algorithm 4: 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 *Deployment dock* D_j **do**
    Mock-deploy the deployment unit U_i on the dock D_j;
    Note the value of $make_D$;
    Cancel the mock-deployment;
  end
  Choose the deployment dock D with the minimum value of $make_D$;
  Deploy the deployment unit U_i on the dock D and update the system;
end

It should be noted that deployment of a deployment unit on a dock may fail due to insufficient capabilities of the deployment dock. In such case no order of deployment units may be found in Algorithm 5 even if a valid order may exists.

Finally the Algorithm 6 describes how the composite components should be deployed. It is assumed that when the composite component is going to be deployed all of its sub-components have already been deployed. In the first step a set of deployment docks where the sub-components should be deployed is obtained. We are particularly interested in deployment docks of such sub-components that use delegation or subsumption of method calls with the composite component. Then the composite component is tried to be deployed on each deployment dock from the set until the deployment is successful. If the composite component fails to be deployed on any of the docks from the set then Algorithm 4 is used to deploy it.

3.4.1 Resource demands of a component

For the purpose of deployment planning it is important to know how long will the execution of component's code last. For that reason every primitive component has to

Algorithm 5: 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 4;
        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 6: 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**
    deplDocks $\leftarrow$ Get set of deployment docks where sub-components of C_i should be deployed;
    foreach *Deployment dock* D_j *in* deplDocks **do**
        Deploy the composite component C_i on the dock 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 4;
    end
end

be described with resource demands – how much resources does the component consume during its execution. There are more possibilities where the resource demands may be specified:

- at a component as a whole
- at each method from provided interfaces of a component

Assignment of resource demands directly to a component is not desirable because it would be necessary to make assumptions on how the component would be used. That is not known if the component is intended for general use in more applications. Instead, every part of the component which causes resource consumptions should be described. There are two types of primitive components that have to be considered:

active component A thread is running inside the component which may constantly consume resources and may call methods from required interfaces of the component.

passive component This component itself does not consume any resources. Resources are consumed only when a method from provided interfaces is called. As a reaction to a method call other methods from required interfaces may be called.

That is why the resource demands should be specified to every method in component's provided interfaces and to every running thread in the component.

The question is how the resource demands should be specified. In [2] they concentrate on performance prediction of component based applications. For this purpose they use detailed specification of components and their behavior. To each method in provided interface (it is called 'provided service') they add so-called *ServiceEffectSpecification* which describes how the provided services call methods from the required interfaces. This is an abstraction of the control flow through the components. For the purpose of performance analysis they extended the *ServiceEffectSpecification* into *ResourceDemandingSEFF*. This specification contains resource usage, transition probabilities, loop iteration numbers and also parameter dependencies to allow accurate performance predictions. *ResourceDemandingSEFF* contains *ResourceDemandingActions* that can place loads on the resources, which the component is using (e.g., CPU, hard disk, network connection, etc.). Demands can be specified as distribution functions, their unit has to be specified (e.g., CPU operations, hard disk accesses, etc.). In fact they do not describe the provided service as a whole, they add resource demands description to each action in the service. Units of the resource demands are specified generally. When the actual hardware is known, on which the component is going to be deployed, the time for executing the service can be computed. For example if the general unit is 'CPU operation' then the time for executing the service can be computed if the execution time for a single CPU operation is specified.

Also in [2] they distinguish active and passive resources. Active resources process jobs on their own (e.g., a CPU processes jobs or a hard disk processes read/write requests). On the other hand the passive resources do not process jobs on their own, but their possession is important as they are limited (e.g., a component needs a database connection). They also support parametric dependencies in resource demand specification, because the resource demand may vary depending on input parameters (e.g., uploading larger files with a component service produces a higher demand on hard disk and network).

In this work we differentiate between active and passive resources – demands on active resources are specified by 'resource demands'; demands on passive resources are specified by 'component's requirements' (see Section 3.4.2).

Resource demands are specified per whole methods/threads, therefore some average values have to be used. Because of the fact that the resource demands may depend on input parameters of a method or on a parametric setting of a component (e.g., an average size of an input parameter can be set via component's parameter), the value of the resource demand may be specified as an expression containing component's parameters.

The main purpose of the resource demands is to specify how long will the execution of the component's code last. Thus, it may be intuitive to specify these values in time units directly. However, this approach would not work correctly in the scenario where different machines with various hardware would be used in the system because the execution time of each component may vary depending on the machine where the component is deployed. Thus, it is convenient to specify the resource demands using general units.

There are more possibilities which units may be used for the resource demand specification. However, then the deployment docks would have to transform the generally specified resource demands into time values. Therefore, proper transformation rules have to be defined for each unit.

Possible specifications of CPU demands are shown in Table 3.1. The only difference between all the specifications is in the accuracy of the resulting time values. For example, proper characterization of CPU demands is more complicated due to different hardware platforms where the component may run. Performance of modern CPUs does not depend only on the clock frequency but on the entire architecture of the CPU. Although it may be easier to specify CPU demands of a block of code as a number of CPU operations, it is not as easy to specify how many of CPU operations can a CPU process per second. The time for processing one CPU operation has to be specified as an average time of processing all the CPU operation types. Big difference is for example in processing a floating point operation and an integer operation. Therefore, performance of modern computers are characterized by FLOPS[1] and MIPS[2]. For that reason better results can be achieved when more

[1]FLoating point OPerations per Second

[2]Million Instructions Per Second

detailed characteristics of resource demands are used.

	Specification of resource demands	Deployment dock's specification
I	#CPU operations	#CPU operations per second
II	#floating point operations & #integer operations	FLOPS & MIPS
III	an average time on a testing system	performance ratio to the testing system
IV	an average time on a testing system & parameters of the testing system (e.g., CPU benchmark points)	parameters of the target system (e.g., CPU benchmark points)

Table 3.1: Possible specifications of CPU demands with needed specifications of deployment docks in order to get resulting execution times.

3.4.2 Component's requirements and deployment dock's capabilities

Section 3.4.1 describes demands on active resources. These demands have been specified at each method from component's provided interface. The character of the demands is that if a method with resource demands is called more times, then also more resources will be consumed by the component.

However, components may require also static resources or available services on deployment docks in order to be deployed on such deployment docks. This is achieved by component's requirements – they are specified to a component as a whole, not to each method as it is at resource demands. All requirements have to be met by deployment dock's capabilities at time when a component is being deployed on the deployment dock. The component has to require static resources for the worst-case scenario as it cannot require more resources during runtime. All allocated resources are released when the component finishes its execution and it is deallocated from the deployment dock.

Specification of requirements and capabilities is done according to the OMG Deployment and Configuration (D&C) specification [17]. Every requirement and capability have the following basic attributes: name, value and kind. The kind classifies the requirement/capability and has implications on the type of the value and the rules how requirements are matched against capabilities. According to the OMG D&C specification following kinds of requirements/capabilities are supported:

attribute The value of the property is of a type that supports equality comparison. To match the requirement, the value of the requirement has to be equal to

the value of the capability. Example: OS type, library version, java runtime environment version, etc.

maximum The property describes a capability with an upper bound. The value of the property is of a type that supports ordering. To match the requirement, the value of the requirement has to be equal or lesser than the value of the capability. Example: CPU speed – e.g., the capability has 700MHz, and there is a requirement on at least 500MHz.

minimum The property describes a capability with a lower bound. The value of the property is of a type that supports ordering. To match the requirement, the value of the requirement has to equal or exceed the value of the capability. Example: latency – e.g., the resource can guarantee 30ms latency, and there is a requirement at least 40ms.

capacity This property has a certain capacity that can be consumed. The value of the property is of a numerical type. The value of the requirement is subtracted from the value of the capability. To match the requirement, the capability has to have a value that equals or exceeds the value of the requirement. Example: memory size, number of database connections, etc.

In order to deploy a component on a deployment dock, all the component's requirements have to be satisfied by the dock's capabilities:

- for each requirement there has to exist a capability such that the name and the kind of the capability match the name and the kind of the requirement
- the value of the requirement has to match against the value of the capability according to the rules that depend on the kind of the capability

Because of the fact that also component's requirements may depend on component's parameters, the values of requirements may be specified as expressions that contains component's parameters. Therefore, the total requirements have to be computed at the start of the deployment planning (e.g., maximal memory for the component, maximal number of connections, etc.).

3.5 Total resource demands of components

Total resource demands of a primitive component depend on a component usage in the application. If a component is used in one application more often then in the other, it consumes more resources and therefore more resources have to be allocated in advance.

Although SOFA 2 supports various communication styles that can be specified at connectors (e.g., *method invocation*, *message passing*, *streaming*, *shared memory*), only *method invocation* is used for counting *total resource demands* of components.

This communication style has strict rules as to how interfaces may be bound together – only provided/required interfaces may be bound together, not required/required or provided/provided as it is possible in other communication styles. In addition, method invocation represents synchronous execution of code. When a method from required interface is called, execution of current component is blocked and the component waits for the result of the method call. The transfer of control between components is obvious and that simplifies the process of counting total resource demands.

Each component (in fact each method from component's provided interfaces) is already specified by resource demands (see Section 3.4.1). In order to compute total resource demands of a component we need to know how many times is each method from provided interfaces called per processing a job or a user request. Then the total resource demands are computed as a sum of methods' resource demands multiplied by the total methods' call counts.

In order to count all method calls, the application may be modeled by *discrete-time Markov chains* (see Section 3.5.1) or following algorithm may be used.

The algorithm assumes that the behavior of the application is known. Particularly we need to know the behavior of each primitive component – which methods from required interfaces are called and how many times they are called as a result of processing an incoming call through a provided interface. If the component is active then also every thread of the component has to be specified by a list of all methods from required interfaces that the thread calls. This list serves as an entry point to the calculation of all method calls. Details are shown in Algorithm 7; these terms are used in the algorithm description:

ComponentMethod uniquely represents one method from a component's provided interface among all components in the application. It has an attribute *totalCallCount* which denotes how many times is the method called per processing a job. Because of the fact that the algorithm counts method calls step by step, it also has an auxiliary attribute *callCount* which represents a number of yet unprocessed method calls by the algorithm (in the end of the algorithm this attribute should be zero at all ComponentMethods).
ComponentMethod is associated with a list of MethodCalls that are called in response to the call of the ComponentMethod.

MethodCall represents one method from a component's required interface. It has an attribute *callCount* which denotes how many times is the method called per processing an incoming method call.

First, function `ProcessMethodCall(methodCall, callCount)` finds mapping from the methodCall to a corresponding ComponentMethod. This mapping depends on actual interconnection of primitive components. Then the attribute *callCount* of the ComponentMethod is increased by the value of callCount. Finally the ComponentMethod is inserted into the queue if it is not there yet.

Algorithm 7: Counting ComponentMethod calls per processing a job.

Input: Set of all primitive components
Output: Set of ComponentMethods together with their total call counts

```
componentMethodSet ← Get a set of ComponentMethods from all primitive
components and initialize their callCount and totalCallCount attributes to 0;
queue ← InitQueueOfUnprocessedComponentMethods();
while queue is not empty do
    componentMethod ← Get a ComponentMethod from queue;
    methodCallCount ← componentMethod.callCount;
    Reset value of componentMethod.callCount;
    Add methodCallCount to totalCallCount attribute of componentMethod;
    methodCallList ← Get list of MethodCalls for componentMethod;
    foreach methodCall in methodCallList do
        callCount ← methodCall.callCount * methodCallCount;
        ProcessMethodCall(methodCall, callCount);
    end
end
Output componentMethodSet together with their total call counts;
```

Function `InitQueueOfUnprocessedComponentMethods()` iterates through all active primitive components. For each component it gets a list of threads that are specified at the component and for each thread it gets a list of MethodCalls that the thread calls. Each MethodCall in the list is finally processed with the function `ProcessMethodCall()`; therefore, the queue is at the end initialized.

However, we need to know the behavior of each primitive component. Behavior protocol (see Section 2.4) describes activity on component's interfaces, but it does not contain information about method call counts. To overcome this issue an extension of behavior protocol specification has been proposed (see Section 4.3). This extension adds probabilities to alternatives and average number of cycles to repetitions. Therefore, it is possible to get number of method calls per processing an incoming call through a provided interface.

If the behavior protocols for primitive components are not specified then all the needed information may be received from an automated observation of the application behavior (see Section 3.5.2).

3.5.1 Discrete-time Markov chains

A *discrete-time Markov chain* (DTMC) [23] is a stochastic process with discrete state space (finite or countably infinite) and discrete parameter space whose dynamic behavior is such that probability distributions for its future development depend only on the present state and not on how the process arrived in that state. A DTMC

is characterized by its states and a transition probability matrix $P = [p_{ij}]$ which contains transition probabilities among the states.

In [20] they use DTMC to model the software architecture in order to get visit counts of all components per processing a job. They model each component by a single state or a set of states such that each state represents a software component in execution at any point in time. Transitions between states represent transfer of control between components. Appropriate probabilities have to be assigned to transitions according to the behavior of the system.

A state i is said to be *transient* if and only if there is a positive probability that the process will not return to this state. A state i is said to be an *absorbing* state if and only if $p_{ii} = 1$. That means that once the process enters such a state, it remains there forever. Reaching an absorbing state indicates the successful completion of a job.

The DTMC can be used to calculate the number of visits to each of the states. Let X_{ji} be the number of visits to the state i, starting at j. Then according to [23] it can be shown that

$$E[X_{ji}] = \sum_{n=0}^{\infty} p_{ji}(n),$$

where $p_{ji}(n)$ is the probability that the process will move from state j to state i in exactly n steps. It follows that if the state i is a transient state, then $\sum_{n=0}^{\infty} p_{ji}(n)$ is finite for all j; hence $p_{ji}(n)$ approaches 0 as n approaches infinity.

As it is shown, the expected total number of visits of a state per job can be calculated using the transition probability matrix of the DTMC. Therefore, if each provided method of each primitive component would be modeled by a single state or a set of states, then it would be possible to count total method calls per processing a job.

3.5.2 Automated observation of an application behavior

In order to count total resource demands of primitive components, behavior of the application has to be known. This can be ensured by specifying behavior protocols to all primitive components in the application. However, if the behavior protocols are not specified, then the behavior of the application can be observed automatically.

For this purpose we need to track each running thread in the application and log which methods of which components it calls. The idea is that the application may be run in a so-called 'logging mode' when all important events on components interfaces would be logged. It is necessary to use the application in this mode as if it would be used normally because the usage of the application may influence obtained results. After this stage the log would be processed and general behavior of each component would be found out.

SOFA 2 supports advanced management of components' control functionality via component aspects [16]. Aspects are specified to a deployment plan and they can extend the control part of every component in the application. Therefore, it is

possible to create a logging aspect that would log activities on components' interfaces without need of modifying 'business' code of the components. The goal is to observe an average behavior of each primitive component – which methods from required interfaces are called and how many times on average they are called per processing an incoming call through a provided interface. In addition, it is convenient to detect active components and note down a list of methods they call. For this purpose we need to log these items:

- An identification of a component, an interface name and a method name, where the event occurred.
- Type of the event: *CALL* – the method is being called; *RETURN* – the method finished the call and it has just returned.
- Point of view of the event: *CALLER* – the event is observed from a component that initiated calling of the method; *CALLEE* – the event is observed from a component that is serving an incoming method call.
- An unique identification of a running thread among all deployment docks. This is necessary for distinguishing different method calls of different threads running in one component concurrently.
- Time when the event triggered.

The algorithm of the acquiring components' behavior is straightforward; the log is processed sequentially according to the time, when the event was triggered. Log entries are bound to specific components; how the statistics about the usage of each component are processed is shown in Algorithm 8.

When the whole log is processed, final statistics are examined and the behavior of each component is obtained. For every called method from component's provided interface we have a list of methods from required interfaces that are called in order to process the incoming call. Method call counts are taken in the ratio to one incoming call. Therefore, these methods' call counts are average values in general application usage.

In addition active threads of active components are detected. For each active thread the list of methods that the thread calls is obtained. An active thread usually has an infinite loop, where it generates new jobs or user requests that are performed by the application. It is not easy to detect which method calls are part of one job. Therefore, there are more possibilities how to handle call counts of the methods in the list. Either each method call from the active thread may be regarded to be the start of one job; then each method in the list would have call count equal to one. Alternatively, the whole bunch of methods can be considered to be part of one job; then the call count of each method should be set proportionally to the whole number of observed method calls of the thread.

Algorithm 8: Processing one log entry corresponding to one component.

Input: A log entry corresponding to one component
Result: Statistics of the component are updated

```
if is CALL event then
    if is CALLEE event view then
        // thread is entering the component via provided method
        Increase call count of the method;
        Mark that the thread entered the component via this method;
    else
        // CALLER event view
        if the thread entered the component earlier via some method then
            M ← Get the method through which the thread entered the
            component;
            Add method call to M;
        else
            // thread is probably in an active component
            Add method call to component's active thread;
        end
    end
else // RETURN event
    if is CALLEE event view then
        Mark that the thread is leaving the component that previously entered
        via this method;
    end
end
```

Problems may arise when a component would use new threads to handle an incoming method call. Then the detection of the fact that some methods are called in response to an incoming call fails, because the algorithm would mistakenly mark the component as active. It would consequently assume that the method calls are initiated by the component's own thread. If a new 'working' thread is created during processing of the incoming method call and this thread ends its execution before the end of processing the incoming call, that means that the 'working' thread will never call any methods from the required interfaces after the incoming call is served. Further, it would be possible automatically detect these threads and assign their method calls correctly to the list of method calls associated with the provided method. This would be possible because in SOFA 2 the thread ID of the newly created thread is deduced from the thread ID of its parent; therefore, we would know that the thread which entered the component via a provided method created some 'worker' threads to perform the action.

However, a more significant issue is that the component may use a thread pool instead of creating new 'working' threads all the time. That would be hard to detect by processing the log, because the threads in the thread pool are created only once and are reused later. Thus, behavior of such components is better specified manually.

Chapter 4

SOFA 2 extensions

Previous chapter covered the problem of automated deployment planning and its possible solution. This chapter describes the implementation issues of the automated deployment planning in the SOFA 2 environment.

Sections 4.1 and 4.2 describe extensions of the component and deployment dock descriptions. Section 4.3 introduces a behavior protocol extension. Section 4.4 describes implementation of the deployment planning algorithm; Section 4.6 covers implemented tools and finally Section 4.7 describes the Cloud Gate application.

4.1 Component description

As mentioned in Section 2.3, a SOFA 2 component is defined by a *frame* which denotes the type of the component, and it is implemented by an *architecture*. The *component frame* and the *component architecture* are represented by *Frame* and *Architecture* meta-classes in the meta-model. The deployment planning algorithm assumes that each primitive component is specified by resource demands and also that each primitive component has a behavior specification. Therefore, the repository meta-model had to be extended appropriately to include new information.

4.1.1 Frame meta-class

A *component frame* serves as a black-box view of the component. It is defined by a *Frame* meta-class in the repository meta-model. This meta-class has to contain information about component behavior – specifically which methods from required interfaces are called and how many times they are called as a result of processing an incoming call through a provided interface, or which methods from required interfaces are called from active threads of the component. This information could be get from the *behavior* attribute of the *Frame*, especially if the extended form of behavior protocol is used (see Section 4.3). However, it is convenient to have this information explicitly specified in the *Frame* meta-class, because then the behavior protocol does not need to be parsed every time when the information is needed, or because the

behavior protocol is not specified at all and the information is obtained automatically (see Section 3.5.2). Therefore, following attributes have been added to the *Frame* meta-class:

`methodDescription` contains a list of method descriptions specified by *MethodDescription* meta-class. Every method from component's provided interfaces is specified by the *MethodDescription* meta-class and it is in this list.

`activeMethodDescription` contains a list of active method descriptions specified by *MethodDescription* meta-class. Every component's active thread that calls any methods from required interfaces is specified by the *MethodDescription* meta-class and it is in this list.

The extension of the *Frame* meta-class together with auxiliary meta-classes is shown in Figure 4.1.

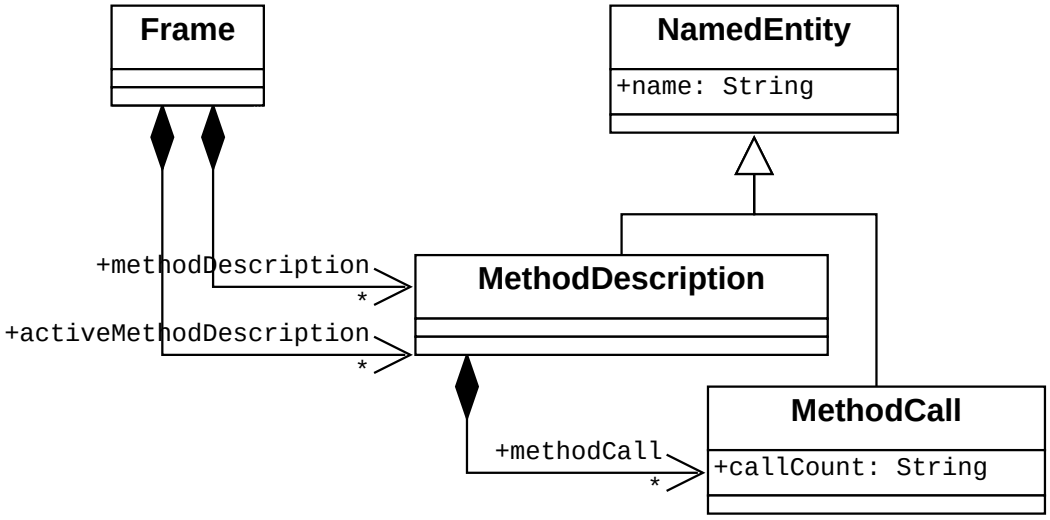


Figure 4.1: Frame meta-class extension

The new frame's attributes do not have to be specified manually; instead, they may be filled to the repository via the new tool *sofa-update-frames* (see Section 4.6.3).

MethodDescription meta-class

A *MethodDescription* meta-class is an extension of the *NamedEntity* meta-class; therefore, it has a `name` attribute. This attribute contains either the name of a component's active thread (e.g., *thread_1*), or it contains the name of a method from component's provided interface together with the name of the provided interface separated by dot (e.g., *provInterfaceA.methodA*). Further, it contains the following attribute:

`methodCall` This attribute contains a list of all methods from required interfaces that are called in response to the processing of the provided method call or as a result of processing one job by the active thread. These method calls are specified by *MethodCall* meta-class.

MethodCall meta-class

A *MethodCall* meta-class is an extension of the *NamedEntity* meta-class; therefore, it has a `name` attribute. This attribute contains the name of the method from a component's required interface together with the name of the required interface separated by dot (e.g., *reqInterfaceA.methodA*). Further, it contains the following attribute:

`callCount` This attribute is of type String; it determines the average number of times the method is called. If the method call counts are get from processing a behavior protocol (particularly the extended form of the behavior protocol, see Section 4.3), then this value may depend on the *Properties* defined at the *Frame*. Therefore, it should be possible to specify this value as an expression containing names of *Properties* defined at the *Frame*.

4.1.2 Architecture meta-class

A *component architecture* serves as a gray-box view of the component. It represents an implementation of the component and it is defined by an *Architecture* meta-class in the repository meta-model. This meta-class has to contain information about resource demands of all methods that the component provides. Also, resource demands can be specified to all threads of the component in case the component is active. Due to this reasons these attributes have been added to the *Architecture* meta-class:

`active` This attribute is of type Boolean. Its value is *true* if the component is active; *false* otherwise. Default value is *false*.

`resourceDemandDescription` This attribute contains a list of resource demand descriptions that are specified by *ResourceDemandDescription* meta-class. These descriptions are referenced from specifications of resource demands that are associated to *MethodResourceDemand* meta-class. The idea is that components may be developed by different developers or even by different companies and everyone of them may use different specification of resource demands; therefore, these resource demands descriptions are specified directly to the *Architecture*.

`methodResourceDemand` This attribute contains a list of method resource demands specified by *MethodResourceDemand* meta-class. Every method from component's provided interfaces that consumes some resources when the method is called is specified by the *MethodResourceDemand* meta-class and it is in this list.

`activeMethodResourceDemand` This attribute contains a list of method resource demands specified by *MethodResourceDemand* meta-class. Every component's active thread that consumes some resources during its execution is specified by the *MethodResourceDemand* meta-class and it is in this list.

All these attributes have proper meaning only for primitive components. Composite components do not have any 'business' code; therefore, they cannot be active as they cannot have any active threads. Further, it is not needed to specify resource demands to methods from composite components' provided interfaces as all method calls are delegated to their subcomponents.

The extension of the *Architecture* meta-class together with auxiliary meta-classes is shown in Figure 4.2.

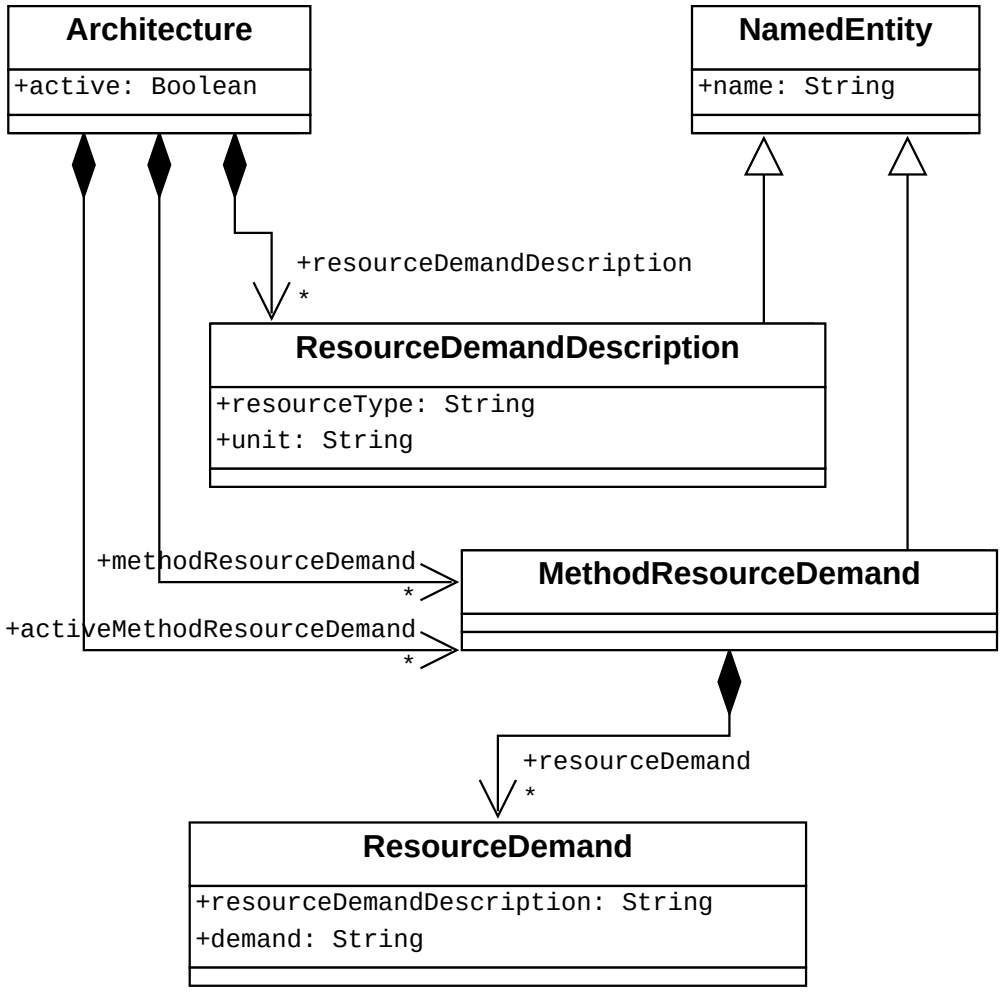


Figure 4.2: Architecture meta-class extension

The `active` attribute can be specified via *adl.xml* file of the architecture. The rest of the new attributes are specified via *method-info.xml* file associated with the

architecture. This file may be generated using *cushion* tool with the action *method-info* (see Section 4.6.1).

ResourceDemandDescription meta-class

A *ResourceDemandDescription* meta-class is an extension of the *NamedEntity* meta-class; therefore, it has a `name` attribute which uniquely identifies resource demand descriptions that are specified at one *Architecture*. Further, it contains the following attributes:

`resourceType` This attribute is of type String; its value determines the type of the resource (e.g., *CPU*, *disk*).

`unit` This attribute is also of type String; its value specifies the units in which the resource demands are counted (e.g., *operations* for *CPU* resource type, *bytes* for *disk* resource type).

MethodResourceDemand meta-class

A *MethodResourceDemand* meta-class is an extension of the *NamedEntity* meta-class; therefore, it has a `name` attribute. This attribute contains either the name of a component's active thread (e.g., *thread_1*), or it contains the name of a method from component's provided interface together with the name of the provided interface and with the name of the frame, where the provided interface is defined. An architecture may theoretically implement more than one frame; thus, the triplet frame name, provided interface name and method name uniquely identifies the provided method. The names are separated by dots (e.g., *frameA.provInterfaceA.methodA*). The name of the frame may contain dots, but the name of the interface and the method's name cannot. Therefore, any original name is retrievable from the `name` attribute. Another attribute of the *MethodResourceDemand* meta-class is:

`resourceDemand` This attribute contains a list of resource demands specified by *ResourceDemand* meta-class. Resource demands are associated with a particular resource type (e.g., *CPU*, *disk*). The same resource demands (corresponding to one resource type) may be specified several times using different units; therefore, more equivalent specifications of the resource demands can be in the list.

ResourceDemand meta-class

A *ResourceDemand* meta-class contains following attributes:

`resourceDemandDescription` This attribute is of type String and it contains the name of the *ResourceDemandDescription*, which has to be defined in the *Architecture*. Thus, this attribute determines the type of the resource and the units in which the `demand` attribute is specified.

`demand` This attribute is also of type String; it specifies the average amount of the resource demand. According to the proposed solution described in Section 3.4.1 it should be possible to specify this value as an expression containing names of *Properties* defined at the *Architecture*.

4.2 Deployment dock description

The deployment planning algorithm (as it is proposed in Section 3.4) introduces new requirements on deployment docks:

- A component may be deployed on a deployment dock only if the dock has sufficient capabilities that satisfy the component's requirements. This requires a management of deployment dock's capabilities (see Section 4.2.1).
- A so-called *dock makespan* is needed from the dock[1]. This is the value of the most loaded resource on the dock; it is either the dock's total CPU execution time or disk I/O time, whichever is greater. In order to get the *dock makespan*, the deployment dock has to keep track of the total execution times associated with the resources. These values are represented by *resource usages* (see Section 4.2.2).
- Component's resource demands are specified using general units but the management of *resource usages* on the dock requires time units. Therefore, the component's resource demands have to be converted into time values on the dock (see Section 4.2.3).

A deployment dock is remotely accessible via the `DeploymentDock` interface. Therefore, if any new functionality is needed to be remotely accessible on the dock then this interface have to be extended.

4.2.1 Management of deployment dock's capabilities

A capability on a deployment dock is defined by the class `Capability`. It has attributes `name`, `value` and `kind`. Meaning of these attributes are the same as it is described in the Section 3.4.2. The `value` attribute is of type String; according to the proposed solution the type of the `value` attribute should depend on the capability kind. Therefore, the value of the capability is converted into appropriate type according to the capability kind when it is needed.

In order to simplify the process of matching component's requirements against the dock's capabilities, the `CapabilityManagementHelper` class has been created. The class has two methods:

[1] In the deployment planning algorithm it is marked as $make_D$, see Algorithm 4.

`allocate()` This method matches component's requirements against dock's capabilities according to the rules specified in the Section 3.4.2. If there are any requirements with *capacity* kind then proper capabilities are modified (their values are diminished by the values of requirements). The method returns a list of *capability allocations*. `CapabilityAllocation` class has two attributes: `name`, which denotes the capability name and `value`, which represents the allocated value. If the matching fails, an exception is thrown.

`revertAllocation()` This method takes a list of *capability allocations* as an argument and reverts the values of previously allocated capabilities.

Dock's capabilities are accessible via the `DeploymentDock` interface through the method `getCapabilities()`. The method returns a list of capabilities with values corresponding to the time of the method call.

Capabilities are initialized during the dock startup. New capabilities may be specified via system property *sofa.dock.capabilities*. The value of this property contains a list of capabilities (they are separated by colon) and each capability may be specified using this pattern: `<name>|<value>|<kind>`.

4.2.2 Management of resource usages

The *resource usage* represents an aggregated average value of the components execution times associated with the resource (e.g., it may represent either CPU execution time for the *CPU* resource, or disk I/O time for the *disk* resource). It is counted as a sum of *resource usages* that are get from deployed components on the dock (see Section 4.2.3 how to get appropriate *resource usages* for a component). Therefore, the value of the *resource usage* is zero when no component is deployed on the dock, and it is increased when new components are deployed on the dock.

The deployment planning algorithm chooses a deployment dock which is the least utilized. From the algorithm point of view, the values of the *resource usages* may increase unlimitedly. But it is convenient to limit the upper bound of resource usages to prevent possible overloading of the whole system; thus, an upper bound of resource usages has to be specified.

There are two possibilities how the *resource usages* may be accessible on a deployment dock:

- via the `DeploymentDock` interface – the interface may be extended with a new method that would return a list of resource usages (e.g., a method named `getResourceUsages()`).
- via dock's capabilities – *resource usages* may be specified as capabilities with *capacity* kind; capabilities are already accessible via the `DeploymentDock` interface.

In the first approach, it would also be necessary to use additional capabilities that would specify the upper bounds of the resource usages. The second approach uses capabilities directly and the existing interface does not need to be extended.

The second approach has been chosen; therefore, the *resource usages* are managed as capabilities with *capacity* kind. In order to distinguish them from other dock's capabilities, they are named in the following way: `system.resource.<resource name>`. The initial value of the capability represents an upper bound of the aggregated component's execution time specified in microseconds. Thus, every deployment dock that runs on a machine with *CPU* and *disk* should have defined these capabilities: *system.resource.cpu* and *system.resource.disk*, with *capacity* kind.

Unfortunately, the problem is that the value of a *resource usage* may be get from the capability only if the original value of the capability is known. Therefore, the `Capability` class has been extended with an attribute `originalValue` of type String, which keeps the original value of the capability. Thus, the value of a *resource usage* can be computed.

4.2.3 Conversion of component's resource demands

Component's resource demands are specified in general units (see Section 3.4.1). But deployment docks need to know the processing time of each component, specifically of each resource that the component uses. Therefore, the demands that are specified in general units have to be converted into time values.

Total execution time of a component depends on the hardware of the machine where the deployment dock runs. For that reason, the deployment dock has to be specified by parameters that will be used in the conversion. Examples of such parameters were shown in Table 3.1.

In order to perform deployment planning it is suitable to count time values remotely by a deployment planner (see Section 4.4). Therefore, the deployment dock has to provide means for converting components' resource demands specified in general units into time values. There are more possibilities what the dock may provide:

- parameters that are needed to perform the conversion. This is not a flexible solution. There may be a lot of parameters (different parameters for different general units specifications) and the planner would have to know all the conversion rules to get correct results.

- a method in the `DeploymentDock` interface that will do the conversion. This is a much better approach as the conversion is performed by the dock itself; the planner does not need to care about the conversion rules. This approach may be little inefficient for the deployment dock as there may be a lot of requests on the conversions and the deployment dock would have to serve them.

- an object – *converter* – that is responsible for the conversion. This solution eliminates disadvantages of the previous approach because the planner gets the

converter from the dock and then it can do the conversions by itself.

The third solution has been chosen. The `DeploymentDock` interface has been extended by a method `getResourceUsageConverter()` which returns the converter. The converter implements interface `ResourceUsageConverter`. This interface has only one method `convert()`, which takes as a parameter a collection of generally specified component's resource demands and returns a collection of environment-specific *resource usages* of type `ResourceUsage`. `ResourceUsage` has two attributes: `resourceType` of type *String* that contains the name of the resource (e.g., *CPU*, *disk*) and `timeConsumption` of type *long* that denotes the time consumption of the specified resource in microseconds. The resulting time consumptions depends on the settings of the converter; therefore, the values are usually different for converters from different deployment docks.

A simple resource usage converter has been created; it is represented by the `ResourceUsageConverterImpl` class. This converter has built in conversion rules that supports *cpu* and *disk* resource types. The *cpu* resource demands may be specified using *operations* units, and the *disk* resource demands using *bytes* units. To be able to use the converter, following parameters have to be specified first:

cpu operations per second A number of abstract CPU operations that the machine (where the deployment dock is running) can perform per second.

disk megabytes per second A number of megabytes that the machine (where the deployment dock is running) can read from/write to a disk per second.

The converter is initialized during the deployment dock startup. Necessary parameters are acquired from system properties: *machine.property.cpu-operations-per-second* and *machine.property.disk-megabytes-per-second*. Both parameters may be specified as floating point values.

It is important to consider how many deployment docks will run on one machine and set the converters' parameters appropriately in order to get anticipated result.

4.3 Behavior protocol extension

Main goal of the behavior protocol extension is to get another relevant information out of the behavior specification. Currently we can get a list of methods that are called during acceptance of a method on a provided interface, and we can get a list of methods that an active components calls. However, we are interested also in method call counts for the purpose of counting total resource demands of components (see Section 3.5). This can be easily done by these extensions:

- adding probability to alternative
- adding mean value of repetitions to finite repetitions

Operator	Description
`A[`P_A`] + B[`P_B`]`	*alternative*; the set of traces which are generated either by `A` (with probability P_A) or by `B` (with probability P_B), $P_A + P_B = 1$
`A*[`r`]`	*repetition*; equivalent to `NULL + A + (A;A) + (A;A;A) + ...` where `A` is repeated any finite number of times, but in average r times

Table 4.1: Extended operators for constructing behavior protocols. `A` and `B` stands for a protocol, P_A and P_B are probabilities; therefore, $P_A, P_B \in [0, 1]$.

Extended operators are shown in Table 4.1.

To provide backward compatibility with the original behavior protocol specification, probabilities at alternatives and mean values of repetitions can be omitted. The notation of the operators would be the same as in the original behavior protocol (see Table 2.1), but the meaning would be extended in the following way:

- alternatives have equal probabilities (e.g., if there is a protocol `A + B + C`, then it is perceived like `A[`P_A`] + B[`P_B`] + C[`P_C`]`, where $P_A + P_B + P_C = 1$ and $P_A = P_B = P_C$)
- mean value of repetitions is equal to 1 (e.g., if there is a protocol `A*`, then it is perceived like `A*[1]`)

Moreover, it is convenient to specify only some probabilities to alternatives. Then the unspecified probabilities of all alternatives have equal value (e.g., if there is a protocol `A + B[`P_B`] + C`, then it is perceived like `A[`P_A`] + B[`P_B`] + C[`P_C`]`, where $P_A + P_B + P_C = 1$ and $P_A = P_C$).

Behavior protocols are specified to frames; therefore, they should be general enough. Sometimes it could be inconvenient to specify precise values of probabilities or precise numbers of repetitions. Thus, it would be suitable to specify these values as expressions with frame's parameters.

4.4 Deployment planning

The process of preparing a deployment plan of a SOFA 2 application manually consists of the following steps:

1. User uses the *cushion* tool with the *deplplan* action; he/she specifies an assembly to which he/she would like to generate the deployment plan. This action creates a new deployment descriptor in the repository and it generates an ADL file for the deployment plan.

2. User edits the generated ADL file – he/she fills in the names of deployment docks where each component should be deployed and he/she fills in values of components' properties.

3. User runs *cushion* with the action *deploy*. This action is implemented by the `Deploy` class in *sofa-tools-api*. It parses the ADL file and updates the deployment plan in the repository – it fills in the names of the deployment docks and the values of properties. At the end it calls the `deploy()` method on the deployment plan. This runs a process of generating connectors between components. If it succeeds then the deployment plan is ready to use and the application can be launched according to it.

The `Deploy` class has been extended such that it counts *total resource demands* for each primitive component and stores the results into the deployment plan. It is essential to have this information in the deployment plan because components' *total resource demands* are needed during the application launching.

First, the deployment plan is analyzed using the static method `processDeploymentPlan()` of the class `DeploymentInfo`. This method retrieves information about component interconnections and counts method calls of all primitive components according to the Algorithm 7. It returns a `DeploymentInfo` object that contains the method call counts. Second, the *total resource demands* of each primitive component are computed (see Section 3.5) and stored to the deployment plan.

Because of the fact that values of component's *environment assumptions*[2] may be specified as expressions containing names of component's properties, it is useful to count the values in this stage and to store them also to the deployment plan.

A deployment plan is in the meta-model represented by the `DeploymentPlan` meta-class. Each application's component is in the `DeploymentPlan` specified by the `InstanceDeploymentDescription` meta-class. This meta-class has the *resource* attribute that contains a list of instances of `ResourceDeploymentDescription` meta-class. The attribute is used for storing computed values of component's *total resource demands* and *environment assumptions* into the deployment plan.

The `ResourceDeploymentDescription` meta-class has following attributes:

`resourceName` contains a name of the component's `EnvironmentAssumption` or `ResourceDemandDescription`. In order to distinguish between the two possibilities, the name is specified in the following way: `ea:<name>` when the name corresponds to the `EnvironmentAssumption`; `rdd:<name>` when the name corresponds to the `ResourceDemandDescription`.

`resourceValue` contains a total value of the component's *environment assumption* or *resource demand*.

[2]Component's requirements are in the meta-model called *environment assumptions*.

Further, the `Deploy` class has been extended to support the automated deployment planning. The process of preparing a deployment plan using the automated deployment planning modifies following steps:

- User may omit the specification of deployment docks in the step 2.
- User runs *cushion* with the action *autodeploy* in the step 3.

The action *autodeploy* is also implemented by the `Deploy` class, but the deployment process is run with parameters indicating that the automated deployment planning should be used; therefore, the names of deployment docks do not need to be specified in the ADL file.

The automated deployment planning is done by the class `AutoDeployHelper`. The planning process runs on a local machine and it consists of the following steps:

1. A resource manager represented by the class `TargetManager` is initialized – it gets a list of all running deployment docks from the *dock registry* and from each dock it obtains a list of capabilities and a *resource usage converter*. A snapshot of all available resources among all running deployment docks is made.
2. The deployment planning is performed according to the Algorithm 3. All component deployments are simulated locally using the resource manager. Actually no component is physically deployed on any dock.
3. If the algorithm succeeds and a valid deployment of all components is found then the deployment plan is updated with the names of respective deployment docks.

Usage of the *autodeploy* action is described in Section 4.6.4.

The planning process is separated from the launching process of the application; deployment plans may be prepared in advance and stored in the repository without immediate launching of the applications. Therefore, there is no reservation of resources on the docks during the planning process. Thus, even if the deployment planning succeeds and the deployment plan is ready to use, launch of the application according to the plan may fail due to insufficient capabilities on a dock (e.g., another application that required the resources has been launched).

4.5 Application launch and termination

Introducing the component's *resource demands* and the management of deployment dock's capabilities require modifications of the launch and termination processes of SOFA 2 applications.

A SOFA 2 application is launched using the tool *sofa-launch*; a deployment plan is specified as a parameter. Application's components are instantiated on deployment docks according to the deployment plan. During the component instantiation the

component's requirements and *resource demands* are matched against the dock's capabilities using the `CapabilityManagementHelper` class (see Section 4.2.1). If the dock has sufficient capabilities then the needed resources are assigned to the component. If not, then the launching process fails and all acquired resources are released.

If the launching process succeeds then the application runs. The application can terminate itself or it may be terminated using the tool *sofa-shut*; an application ID is specified as a parameter. Then all components of the running application are stopped and they are released from docks. Further, all component's acquired resources are released.

4.6 Implemented tools

This section describes auxiliary tools that have been created to support the automated deployment planning in the SOFA 2 environment. It also describes the usage of the tools.

Some of the tools are accessible via new actions for the *cushion* command like tool. The *cushion* tool is used for development and management of SOFA 2 applications; how it works and how it is used is described in the SOFA 2 documentation [21].

Section 4.6.1 describes the tool that simplifies specification of *resource demands* to architectures. Sections 4.6.2 and 4.6.3 describe tools for observing and analyzing an application behavior and for updating the information in corresponding frames. Finally, Section 4.6.4 describes the tool for creating deployment plans using the automated deployment planning algorithm.

4.6.1 Cushion methodinfo

The *methodinfo* action of the *cushion* tool simplifies specification of *resource demands* to primitive components. It generates a *method-info.xml* file for a particular primitive architecture. The usage is following:

```
cushion methodinfo [architecture name]
```

If the name of the architecture is not specified then it generates *method-info* files for all primitive architectures in the workspace.

The action is implemented by the `PrepareMethodInfo` class in *sofa-tools-api*. It retrieves all provided interfaces from frames that the architecture implements. From each interface it obtains names of methods that are defined in the interface; this is done using the `CodeProcessor` interface. For each method it generates an instance of the `MethodResourceDemand` meta-class and inserts it into the `methodResourceDemand` attribute of the architecture.

If the architecture is marked as *active* then also one instance of the `MethodResourceDemand` meta-class is generated and inserted into the `activeMethodResourceDemand` attribute of the architecture.

Finally, the *method-info.xml* file is generated according to the data in the architecture.

When the *method-info* file is created then the *resource demands* of each method may be specified into it. The file is uploaded into the repository when the action *commit* of the tool *cushion* is called on the corresponding architecture.

4.6.2 Logging aspect

For the purpose of logging method calls the *MethodCallLog* aspect has been created. It is based on the example aspect *LogAspect* that monitors method calls between components and output the information to the console. It has been extended to output all the necessary information (see Section 3.5.2) into a file.

The *MethodCallLog* aspect uses the `InterceptorGenerator` class (a microcomponent generator) to create new microcomponents to each interface of each component in the application. The generated microcomponents notify method calls on the interfaces – they specify the name of the method that is being called or that has just returned. The notifications are received by the microcomponent `MLogTransceiver` which sends log messages to the `MLogger` microcomponent.

In Section 3.5.2 is described a list of items that are needed to be logged. All the items are accessible by the aspect except the name of the 'business' interface that the method belongs to. This was solved such that the `InterceptorGenerator` has been modified. It generates microcomponents that return the 'business' interface names together with the method names in the method call notifications (the interface name and the method name are separated by dot).

The microcomponent `MLogger` receives logging messages and writes them into a file. There is just one logger per Java virtual machine through which the log messages are written to the file. The name of the log file is set via the parameter *output-file* of the `MLogger`. Its initial value is set to `method-log-file_$dock-name$.tmp`; the string `$dock-name$` is replaced by the name of the dock during the aspect initialization. Therefore, if more docks run on one machine then more log files will be created.

In order to run an application in the 'logging' mode the *MethodCallLog* aspect has to be specified in the deployment plan. When the application is launched the log files are created and the communication between components is logged.

The obtained log files can be processed with the tool *sofa-update-frames* (see Section 4.6.3).

4.6.3 Tool sofa-update-frames

Previous section describes possibilities to log method calls between components. The result of that process are log files containing all method calls observed from components' interfaces.

The log files may be processed by the tool *sofa-update-frames*. It analyzes the log files and it updates corresponding frames in the repository. Usage of the tool is following:

```
sofa-update-frames.(bat|sh) <log-file> [log-file ...]
```

The tool is implemented by the `UpdateFrameStats` class. Log files are processed one by one as it is described in Section 3.5.2. For each frame that is identified in the log, statistics about method calls are held. When the log files are processed then all the identified frames are updated in the repository – content of the attributes `methodDescription` and `activeMethodDescription` is newly generated according to the observed statistics.

If more active threads are observed in one component then statistics from all threads are merged into one active thread. Although the component could have more active threads, this is just a prevention against possible usage of a thread pool by the component (see Section 3.5.2).

4.6.4 Cushion autodeploy

The *autodeploy* action of the *cushion* tool automatically assigns deployment docks to components, commits the deployment plan to the repository and prepares it to be used for the application launch. Usage of the action is following:

```
cushion autodeploy [online|offline] [unset|erroneous|all]   \
    <deployment plan name>
```

The first parameter specifies which values of capabilities from all running deployment docks are used in the planning process:

online Current values are used; this is default behavior.

offline Original values are used. This behavior simulates a state as if no application is running on any of the docks.

Some components in the deployment plan may have specified names of docks where they should be deployed. The second parameter determines how to cope with the components, whether they should be deployed on the specified docks or if the docks should be selected by the deployment planning algorithm. There are following options:

unset It assigns deployment docks to components with unspecified dock names. This is default behavior.

erroneous It assigns deployment docks to components with unspecified dock names and to all components where the deployment on the specified dock would cause an error (e.g., the dock is not available or it has insufficient capabilities).

all It assigns deployment docks to all components even if the components had the docks specified in the deployment plan.

4.7 Cloud Gate application

The *Cloud Gate* application is a new application that serves as an entry point to the 'cloud' represented by the SOFA 2 runtime environment. It simplifies access to the *SOFAnode* and enables easier management of SOFA 2 applications. Its main features are:

- possibility to run its own deployment dock
- launching SOFA 2 applications according to a deployment plan or even according to an assembly
- listing running SOFA 2 applications with option to stop them

In order to use the *Cloud Gate* application the *SOFAnode* has to be running. The application is launched using the following script:

```
sofa-cloud-gate.(bat|sh)
```

During the startup of the application the settings of the *Cloud Gate* application is loaded from the *cloudgate.properties* file located in the current directory. If the settings file does not exists, default settings is used.

After the startup a sofa icon emerges in the system tray. It makes accessible a menu through which the *Cloud Gate* application is operated. The menu consists of the following items:

Settings shows the *Settings* form that contains settings of the whole application. It has three sections:

- In *General settings* section it is possible to specify whether the deployment dock should be launched immediately after the *Cloud Gate* startup.
- *Deployment dock* section contains the name of the deployment dock and a list of capabilities that will be available on the dock. The capabilities may be edited or new capabilities may be added.
- *Machine properties* section contains description of the machine where the *Cloud Gate* application runs. The values specified in this section will be used for initialization of the deployment dock's *converter*.

Settings in the sections *Deployment dock* and *Machine properties* have to be set before the start of the deployment dock.

Run deployment dock starts the deployment dock with the settings specified on the *Settings* form. Output of the dock is written to the console.
The option is available only if the deployment dock is not running yet.

Stop deployment dock stops the deployment dock. The option is available only if the deployment dock is running.

Launch sofa application allows launching a SOFA 2 application according to a selected deployment plan. First, a dialog with a list of all available deployment plans in the repository is shown. The list may be filtered according to the substrings in the name. When the deployment plan is selected a dialog with all versions is shown in the case that more versions are available. The versions are sorted according to the creation time such that the newest version come first.

After the deployment plan with proper version is selected then the *Deployment preview* dialog is shown. It shows the tree structure of the application together with associated deployment docks to components. The names of the docks are highlighted – green color means that the dock is online; red color means that the dock is unavailable (see Figure 4.3). If all the deployment docks are online then the application may be launched.

Quick launch allows to launch a SOFA 2 application according to a selected assembly without a specific deployment plan. This is useful in the case that the user just needs to launch the application and he/she is not interested in the actual components deployment.

First, a dialog with a list of all available assemblies in the repository is shown. The list may be filtered according to the substrings in the name. After the selection of the assembly a dialog with all versions is show in the case that more versions are available. The versions are sorted according to the creation time such that the newest version come first.

When the assembly with a proper version is selected the process of automated generation of the deployment plan is launched (see Section 4.7.1). If the deployment plan generation succeeds then the *Deployment preview* dialog is shown. It shows the tree structure of the application together with associated deployment docks to components. The names of the docks are highlighted in green color (the automated deployment planning works only with docks that are online). Only if a dock becomes unavailable during the generation of the deployment plan then the dock's name is highlighted in red color. If all the docks are online then the application may be launched.

Although the menu item is named as 'quick launch', the 'quick' word refers to the fact that the deployment plan does not need to exists and the SOFA 2 application is launched directly using only the assembly. But the generation of the deployment plan itself may last longer due to the generation of connectors between components.

Running applications shows a dialog with all running SOFA 2 applications in the *SOFAnode*. The dialog allows to terminate the applications.

Console shows a *Console* form containing output of the *Cloud Gate* application. It contains the same output as the original console; it is useful when the *Cloud*

Gate application is run in the background.

Exit stops the deployment dock if it is running and closes the *Cloud Gate* application. It also saves the settings into the *cloudgate.properties* file in the current directory.

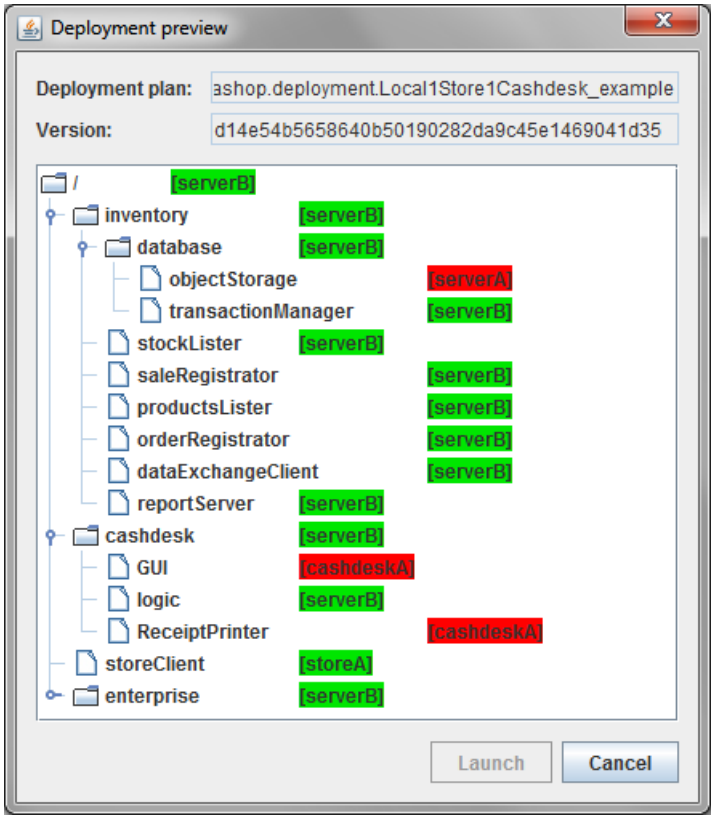


Figure 4.3: *Deployment preview* dialog. It shows a tree structure of the application with associated deployment docks. The names of the docks are highlighted – green color means that the dock is online; red color means that the dock is unavailable.

The *Cloud Gate* application is implemented by the `CloudGate` class in *sofa-j-cloudgate* meta-project. The deployment dock and SOFA 2 applications are launched in separate processes; this is done by the class `Launcher`.

4.7.1 Automated generation of a deployment plan

The possibility to launch a SOFA 2 application directly from a specified assembly is one of the main feature of the *Cloud Gate* application.

First, new deployment plan has to be created according to the assembly and stored into the repository. The name of the deployment plan is based on the assembly

name; it is set to: `<assembly name>.autodeplplan`. If a deployment plan with this name already exists then new version of the plan is generated.

The generated plan does not have set any components' properties; it is used directly in the automated deployment process. If the process succeeds then the deployment plan is ready to be used.

Chapter 5

Example SOFA 2 application

The following chapter describes the process how the automated deployment planning may be used with the implemented tools on an example application *SOFAShop*.

The *SOFAShop* [22] is a SOFA 2 application developed according to the use cases of the *CoCoME*[1] model [7]. It represents a simplified environment for managing an enterprise with shops and stores. The main goal of the *SOFAShop* is to model the complex architecture in the SOFA 2 environment.

The application is separated into three basic parts [22]:

shop is represented by *cashDesk* components. Each *cashDesk* consists of the following components: *GUI* (simple graphical user interface for managing sales), *logic* and *receiptPrinter*.

store is represented by an *inventory* component (store server) and *storeClient* component that provides simple GUI for managing orders and observing sales. The *inventory* consists of the following components: *database* (*objectStorage* and *transactionManager*), *stockLister*, *saleRegistrator*, *productLister*, *orderRegistrator*, *dataExchangeClient* and *reportServer*.

enterprise server is represented by an *enterprise* component; it provides information about products and suppliers. It consists of the following components: *dataExchangeServer* and *database* (*objectStorage* and *transactionManager*).

Following section describes how the specification of components has been extended in order to support automated deployment planning. Finally, the planning process is shown on the example application.

5.1 Adding new information to components

The components of the *SOFAShop* application were not specified by *resource demands* and they did not have any behavior description. In order to allow the auto-

[1] Common Component Modelling Example

mated deployment planning for the application, following tasks had to be done:

- mark active components
- add specification of *resource demands* to primitive architectures
- run the *SOFAShop* application with the *MethodCallLog* aspect
- analyze obtained log file and update corresponding frames

At the first point it is necessary to mark active components. In the *SOFAShop* application all the events are triggered by users via GUI components. Therefore, the architectures of the *GUI* component (from the *cashDesk* component) and the *storeClient* component were marked as *active*; then they were committed into the repository using the *commit* action of the *cushion* tool.

Second, *method-info* files were generated for each primitive architecture using the *methodinfo* action of the *cushion* tool (see Section 4.6.1). The files were edited – the *resource demands* descriptions were filled in. The specified demands were estimated in regard to the architecture of the *SOFAShop* application. Majority of the components just delegate the method calls to other components; therefore, the demands are quite low. All components have demands on the *cpu* resource specified in *operations*; but only the *objectStorage* component has also demands on the *disk* resource specified in *bytes*. At the end all the primitive architectures were committed into the repository using the *commit* action of the *cushion* tool.

Next, new deployment plan for one store and one cash desk was created using the *deplplan* action of the *cushion* tool. The *MethodCallLog* aspect was specified to the deployment plan and all the deployment docks were set to the dock 'nodeA'. Then the plan was committed into the repository using the *deploy* action of the *cushion* tool. The application was launched according to the plan and it was used as if in general operation. This process generated the log file *method-log-file_nodeA.tmp*.

Finally, the tool *sofa-update-frames* was used on the generated log file. This tool processed the file and updated all respective frames in the repository (see Section 4.6.3). Now it is possible to use the automated deployment planning for the *SOFAShop* application.

5.2 Test of the automated deployment planning

The test of the automated deployment planning is shown on the *SOFAShop* application with one store and one cash desk. The testing environment consists of four running deployment docks: *cashdeskA*, *storeA*, *serverA*, *serverB*. Capabilities and hardware parameters of the docks are shown in Table 5.1. The *cashdeskA* and *storeA* have set intentionally lower values of hardware parameters as we do not wish to deploy there many components. The *serverA* has faster hard disk then *serverB*; therefore, it is appropriate for components with higher *disk* demands.

Parameters	Deployment docks			
	cashdeskA	*storeA*	*serverA*	*serverB*
system.resource.cpu	1000000	1000000	1000000	1000000
system.resource.disk	1000000	1000000	1000000	1000000
cpu-operations-per-second	1e6	1e6	1e9	1e9
disk-megabytes-per-second	20	20	80	40

Table 5.1: Capabilities and hardware parameters of the deployment docks.

A new deployment plan has been created using the *deplplan* action of the *cushion* tool. Deployment docks were set at components *GUI* and *receiptPrinter* to the value 'cashdeskA' and at the component *storeClient* to 'storeA'. These components have GUI interfaces; therefore, they should run where we expect. It would be also convenient to specify deployment docks to the components *objectStorage* of the two databases as they require specific files located on the disk. This was omitted because the result of the automated deployment planning would not be so interesting. Finally, the deployment plan was committed into the repository using the *autodeploy* action of the *cushion* tool (see Section 4.6.4).

Total resource demands of the components were computed (they can be seen in Table 5.2) and deployment docks were assigned to the components. Output of the automated deployment planning is following:

```
Processing deployment plan...
Start of the automated deployment planning.
Deploying components with specified dock name:
[/storeClient]: storeA ... OK
[/cashdesk/ReceiptPrinter]: cashdeskA ... OK
[/cashdesk/GUI]: cashdeskA ... OK
Creating order of components for deploying... OK
Deploying components in the specified order:
[/inventory/reportServer]: serverB
[/inventory/database/objectStorage]: serverA
[/inventory/database/transactionManager]: serverB
[/cashdesk/logic]: serverB
[/inventory/productsLister]: serverB
[/inventory/dataExchangeClient]: serverB
[/enterprise/database/objectStorage]: serverB
[/enterprise/database/transactionManager]: serverB
[/inventory/saleRegistrator]: serverB
[/inventory/orderRegistrator]: serverB
[/enterprise/dataExchangeServer]: serverB
[/inventory/stockLister]: serverB
```

```
Deploying composite components:
[/enterprise/database]: serverB
[/inventory/database]: serverB
[/inventory]: serverB
[/cashdesk]: serverB
[/enterprise]: serverB
[]: serverB
```

The components *GUI*, *receiptPrinter* and *storeClient* were deployed on the specified docks. The most resource demanding component *objectStorage* from the *inventory* database was put on the dock *serverA* as it has the fastest hard disk. The rest of the components were put on the dock *serverB* as their demands are quite low.

Component	Resource demands	
	cpu	*disk*
/inventory/database/objectStorage	295952	172756
/inventory/database/transactionManager	16365	
/inventory/stockLister	203	
/inventory/saleRegistrator	254	
/inventory/productsLister	1200	
/inventory/orderRegistrator	500	
/inventory/dataExchangeClient	1200	
/inventory/reportServer	1340	
/cashdesk/GUI	3000	
/cashdesk/logic	9958	
/cashdesk/ReceiptPrinter	1695	
/storeClient	1000	
/enterprise/dataExchangeServer	240	
/enterprise/database/objectStorage	460	200
/enterprise/database/transactionManager	40	

Table 5.2: Counted total resource demands of components. Demands for *cpu* are specified in *operations* units; demands for *disk* are specified in *bytes*. Values are rounded.

Chapter 6

Related work

In addition to the SOFA 2 there are also other academic component models providing rich set of features. However, they typically have insufficient runtime environment as they are oriented mainly on design. *Fractal* [4] is a component model with similar capabilities as SOFA 2. It has a number of implementations; one of them is *Julia*. Their approach to the deployment problem is mentioned in Section 6.1.

Section 3.2 have introduced possible approaches to the automated deployment planning. As far as we know only one of them was embodied into a real planner and that is the *Sekitei* planner (see Section 6.2).

The deployment problem is solved in solutions interested in grid and cloud computing. One of the solution is *ProActive* (see Section 6.3).

As far as we know there is no real implementation of the automated deployment planning for hierarchical component systems.

6.1 Julia

Julia [4] is the reference implementation of the *Fractal* [4] component model. It does not solve the deployment problem itself but it can use the *Fractal Deployment Framework* (FDF) [10]. The FDF provides means to facilitate the deployment of distributed applications and middleware on networked systems. It introduces a high level deployment description language and provides a set of end-user tools. The distributed system has to be described by a FDF configuration and then it can be deployed by the user via the auxiliary tool. However, the FDF does not provide any automated deployment planning.

6.2 Sekitei

The *Sekitei* planner has been described in Section 3.2.1. It solves the *Component Placement Problem* and it is implemented in Java as a pluggable module for component-based frameworks. It is used in the framework *Smock* [12] that serves as a

run-time environment of the *Partitionable Services Framework* [12]. The framework "enables services to be flexibly assembled from multiple components, and facilitates transparent migration and replication of these components at locations closer to the client while still appearing as a single monolithic service" [12].

In comparison to our solution, their approach solves more general problem. The planner decides on a particular set of components that will be deployed and may also introduce some auxiliary components during the deployment for fulfilling application constraints. It finds deployments also in resource constrained environments. Due to the generality of the problem the deployment planning may suffer from performance issues.

6.3 ProActive

ProActive Parallel Suite [19] is open source solution for parallel, distributed, grid and cloud computing. It contains tools for scaling up demanding applications. The suit is divided into three parts:

Programming is done by *Java Parallel Toolkit*. It is a framework that provides Java API for easy developing parallel and distributed applications.

Scheduling is provided by multi-platform *Job Scheduler*. It manages a user-defined pool of resources and parallel execution of tasks using the resources.

Resourcing provides smart and adaptive engine for application deployment.

ProActive Scheduling provides a framework for job definition and submission. The input for the scheduling are submitted jobs (e.g., Java or native applications, scripts, *ProActive Programming* applications) and descriptions of the task flow construction. Each job contains one or more tasks. It is possible to specify temporal and data dependencies, sequences of tasks; the specification is done via Java API, XML file of flat file. The *Job Scheduler* permits dynamic changes in the resource pool (hot plug and unplug of resources) and it automatically discovers new resources. This allows an increased resource utilization and a flexible infrastructure. It offers different scheduling policies of the jobs (e.g., first-come-first-served, priorities, planning).

The *ProActive Resourcing* is supported by *ProActive Agents*. The agent runs on a host computer and registers the resource at the resource manager when it is available (e.g., the last processing task is done or the host computer is not used – screen saver is on). It provides monitoring and controlling the computing resources and it manages deploying of applications.

The scheduling algorithm simply deploy tasks of a job one by one 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 run on a dock concurrently.

Chapter 7

Conclusion and future work

The goal of the thesis was to add support of cloud computing to the SOFA 2 system. In particular it was meant to make the access to the SOFA 2 system easier and to simplify launching of SOFA 2 applications, especially when preparing deployment plans of the applications. The issue related to the automated generation of deployment plans based on the current status of the SOFA 2 environment. To solve the problem, an analysis of available solutions had to be made. However, no approach of automated deployment planning dealt with hierarchical component systems. Therefore, one suitable algorithm has been chosen and it has been extended to support hierarchical component systems.

The problem of the automated deployment planning has been analyzed theoretically and the proposed algorithm has been described in general; thus, the solution can be used in any hierarchical component systems. It covers utilization of resources in the system and demands of components to find efficient deployment of the application.

The algorithm has been implemented in the SOFA 2 component system. All necessary changes to the SOFA 2 model have been described and auxiliary tools have been created. Finally, the *Cloud Gate* application has been created in order to simplify access to the SOFA 2 system and to simplify the management of applications, especially the launching of the applications.

To fully utilize the proposed deployment planning, precise values of components' resource demands are needed. Currently they have to be specified by user manually; therefore, it would be convenient to create a profiling tool that would measure resource demands of each component automatically. Also the *total resource demands* of components are currently computed based on the automated observation of the application behavior. In order to get more accurate results an extension of the behavior protocol has been proposed. However, auxiliary tools for processing the extended behavior protocol were left for future work on this topic.

Bibliography

[1] Arbab, F.: *Reo: a channel-based coordination model for component composition*, Mathematical Structures in Computer Science, Vol. 14, No. 3, Jun 2004.

[2] Becker, S., Koziolek, H., Reussner, R.: *Model-Based Performance Prediction with the Palladio Component Model*, Workshop on Software and Performance (WOSP'07), pp. 54–65, 2007.

[3] Broy, M., Deimel, A., Henn, J., Koskimies, K., Plasil, F., Pomberger, G., Pree, W., Stal, M., Szyperski, C.: *What characterizes a (software) component?*, Software – Concepts & Tools Journal, Vol. 19, No. 1, ISSN 1432-2188, pp. 49–56, Jun 1998.

[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, special issue on Experiences with Auto-adaptive and Reconfigurable Systems, 36(11-12), 2006.

[5] Bures, T., Hnetynka, P., Plasil, F.: *Runtime Concepts of Hierarchical Software Components*, In International Journal of Computer & Information Science, Vol. 8, No. S, ISSN 1525-9293, pp. 454–463, Sep 2007.

[6] Bures, T., Hnetynka, P., Plasil, F.: *SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model*, Proceedings of SERA 2006, Seattle, USA, Aug 2006.

[7] Common Component Modelling Example (CoCoME), `http://agrausch.informatik.uni-kl.de/CoCoME`

[8] Crnkovic, I., Chaudron, M., Larsson, S.: *Component-Based Development Process and Component Lifecycle*, icsea, pp. 44, International Conference on Software Engineering Advances (ICSEA'06), 2006.

[9] Eclipse Modeling Framework, `http://www.eclipse.org/modeling/emf/`

[10] Fractal Deployment Framework, `http://fdf.gforge.inria.fr/`

[11] Heydarnoori, A., Mavaddat, F., Arbab, F.: *Towards an Automated Deployment Planner for Composition of Web Services as Software Components*, Proceedings of FACS, 2005.

[12] Ivan, A.-A., Harman, J., Allen, M., Karamcheti, V.: *Partitionable Services: A Framework for Seamlessly Adapting Distributed Applications to Heterogeneous Environments*, Proceedings of HPDC-11, 2002.

[13] Kichkaylo, T., Ivan, A., Karamcheti, V.: *Constrained Component Deployment in Wide-Area Networks Using AI Planning Techniques*, Proceedings of IPDPS, 2003.

[14] Kichkaylo, T., Karamcheti, V.: *Optimal Resource-Aware Deployment Planning for Component-based Distributed Applications*, Proceedings of HPDC, 2004.

[15] Knorr, E., Gruman, G.: *What cloud computing really means*, InfoWorld, `http://www.infoworld.com/d/cloud-computing/what-cloud-computing-really-means-031`

[16] Mencl, V., Bures, T.: *Microcomponent-Based Component Controllers: A Foundation for Component Aspects*, Proceedings of APSEC 2005, Taipei, Taiwan, Dec 2005.

[17] OMG: *Deployment and Configuration of Component-based Distributed Applications Specification*, v4.0, Apr 2006, `http://www.omg.org/cgi-bin/doc?formal/06-04-02`

[18] Plasil, F., Visnovsky, S.: *Behavior protocols for Software Components*, IEEE Transactions on Software Engineering, Vol. 28, No. 11, Nov 2002.

[19] ProActive, `http://proactive.inria.fr/`

[20] Sharma, V. S., Jalote, P.: *Deploying Software Components for Performance*, Proceedings of CBSE, 2008.

[21] SOFA 2, `http://sofa.ow2.org/`

[22] SOFA 2 Component System Documentation: *User's guide*, `http://sofa.ow2.org/docs/pdf/users_guide.pdf`

[23] Trivedi, K. S.: *Probability and Statistics with Reliability, Queuing and Computer Science Applications*, Second Edition, John Wiley & Sons, New York, 2002.

Appendix A

Contents of the CD-ROM

The enclosed CD-ROM is organized as follows:

readme.txt A description of the contents of the enclosed CD-ROM and instructions for using it.

master-thesis.pdf This thesis in PDF format.

src/ Source code of the implementation.

bin/sofa-j/ Binary distribution of the SOFA 2 system with example applications in the repository.

bin/cushion/ Binary distribution of the *Cushion* tool.