Performance Simulation of Runtime Reconfigurable Software Architectures

Robert von Massow

1. April 2010

First examiner: Prof. Dr. Andreas Winter
Second examiner: Prof. Dr. Wilhelm Hasselbring (University of Kiel)
Advisor: Dipl.-Inform. André van Hoorn
Abstract

Resource efficient computing of software systems is increasing in importance in software development. While resource efficiency is mainly an internal quality attribute, the system performance is an external quality attribute, visible to system users. In order to provide an adequate system performance, i.e., assure short response times to user requests, the necessary hardware capacity is commonly provided in a static and pessimistic way. This thesis is the context of a self-adaptive approach for online capacity management for component-based software systems, called SLAStic. SLAStic proposes a set of reconfiguration operations to adapt the system capacity to the current workload of the system, thus optimizing the resource efficiency while satisfying the system’s performance objectives.

In this work we present SLAStic.SIM, a simulator for runtime reconfigurable, component based software architectures, supporting the reconfiguration operations proposed by the SLAStic framework. SLAStic.SIM is able to simulate Palladio Component Models (PCM) extended by information on a system’s reconfiguration capabilities. Our evaluation yields that SLAStic.SIM is a suitable simulator for performance evaluation of runtime reconfigurable component based software architectures model with PCM and can thus be used to evaluate the SLAStic approach for online capacity management in a simulative environment.
## Contents

1 Introduction .......................... 1
   1.1 Motivation .......................... 1
   1.2 Goals .................................. 3
   1.3 Document Structure .................. 6

2 Foundations ............................ 7
   2.1 Eclipse Modeling Framework .......... 7
   2.2 Palladio Component Model (PCM) ...... 8
   2.3 Discrete Event Simulation ......... 14
   2.4 Kieker Monitoring Framework ...... 16
   2.5 SLAStic .............................. 17

3 SLAStic.SIM Architecture and Integration 21
   3.1 SLAStic Architecture ............... 21
   3.2 SLAStic.SIM Architecture ............ 23
   3.3 Interaction with SLAStic.SIM ....... 23
   3.4 SLAStic.SIM Extension Points ...... 24

4 Simulation ............................. 27
   4.1 Simulation Model ..................... 27
   4.2 Simulation ............................ 32
   4.3 Reconfiguration ....................... 35
   4.4 Monitoring ............................ 40
   4.5 Workload .............................. 40

5 Evaluation ............................. 43
   5.1 Evaluation Goals ..................... 43
   5.2 Performance and Scalability ......... 44
   5.3 Simulation Setup ..................... 45
   5.4 Bookstore Example ................... 46
   5.5 Single Call Experiment ............... 50
   5.6 Multi Call Experiment ............... 53
   5.7 Multi Call Experiment with Reconfiguration 55
   5.8 Simulating a Workload Curve ......... 59
6 Related Work 65
6.1 SimuCom ................................. 65
6.2 Java Modeling Tools .................. 66
6.3 ArgoSPE ................................. 67
6.4 Simulating Service Oriented Architectures with OMNeT++ ... 68
6.5 MOSES .................................. 69

7 Conclusions and Future Work 71
7.1 Summary .................................. 71
7.2 Discussion ............................... 71
7.3 Future Work .............................. 72

A Workload Generation Script 75

Bibliography 77

Declaration 81
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Workload recorded at a commercial web shop (vHRGH09)</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Schematic visualization of the goals of this work</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>A simple repository example</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>The RDSEFF describing the behavior of the service a implemented by the component AComponent</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>The RDSEFF describing the behavior of the service b implemented by the component BComponent</td>
<td>11</td>
</tr>
<tr>
<td>2.4</td>
<td>A simple example system showing how the components interact</td>
<td>12</td>
</tr>
<tr>
<td>2.5</td>
<td>An example of a resource environment</td>
<td>13</td>
</tr>
<tr>
<td>2.6</td>
<td>An example allocation in which the assembly context (&lt;BComponent&gt;) is replicated</td>
<td>13</td>
</tr>
<tr>
<td>2.7</td>
<td>An example of an open workload specification in PCM</td>
<td>14</td>
</tr>
<tr>
<td>2.8</td>
<td>Replication of component C2 to Resource3</td>
<td>19</td>
</tr>
<tr>
<td>2.9</td>
<td>The final result of a Migration operation. For step one, see Figure 2.8</td>
<td>20</td>
</tr>
<tr>
<td>3.1</td>
<td>Overview of the SLAStic architecture</td>
<td>22</td>
</tr>
<tr>
<td>3.2</td>
<td>Architecture of SLAStic.SIM</td>
<td>23</td>
</tr>
<tr>
<td>3.3</td>
<td>The SimulationController, its implementing interfaces and owned classes</td>
<td>24</td>
</tr>
<tr>
<td>4.1</td>
<td>The ModelManager serves as our simulation model</td>
<td>28</td>
</tr>
<tr>
<td>4.2</td>
<td>A class diagram of the hardware model including schedulers</td>
<td>31</td>
</tr>
<tr>
<td>4.3</td>
<td>Activity diagram of the processor sharing scheduler</td>
<td>35</td>
</tr>
<tr>
<td>4.4</td>
<td>Activity diagram of the replicate component operation</td>
<td>36</td>
</tr>
<tr>
<td>4.5</td>
<td>Activity diagram of the dereplication operation</td>
<td>37</td>
</tr>
<tr>
<td>4.6</td>
<td>An activity diagram of the migration operation</td>
<td>38</td>
</tr>
<tr>
<td>4.7</td>
<td>Activity diagram of the allocate resource operation</td>
<td>39</td>
</tr>
<tr>
<td>4.8</td>
<td>Activity diagram of the deallocate operation</td>
<td>40</td>
</tr>
<tr>
<td>4.9</td>
<td>Sequence diagram of a user’s call trace annotated with Kieker log information (vHRH^+09)</td>
<td>41</td>
</tr>
<tr>
<td>5.1</td>
<td>The Bookstore Repository</td>
<td>47</td>
</tr>
<tr>
<td>5.2</td>
<td>The service searchBook() as implemented by the component Bookstore</td>
<td>48</td>
</tr>
</tbody>
</table>
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3</td>
<td>The service <code>getOffers()</code> of the component <strong>CRM</strong></td>
<td>48</td>
</tr>
<tr>
<td>5.4</td>
<td>The service <code>getBook()</code> of the component <strong>Catalog</strong></td>
<td>49</td>
</tr>
<tr>
<td>5.5</td>
<td>The Bookstore system model</td>
<td>49</td>
</tr>
<tr>
<td>5.6</td>
<td>The initial allocation model of the Bookstore</td>
<td>50</td>
</tr>
<tr>
<td>5.7</td>
<td>Two sequence diagrams generated with Kieker.Tpan</td>
<td>52</td>
</tr>
<tr>
<td>5.8</td>
<td>Workload scenario used to drive SimuCom</td>
<td>53</td>
</tr>
<tr>
<td>5.9</td>
<td>Time series plot of response times of the service <code>searchBook</code> for the multi call experiment performed with SLAStic.SIM</td>
<td>54</td>
</tr>
<tr>
<td>5.10</td>
<td>Time series plot of response times of the service <code>searchBook</code> for the multi call experiment performed with SimuCom</td>
<td>54</td>
</tr>
<tr>
<td>5.11</td>
<td>Time series plot of the response time to the <strong>Catalog.searchBook</strong> service without replication</td>
<td>57</td>
</tr>
<tr>
<td>5.12</td>
<td>Time series plot of the response time to the <strong>Catalog.searchBook</strong> service with replication and dereplication occurring at the time listed in Table 5.2</td>
<td>57</td>
</tr>
<tr>
<td>5.13</td>
<td>Time series plot of the response time to the <strong>Catalog.getBook</strong> service with replication and dereplication occurring at the time listed in Table 5.2</td>
<td>58</td>
</tr>
<tr>
<td>5.14</td>
<td>Dependency graph for the multi call experiment without reconfiguration</td>
<td>58</td>
</tr>
<tr>
<td>5.15</td>
<td>Dependency graph for the multi call experiment with reconfiguration</td>
<td>59</td>
</tr>
<tr>
<td>5.16</td>
<td>Number of active sessions</td>
<td>60</td>
</tr>
<tr>
<td>5.17</td>
<td>Response time of the service <code>searchBook</code> under varying workload</td>
<td>60</td>
</tr>
<tr>
<td>5.18</td>
<td>Response time of the service <code>getBook</code> under varying workload</td>
<td>61</td>
</tr>
<tr>
<td>5.19</td>
<td>Response times of the service <code>searchBook</code> under varying workload with replication</td>
<td>62</td>
</tr>
<tr>
<td>5.20</td>
<td>Response times of the service <code>getBook</code> under varying workload with replication</td>
<td>62</td>
</tr>
<tr>
<td>5.21</td>
<td>Operation dependency graph of the workload curve experiment without reconfig</td>
<td>63</td>
</tr>
<tr>
<td>5.22</td>
<td>Operation dependency graph of the workload curve experiment with reconfig</td>
<td>63</td>
</tr>
</tbody>
</table>
List of Tables

5.1 Evaluation Hardware.................................................. 45
5.2 Reconfiguration operations and the simulation time at which they occur................................................. 56
6.1 Comparison between SLAstic.SIM and SimuCom.............................................................................. 66
6.2 Comparison between SLAstic.SIM and JMT.................................................................................... 67
6.3 Comparison between SLAstic.SIM and ArgoSPE................................................................................ 68
6.4 Comparison between SLAstic.SIM and ProC/B.................................................................................. 69
6.5 Comparison between SLAstic.SIM and MOSES.............................................................................. 70
Chapter 1

Introduction

Today, large scale software architectures are often configured statically. Nevertheless, it is desirable to develop software architectures to be runtime reconfigurable due to several reasons. One of them is the online update of software components without reducing the availability of the system [Bun08; Mat09]. However, in this thesis we focus on resource efficiency and performance, thus using different reconfiguration operations like replication and dereplication of components defined by van Hoorn et al. [vHRGH09]. We present a simulation framework (SLAStic.SIM) for simulating models of reconfigurable software systems, thus making it possible to evaluate the SLAStic approach to self-adaptive online capacity management. Our system will integrate into the SLAStic framework and can be used in two scenarios. (1) Offline evaluation (i.e., standalone simulation) makes it possible to assess the performance of a software system with regard to different configurations. (2) Online evaluation (i.e., simulation while running the system) makes it possible to evaluate different configurations before the running system is reconfigured.

1.1 Motivation

Simulating reconfigurable software systems before actually executing them gives us the ability to evaluate several adaptation plans and thus assess their effects on the system’s performance. The adaptation of the system can also have an impact on its power consumption and hence reduce the costs to run it while still being able to assure a certain quality of service regarding the response times to user requests.

Another advantage of this approach to large scale system development is the possibility to forecast the necessary resources (capacity planning) by configuring the runtime analysis framework and the simulator to have a varying set of resources and simulate the system with the expected workload. Often, for large scale systems a customer and a provider negotiate specific SLAs (Service Level Agreement) to be satisfied. This could be, for instance, '90% of all requests to the system have to be answered within 500ms', otherwise a contractual penalty
Chapter 1 Introduction

has to paid. If the SLAs do not hold for a given configuration, the adaptation controller will trigger a reconfiguration. The system will then asymptotically approach an optimal deployment. Now we can extract the deployment environment and hence generate an optimal hardware setup that is needed to run the system optimally thus reducing maintenance costs and down times due to hardware changes.

With SLAStic.SIM it would be possible to predict a systems performance for different workloads and architectural configurations. Due to SLAs serving as a contract for a given system, it is highly desirable to have systems that will never violate the SLA contracts. To achieve this, currently the worst case, i.e., the situation with the highest workload, is estimated and then the system is deployed to the hardware that is needed to satisfy the SLAs in the worst case. Often, this means that the hardware works far below its capabilities. In Figure 1.1 a graph of a real world’s software system workload is depicted. We can see easily that there are times with really low workload, and high peaks on the other side.

Due to the static deployment, the provided system capacity is constant throughout the whole period. It is marked by a green line in the same figure. In order to save power and thus reduce costs, it is now desirable to dynamically adapt the system capacity in such a way that on one hand the SLAs will always be satisfied and on the other hand the fewest possible servers run. The red line in Figure 1.1 shows the desired (dynamic) system capacity. It over-approximates a little to ensure there sufficient capacity in the case of bursts, i.e., the arrival of many users at a short period. To meet this goal, the future workload of a system has to be predicted and the system has to be reconfigured accordingly.
1.2 Goals

In order to evaluate this approach, we present a simulator for runtime reconfigurable component based software architectures. A priori (offline) simulation and evaluation has the advantage that no business critical services of a running software system are subject to failures due to reconfiguration. Also it is cheaper, as the hardware costs for the resource environment for large scale systems are very high, thus it is not feasible to run the system experimentally. If the approach is feasible and implemented in a real system, SLAStic.SIM can be used to evaluate different reconfiguration plans at runtime by evaluating the response time of the simulated services and the utilization of the servers. We aim at resource efficient computing.

Additionally, runtime reconfiguration also gives the advantage to save power during low workload periods by shutting down servers. Today’s servers are already often designed to consume as little power as possible, as the cost for power is not neglectable for data center operators. But due to the static deployment of the systems, all the servers have to be running permanently and can only be shutdown by human interaction, making a shutdown a cost intensive operation. Using the SLAStic framework it would be possible to automate this process. SLAStic.SIM could again be used to estimate the power consumption by monitoring the servers’ running periods and their load. These monitoring records could then be analyzed by SLAStic.CONTROL.

1.2 Goals

In this section we give an overview over the goals of this work. A schematic visualization is given in Figure 1.2. SLAStic.MON will receive performance data from a simulated system which is monitored using Kieker by Rohr et. al. [RvHM+08]. The data is analyzed by SLAStic.CONTROL, presented by Stöver [Stö09], which outputs an adaptation plan via SLAStic.REC if necessary. The adaptation plan can contain one or more reconfiguration operations which will be treated as a transaction. SLAStic.SIM will then reconfigure the system according to the adaptation plan without stopping the simulation.

1.2.1 Simulation

The main goal is to implement a simulator for runtime reconfigurable software systems (G2 in Figure 1.2). The simulator will be fed with a system model and should take generated or previously recorded workload as its input. The simulator should run at an adaptable speed, because the runtime analysis framework must not be slower than the simulator. Otherwise there will be an accumulating amount of performance data and the SLAStic.CONTROL is possibly not able to analyze this data in a reasonable time. Additionally, the SLAStic.CONTROL and
SLAStic.SIM should have a common interpretation of time, hence there might have to be a global clock that runs at an adjustable speed or there are some synchronization point, at which SLAStic.SIM tells SLAStic.CTRL about the model time.

The performance data should be collected using the Kieker monitoring framework or SLAStic.SIM has to output Kieker-compatible data. Therefore, the simulator has to monitor the simulated system. Monitoring the simulator is not feasible as the simulator itself is not the target of the performance analysis. To solve this problem we need instrumentation support for the simulated system model.

Finally we need support for the dynamic reconfiguration operations described in Section 2.5. These will be triggered as reconfiguration plans, generated by the runtime analysis framework. These reconfiguration plans can be used as a transaction on the model. One of the main aspects here is that we must not lose any performance information due to a transaction on the simulated system model. This means, if we stop the simulator, and then restart it with a new, reconfigured model, the user data on their component usage must be the same, i.e., the simulator has to store the user information and then remap it to the new model.
1.2 Goals

1.2.2 Simulation Model

To simulate a system we need a simulation (meta) model. This (meta) model will be for simulator-internal use only. Hence, it can differ from the model used inside the runtime analysis framework. The core of the meta model has to be made up of components and connectors as we are simulating component based software systems. In order to evaluate performance, the component’s services have to be annotatable with performance characteristics, e.g. CPU usage or memory consumption. Furthermore, we need to be able to model deployment environments, offering the possibility to express such things as CPU capacity, the amount of memory and network links. Otherwise the modeled performance characteristics of the component’s services will not be evaluable.

In order to support system reconfiguration we also need to store some information whether certain reconfiguration operations are allowed to be performed on a specified component. Additionally, we have to keep track of the amount of instances for replicable components to make sure there is at least one instance left during runtime. This information has to be combined with a load balancer that distributes the workload among replicated components of the same type.

As we want to monitor the modeled system during runtime, we also need instrumentation support. It has to be expressible that a call to an external service, i.e., a service that is provided by another component than the calling one, is monitored. Respectively, for evaluating the internal control flow of a component’s service, branches have to be monitorable, too.

1.2.3 SLAstic Reconfiguration Operations

Finally, SLAstic.SIM has to support the SLAstic reconfiguration operations described by van Hoorn et.al. [vHRGH09] (G3 in Figure 1.2).

1.2.3.1 Allocate/Release Server Node

Allocating a server node means to make it available (e.g. switching it on or spawning a virtual machine) so it can be used for computing service requests in the future. In order to do this, of course, components have to be deployed to the server.

Deallocating a server means to shut it down, hence saving power. Of course, a deallocated server cannot be used for computing any service requests. To keep the system in a consistent state, no component must reside on a server that is to be deallocated.
1.2.3.2 Replicate/Dereplicate Component

Replicating a component means to deploy another instance to a server node. This technique has been implemented in database systems in order to improve the availability and reliability of the database management system [FKT92]. However, the SLAstic replication operation is targeted at decreasing the response time of a service by distributing the computation of requests among different server nodes.

Dereplication of a component is just the opposite of replication, i.e., it removes an instance from the system. In order to keep the system consistent, no transactions must be run on the component that is to be dereplicated.

1.2.3.3 Migrate Component

The component migration operation moves a component’s instance from one server to another, thus changing its deployment. Again, running transactions must not be broken by the migration, too.

1.3 Document Structure

The document is structured as follows. Chapter 2 provides an overview of the foundations, starting with the Eclipse Modeling Framework in Section 2.1 followed by the Palladio Component Model in Section 2.2, then introducing the technique of discrete event simulation in Section 2.3, afterwards giving an overview over the Kieker monitoring framework in Section 2.4, and finally giving an introduction into the SLAstic approach for self-adaptive online capacity management in Section 2.5. This is followed by a brief description of the architecture that has been developed throughout this work in Chapter 3. Then we go into detail by describing the simulation model and its interpretation in Chapter 4. An evaluation of SLAstic.SIM is given in Chapter 5 using multiple workload and reconfiguration scenarios. We state some related work in Chapter 6 and finally conclude and discuss our results in Chapter 7, where also future extensions are proposed.
Chapter 2

Foundations

In this chapter we give an overview over the foundations of our work. We start
with the Eclipse Modeling Framework, a simple meta modeling language. Then
we introduce the Palladio Component Model and its graphic notion. Afterwards
we describe the technique of discrete event simulation and introduce Desmo-J,
which is a framework for discrete event simulation followed by a brief introduction
into the Kieker monitoring framework. Finally, we give an overview over the
SLAStic reconfiguration operations developed by van Hoorn et.al. [vHRGH09].

2.1 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) is a (meta) modeling facility for the
development of software based on a structured data model. It uses the Ecore
meta modeling language, which is a self contained meta model similar to UML’s
Meta Object Facility (MOF).

Meta models are edited by a special Ecore editor. This editor is rather graph-
ical than textual and looks more like a high level XML editor (i.e. “tree-style”).
There also exists a graphical way to edit meta models via the Graphical Modeling
Framework (GMF). GMF comes with a graphical Ecore editor which looks very
much like other common editors for (meta-)modeling like for example Netbeans
UML.

Meta models formulated in Ecore can be instantiated and manipulated pro-
grammatically using the emf.edit API [BSM09]. There are two implementa-
tions, reflective and static EMF. For reflective EMF one can directly work with
a parsed serialized EMF model accessing the attributes of classes by their names
using a generic property getter or setter method that takes the property’s name
as an argument. For static EMF we have to generate the meta model’s code.
This would result in a bean style code and thus enables the programmer to access
attributes of the generated classes by generated getters and setters. It is also pos-
sible to use the EMF generation feature to generate a specialized emf.edit project,
and an Eclipse editor plugin and a test suite. The default Ecore meta model ed-
itor is generated from the Ecore meta model which is Ecore again. The editor’s
interface is an implementation of Eclipse’s org.eclipse.jface.viewers.TreeViewer combined with a properties editor. Saving a model results in XMI-serialization [BSM+09].

In our work, we use EMF to model the reconfiguration plans and the component reconfiguration model using the possibility to import a given meta model and thus extend it.

### 2.2 Palladio Component Model (PCM)

The Palladio Component Model (PCM) is a meta model designed for the performance prediction of static software system architectures. The modeling of software architectures is done in several viewpoints, which focus on different aspects of the software architecture. Using different viewpoints helps domain experts to model system properties without having to have knowledge about other domains. Every view has its own diagram style, hence it is possible to edit the architectural model graphically. The diagram styles are inspired by the UML but differ in several ways. The reason for not using UML was the weak semantics [BKR09]. The viewpoints and diagrams are described in the following.

The PCM is distributed as an Eclipse plugin and also features several possibilities to analyze a given architecture model.

#### 2.2.1 Repository

The repository view enables component developers to model system components and their required interfaces’ and services. Performance characteristics of service implementations provided by software components are specified in this viewpoint. The result is a repository of components which can be used by a system architect to interconnect different instances of the modeled components using the system viewpoint (see Section 2.2.2).

#### 2.2.1.1 Interfaces and Components

Interfaces specify services in the form of signatures that can be provided or required by components. A component that provides an interface has to implement the services declared in the interface. Interfaces themselves are neither providing nor requiring as they only specify which services a component has to implement. The required services cannot be specified on an interface level as the interface does not offer any information about the implementation of its declared services. The components however specify how the services declared by an interface are implemented and whether other services are needed.
Components and interfaces are modeled using a diagram that is similar to the UML’s component diagram. It does not allow for the “lolly” notation of interfaces as their declared services are essential for modeling a sane repository. The components on the other hand have only one compartment, in which the services execution behavior has to be described (see Section 2.2.1.2). Components cannot contain any classifiers as specified in the UML [OMG09], they can be composite, i.e., contain other components, though. A component that does not contain other components is called basic component.

A simple repository diagram is depicted in Figure 2.1. The interface A declares a service a which has no parameters and returns a bool-value. The interface’s role is provided by the component AComponent. The component has a Service Effect Specification (SEFF), described in Section 2.2.1.2, describing the service a. AComponent requires a service b that is declared by the interface B. The role of the interface B is provided by the component BComponent, which has a SEFF describing the execution behavior of the service b.

![Figure 2.1: A simple repository example](image)

**2.2.1.2 Resource Demanding Service Effect Specifications**

Components have to specify their services’ visible behavior by Resource Demanding Service Effect Specifications (RDSEFF). A RDSEFF is a gray box model of a service’s execution. It specifies the resource usage (e.g., CPU or hard disk) during a service’s execution but gives no information about the operations that cause the resource usage.

This is done using an UML activity diagram style graphical notation. The nodes of the diagram are AbstractActions. This can be a Branch or LoopAction that is attributed by branch or loop probabilities or guards respectively, an InternalAction which specifies resource usage, an ExternalCallAction which defines how the service interacts with another component’s service, a ForkAction or JoinAction that specifies the components threading behavior, or an Acquire— or
ReleaseAction for modeling the usage of so called passive resources, e.g., threads from a thread pool.

In contrast to activity diagrams in the UML, branches in PCM cannot be used to iterate. Therefore loops have been introduced. In general there is no possibility to use direct recursion. Also many modeling elements of activity diagrams are not used, e.g., there is no possibility to send or receive signals or separate between control flow and data flow. There is however a notion for variable characterization and usage as the services can have parameters and return types. The resource demands of InternalActions can be parametrized using the values of parameters and variables. We give a small example in Figure 2.2 and Figure 2.3, which are the diagrams describing the service a and b defined in Figure 2.1.

![Diagram of RDSEFF describing the behavior of service a implemented by component AComponent](image)

**Figure 2.2:** The RDSEFF describing the behavior of the service a implemented by the component AComponent

The service a first sets the return value to true and then calls the service b. As service b does not return any value and also has no parameters, the InputVariableUsage and OutputVariableUsage are empty. Note that service a does not contain any internal actions. This means it does not need any time to be computed. Although this is not realistic, it enables an optimized view on the system.

Service b features a probabilistic branch. Each possible branch transition contains an InternalAction, demanding some CPU units. The unit of measurement is not specified, it has to be time independent though, as a faster CPU uses less time to compute the same task than a slower one. Hence, cycles is a feasible
unit to specify the resource usage for CPUs, although, as we don’t know which operations are performed, it is impossible to calculate how many CPU cycles are actually needed to complete a specific task without knowing anything about the hardware. This is due to the complexity of modern CPUs, which use features such as pipelining or Streaming SIMD Extensions (SSE), that have an essential impact on the CPUs performance [Tak07] as they parallelize the computation, thus reducing the number of cycles needed to perform a certain task.

### 2.2.2 System

In a system model a system architect can instantiate components by creating so-called Assembly Contexts. Each Assembly Context contains exactly one component and provides ports for calling the containing components services. It can also require ports if the component depends on other components. It is possible to instantiate the same component several times. This can be useful if two different components require the service of a third component. In the system model also the system’s externally provided services are specified using delegation connectors.

The diagram style that is used is similar to component diagrams again, but in contrast to the repository view, it is rather a black box view on the components and interfaces.

---

**Figure 2.3:** The RDSEFF describing the behavior of the service \( b \) implemented by the component \( 
BComponent \n\)
In Figure 2.4 an example system is depicted. We can see that the component \texttt{AComponent} has been instantiated in the assembly context \textless \texttt{AComponent} \textgreater. It is connected via the interface \texttt{B} to the assembly context \textless \texttt{BComponent} \textgreater, which is an instance of the component \texttt{BComponent}. The interface \texttt{A}, which is provided by \textless \texttt{AComponent} \textgreater is delegated to the system boundary, so the only way to interact with the system is via the interface \texttt{A}.

### 2.2.3 Resource Environment

The Resource Environment view is used by infrastructure experts. Here the hardware infrastructure is modeled by using Resource Containers. These containers can be seen as servers in the real world. They contain different resource specifications, i.e., CPU, hard disk and network interfaces speed. There also exists a special delay resource. The unit of a resources speed is derived from the resource itself. It has to be scaled in a way that it fits the needs. For example, it is possible to specify a CPU speed of 1000, which could be 1000 Hz, 1000 kHz or 1000 MHz. As it is mapped to simulation time anyway, the interpretation of the results inherently determines what a CPU speed of 1000 means. This holds for all resources but has to be consistent among them. This means that the scale for a hard disk has to be Megabytes/s if and only if we also define the CPU speed in MHz because otherwise the time consumption among the resource usage would not be consistent, thus rendering the performance results invalid.

It is also possible to select a scheduling strategy per resource. Currently there are \texttt{DELAY}, simply delaying the execution for a given time, first come first served (\texttt{FCFS}), which queues all request and serves them strictly sequential and processor sharing (\texttt{PS}), which is an idealized derivative of round robin scheduling, assuming that switching between processes does not consume time.

The resource environment view does not use any diagrams. Instead resource containers are specified in a tree editor. An example is given in Figure 2.5.

There are two resource containers, each of them containing a CPU with a speed of 1000, a hard disk with a speed of 27 and a network interface with a speed

---

*Figure 2.4: A simple example system showing how the components interact*
2.2 Palladio Component Model (PCM)

2.2.4 Allocation

The allocation view is used by system deployers. In this view the deployer can map AssemblyContexts to AllocationContexts which can then be deployed to a resource container. Each AssemblyContext can be assigned one or more AllocationContexts, enabling the deployer to replicate components, e.g., to achieve load balancing.

2.2.5 Workload

PCM also features a workload specification language. It is used to define interaction of users with the modeled system. A user behavior is modeled in a usage...
scenario. The scenario basically uses the same syntax as a RDSEFF, except for the resource usage, which of course cannot be modeled. It is a graph, whose nodes are either system calls, loop actions or branches. Two different types of workload can be specified: open and closed workload. Whereas, in an open workload the number of users is unbounded and only an arrival rate is specified, the number of users in a closed workload is fixed and a think time, which is the time between two actions in the scenario, is specified.

![Diagram of an open workload specification in PCM]

Figure 2.7: An example of an open workload specification in PCM

A very simple example is given in Figure 2.7. It is an open workload specification with an arrival time of 42. This means, every 42 time units this scenario is started regardless how many instances of this scenario are active. The only action that is performed is a call to the system service a of the interface A.

2.3 Discrete Event Simulation

In this work we use the discrete event simulation technique to simulate a given software architecture. This technique is state of the art for systems’ performance diagnosis for which analytical methods are not feasible [Ban98].

2.3.1 Entities

In discrete event simulation the simulation model consists of so-called entities. Entities are models of the real world objects of the system that is to be simulated. They have a state which can change over the time of a simulation run due to the occurrence of events. The state is defined as the entity’s attributes’ value assignments, e.g., a communication channel is used or free, or the number of processes in a queue waiting for a processing resource.

In order to model the dynamic change of an entity’s state over time it also has to provide methods for triggering well-defined state changes. This could be
setting the usage state of a communication channel or a method for adding a process to a CPU’s queue. A model’s state emerges from all its entities’ states.

2.3.2 Events
During simulation, events trigger the evolution of the simulation model. Events can trigger state changes of entities and state changes of entities can in turn cause more events. These events are scheduled to occur at a given time, which must not lie in the past. A scheduler maintains a queue of scheduled events. During simulation, the simulation time is advanced to the first event in the queue and the event is triggered. After completion of the processing of an event this step is repeated until there is no event left in the queue.

Events for our example entities could be arrival of a communication task or a CPU job. Also, for modeling processing of a communication or CPU task, it would make sense to schedule an event for the end of the processing.

2.3.3 Time
In a discrete event simulation we do not use the real time that is passing during computation. We rather use an internal clock that is decoupled from the real world’s time. We refer to this as the model time. As mentioned before, this time is advanced by the scheduler according to the next event in the event queue, thus stepping in discrete time intervals of possibly variable length.

There is actually a trade off between the granularity of time and the simulation performance. Consider an interval of length $\Delta t$ between the occurrence of two events. This would advance the model clock by $\Delta t$ obviously and cause the computation of one event. If we consider the interval $\delta t$ between two events, it would need two of these intervals to advance the model clock by $\delta t$ while processing two events. So this forms an anti-proportional relation between the granularity of time and the computation time of the simulation.

2.3.4 Desmo-J
The Desmo-J framework is an extensible Java framework for discrete event simulation [PK05]. It is possible to design a simulation using two different approaches: event-oriented or process-oriented. We focus on the event oriented fashion here, as this is used in SLAStic.SIM.

The framework clearly distinguishes between the model and the experiment, which can be seen as a simulator core. The API features often used data structures like queues or different models of resources. By extending abstract classes for entities and events, a model is built. This model is then connected to an experiment, which in turn can be started, causing the model to be simulated. Before starting
an experiment, it can be configured. This means we can set a stop condition, either by setting a simulation time at which the simulation has to stop or by setting a dedicated Condition-object. In the latter case, the Condition’s check() method is called and the simulation is stopped iff true is returned. It is also possible to set trace and debug periods, causing the events and entities, that are configured to be included in trace or debug reports to be included in the Desmo-J output. The output is generated using the log4J API and is pretty-printed in HTML pages.

In order to build a model, a simulation designer extends the abstract class Model of the Desmo-J API, which declares the abstract methods init() and doInitialSchedules(). The init() method is used to set up the model’s components, i.e., entities, resources, and random number generators. The doInitialSchedules() method is called before the start of the simulation. It needs to schedule at least one event, because the simulation will fail as soon as the event queue is empty.

Events have to extend the abstract class ExternalEvent, which declares the method eventRoutine(). This method is called when the scheduler triggers the event. The model time is stopped during the execution of an event routine. In order to not let the event queue run empty it is necessary that new events are scheduled directly by the execution of the event routine or by the state change of a modified entity.

Some models also feature periodic events, like for example polling some model statistics at given intervals. To achieve this, event generators have to be implemented emitting a dedicated event at the given interval time. In order not to pollute the event queue with masses of periodic events, the dedicated event routine triggers a callback in the event generator. Unfortunately, it is not possible to schedule the same event instance again, thus causing a garbage collection overhead.

In SLAstic.SIM we use the Desmo-J API to implement the simulation core. A runtime reconfigurable model serves as input and is translated into a Desmo-J simulation model. External calls to the simulated system drive the simulation, which are processed internally by the implemented simulation core (see Chapter 4).

### 2.4 Kieker Monitoring Framework

The Kieker monitoring framework is a flexible Java framework for continuous monitoring and analysis of arbitrary Java software systems on application and middleware level [RvHM+08; vHRH+09]. This way it is possible, e.g., to detect performance bottlenecks in a software system.
During runtime of the monitored system, Kieker enables to collect trace information. Traces consist of Kieker records, which are produced by monitoring probes. Due to Kieker's architecture, it is possible to add different monitoring record types as needed by extending the AbstractKiekerMonitoringRecord. At the moment there are two important monitoring record types that are shipped with Kieker. The KiekerExecutionRecord, which is designed to measure timing and control flow information of method executions. It records the execution order index (eoi), which is the index of the operation in the monitored trace, the execution stack size (ess), denoting the stack depth of the operation, the entry time (tin), the return time (tout), the session and trace id, and the names of the operation, component, and host of the execution. In addition to this record type, there also exists the KiekerBranchingRecord, which is designed to monitor the control flow inside a method. Using this record type, it is possible to monitor the outcome of a branch.

Monitoring probes can be inserted using aspect oriented programming. This results in a low runtime overhead. It is also possible to generate the monitoring data programmatically. This is not suitable for most systems, as it is often not desired to mix the business logic code with the monitoring code, hence losing the separation of concerns.

In SLAstic.SIM we use Kieker to monitor the simulated software system. We implemented a specialized monitoring probe which enables us to monitor the simulated system using the model time.

2.5 SLAstic

SLAstic is a framework aiming at resource efficient operation of component based software systems [HRGH09]. It uses the idea of online capacity management by reconfiguring the architecture during runtime, thus implementing a self adaptive system. The SLAstic.SIM will implement the reconfiguration operations defined by the SLAstic approach. Namely, these are

- allocate/release server node
- replicate/dereplicate component
- migrate component

These are described in detail in the following.

2.5.1 Allocate/Release Server Node

In a deployment every component is mapped to a resource in the deployment context. This can be seen as the piece of hardware the specified component runs
on or the container it resides in, e.g., an application server. Hardware resources have to be divided up into CPU power, hard disk space, memory and their connectivity. Software resources have to be divided into threads (for thread pooling), database connections (for connection pooling) and semaphores (for synchronization of a large amount of processes). These resources reside in a resource pool, which is not static. It should be possible to add or remove resources during runtime, e.g., if a new server has been bought or some server broke down the resource pool has to be adjusted.

Allocating a new resource means that we have to add it to our current deployment context. For hardware resources this means not only to add the requested resource but add a whole server, which the resource belongs to. This means that it is very useful to know about future resource demands of the system. These forecasting tasks will be part of the runtime analysis framework [Stö09]. A problem that can arise is network inconsistency. We have to assure that the server we are adding is connected to the other servers to be usable. An unconnected server renders every migration or replication request to that specific server impossible, thus could make the whole analysis framework unusable. If the resource pool does not contain enough resources to fulfill an allocation request, we have to inform the runtime analysis framework via the SLAstic middleware so it can decide how to solve the problem. The information provided in that feedback should also contain data on how much of the requested resource is left in the pool to be able to reason about how to continue.

Releasing resources means to remove them from the deployment environment and re-add them to the resource pool. Again, we cannot only remove a CPU for instance, but have to remove the whole server the CPU belongs to. This means that there will be either a communication overhead with the SLAstic.REC for negotiating which server can be removed or the runtime analysis framework has to be aware of the resource to server mapping and make appropriate decisions regarding this fact. Networking problems can also occur. When we remove a server from the deployment context we have to make sure that this does not partition the network of the remaining servers thus rendering the system inoperable. Of course, a release request has to fail if a resource is still in use, i.e., a component is deployed to it (in the case of a hardware resource) or uses it (in the case of software resources like database connection for instance).

### 2.5.2 Replicate/Dereplicate Component

Replication of components in our context means adding another instance of a specific component to the deployment. This is possible if there are enough resources to run the specified component. It does not make sense to let one server host two identical components. Furthermore, not every component is replicable. Take a database component for instance, if the database is run on one specific
server and only locally accessible, replicating the component does not make sense and will result in undesired effects if the services of the unconnected database component’s replicate are called. Additionally, if a component is stateful (i.e., has an internal state) that is not serializable or reproducible, it cannot be replicated.

When replication is possible, another instance of a component will be added to the deployment. The new instance and every other replica of the component now has to be connected to a load balancer which has to solve the problem of distributing the service requests to the replicas. Optionally, the distribution could be parametrized, e.g., if there are two replicas, one of them serves 70% and the other one 30% of the incoming service calls.

In Figure 2.8 a graphic example on how the replication operation should be performed is given. First, the original system in Figure 2.8a is shown. The component C2 has only one instance running on Resource2. Now we receive a replication request. Assuming that the request is valid (i.e., the component is migratable) the replication should take place. The effect is depicted in Figure 2.8b.

![Figure 2.8: Replication of component C2 to Resource3](image)

2.5.3 Migrate Component

The migration operation will migrate a component from one resource to another, i.e., changing a components deployment. Moving a component from one resource to another can probably not be done in one step, as this would mean to move running transactions as well. The approach we will take is replicating the
component to the target resource and telling the load balancer that the original component should not receive any new service request. For instance say, we want to move component C2 from Resource2 to Resource3. The first step is illustrated in Figure 2.8. Eventually, the original instance is not in use and we can finally dereplicate the original instance of C2, thus finishing the migration. The final result is depicted in Figure 2.9.

**Figure 2.9:** The final result of a Migration operation. For step one, see Figure 2.8.
Chapter 3

SLAStic.SIM Architecture and Integration

In this chapter we describe the basic architectural and design decisions of SLAStic.SIM and its integration into the SLAStic framework.

3.1 SLAStic Architecture

In this section we describe how the simulator is embedded into the SLAStic framework.

The SLAStic framework consists of several components. Figure 3.1 provides an overview of the framework’s components and their assembly. Currently it consists of three tiers, SLAStic.SIM, a middleware (SLAStic.MON and SLAStic.REC) and SLAStic.CONTROL.

SLAStic.SIM is connected to a LogReplayer instance, which is responsible for driving the simulation by replaying previously recorded workload. As SLAStic.SIM produces Kieker traces, it is connected to a TpmonController instance, which itself writes the monitoring data to a pipe from which they can be read by a Tpan instance. The Tpan instance is connected to a SLAStic.CONTROL instance, which checks for the satisfaction of SLAs, bottlenecks or low hardware utilization. If SLAStic.CONTROL discovers a potential optimization, it generates a reconfiguration plan. This plan is fed to the ReconfigurationManager, which is responsible for updating the model according to the reconfiguration plan and notify the simulator of a plan’s arrival via the ReconfigurationPipe. SLAStic.SIM reconfigures the internal simulation model according to the operations defined in 4.3. This is done asynchronously, as some of the operations depend on the evolution of the system, and thus need simulation time to pass.
Figure 3.1: Overview of the SLAStic architecture
3.2 SLAStic.SIM Architecture

The architectural model of SLAStic.SIM is given in the form of a component diagram in Figure 3.2.

SLAStic.SIM itself implements the IKiekerRecordConsumer and IReconfPlanReceiver interfaces. The first one is used to feed the SLAStic.SIM with workload. It is connected to replayer for Kieker execution records. The second interface is connected to a reconfiguration plan sender and is used to interact with internal model manager, which is responsible for reconfiguring the internal simulation model.

3.3 Interaction with SLAStic.SIM

The simulation itself is driven by external workload. The SimulationController←→ is a facade GHJV95 to the simulator, implementing the interfaces IReconfPlanReceiver, which is used to deliver reconfiguration plans to the simulation, and IKiekerRecordConsumer, which drives the simulation by receiving workload from Kieker (see Figure 3.3).

The image shows how the SimulationController acts as a facade by implementing the two interfaces. It has to synchronize with two different threads. One of them is the simulation and the other one is the LogReplayer, which accesses the controller via the IKiekerRecordConsumer interface and feeds the simulation with workload. The synchronization is achieved by creating ExternalCall instances and storing them in the ExternalCallQueue instance. The ExternalCallQueue←→ is a synchronized queue implementation that fulfills two constraints. On the one hand it blocks the consumer, i.e., the DynamicSimulationModel if there is no
3.4 SLAStic.SIM Extension Points

SLAStic.SIM provides several extension points where implementations could be exchanged easily. These extension are discussed in throughout this section.

3.4.1 Load Balancer

Load balancers can be implemented by implementing the LoadBalancer interface. This interface declares the getServerMapping(String, Collection<String>) which takes an identifier for the assembly context to look up a server on which it
is allocated and a collection of identifiers, denoting the possible choices. As the choices may vary between two look ups for the same assembly contexts, implementing strategies that use information about previous look ups have to store all necessary information in the load balancer instance itself. Possible extensions are also discussed in Section 7.3.1.

3.4.2 Scheduler
All hardware schedulers within SLAStic.SIM inherit the AbstractScheduler class. By extending this class it is simple to build new schedulers. The AbstractScheduler class is a generic class which takes a ProcessingRessource as its parameter. Four abstract methods are declared. The two most important are described in the following. The first one is the schedule(SchedulableProcess) method, which has to add the process to a process an internal data structure for being able to eventually schedule it. The second one is the SimTime tick() method. This method is invoked by a dedicated TickEventGenerator and simply asks the scheduler for a time at which this method has to be called again. Within this method the processing of events has to be modeled.

3.4.3 Reconfiguration Events
Currently we implemented only the most necessary reconfiguration events. But as we build an abstract class ReconfigurationEvent, which is extending the Desmo-J ExternalEvent class. It only adds a constructor argument, the ReconfigurationOperation defined by the SLAStic reconfiguration meta model.
Chapter 4
Simulation

In this chapter the implementation of the simulation is described. We start by introducing our simulation model and define how it is interpreted during simulation time. Then we show how the SLAStic reconfiguration operations are implemented in SLAStic.SIM. The workload input is described afterwards. The chapter is finished by a characterization of the monitoring instrumentation.

4.1 Simulation Model

The simulation model is the main part of our simulator. It is build from a PCM model (see Section 2.2) and uses a similar layered structure.

Figure 4.1 shows the ModelManager and the contained components. The ModelManager is managing the dynamically built simulation model. It is implemented using the singleton pattern [GHJV95], hence making sure, there is only one instance of it. It contains different other managers, each of which holding and controlling different parts of the simulation model. All components contained by the ModelManager are also implemented using the singleton pattern.

The ComponentController is responsible for managing the PCM repository model (see Section 4.1.1), which is not changing during a simulation run. The PCM system model is managed by the AssemblyController. This model contains information about which components are instantiated and how they are interconnected (see Section 4.1.2). The HardwareController holds the PCM resource environment model and manages the state of servers or usage of resources (see Section 4.1.3). The AllocationController maps instantiated components (i.e., AssemblyContexts) to servers (see Section 4.1.4). It is also responsible to balance the load for replicated components. The AllocationController thus holds the PCM allocation model. Finally, the ReconfigurationController manages the execution of the runtime reconfiguration operations (see Section 4.3). It does not map to any PCM model, as reconfiguration is not implemented in PCM. The interface IReconfigurationPlanReceiver is implemented by the ReconfigurationController and the ModelManager delegates the calls to the dedicated controller.
These components are further described in the Sections 4.1.1–4.1.4, except for the reconfiguration controller, which is described in detail in Section 4.3.

### 4.1.1 Component Repository Model

In the component repository model all the components specified in the PCM input model are transferred into simulatable components. This means, that we have to transfer the given RDSEFFs defining the components’ provided services into a model that is simulatable by Desmo-J. The RDSEFFs are stored in a hash table for each component’s services. This makes it possible to look them up easily during runtime.

The RDSEFFs describe a control flow graph. In order to interpret a service call during simulation we have to find a path through the corresponding RDSEFF graph.

#### 4.1.1.1 Branches and Loops

As PCM provides the possibility to use probabilistic branches and loops, the control flow cannot be evaluated in a static way because each service call can result in a different chain of execution. Thus we have to evaluate the RDSEFFs during the simulation and generate a control flow for each service call. For
branches we evaluate which branch transition occurs and then recursively walk
through the branch transition’s body, as it is an RDSEFF as well. Loop bodies
are handle in the same way.

Loops are then unfolded to a control flow chain of their contained actions. The
generated control flow is then saved for execution at call time.

4.1.1.2 External Calls

We need to capture external calls and their return events, because these are sub-
ject to monitoring. We do this by adding a simple stack structure to each control
flow. The stack’s elements contain the simulation time at which the external ser-
vice was called, the server node on which the called component resides, the name
of the called service and the execution order index. This way it is possible to
generate the monitoring data directly from a stack frame.

4.1.1.3 Internal Actions

Internal actions have to be mapped to hardware for execution. They are sched-
uled according to their attached resource demanding behavior to the needed
resources (see Section 4.1.3) and notify the CallHandler on return. Control flow
nodes of internal actions contain a set of resource demands, which is built during
control flow evaluation. A resource demand simply consists of a resource type
and a demand. Depending on the type of the resource, the demand unit is pre-
determined. For CPU demands, the unit is cycles and for hard disk demands it
is the number of bytes to be transferred. Currently PCM makes no difference
between read and write operations, also the number of cycles is fixed, i.e., PCM
does not consider techniques like pipelining or even streaming SIMD extensions
(SSE). Using these extensions also implies, that the instruction set architecture
of the CPU has to be modeled and of course we need to know the concrete imple-
mentation of a service, thus losing abstraction. This way it would not be possible
to achieve the goal of fast simulation.

4.1.2 System Model

In the PCM system model components can be mapped to an assembly context,
which means that this component is instantiated as an assembly context. These
assembly contexts are connected to each other in order to provide the systems
functionality.

In our simulation model the mapping of components to assembly contexts and
their connections to each other are stored in hash tables. This way it is possible
to look up which component is instantiated by a specific assembly context and
vice versa. This information is needed in order to determine which service’s
implementation is actually called, i.e., which RDSEFF has to be evaluated, if a call occurs. Also, we store externally provided interfaces, i.e., system’s interfaces, so that we can look up which services are called by actors during simulation.

### 4.1.3 Resource Environment

The resource environment specifies the hardware infrastructure on which the simulated system executes. It contains at least one resource container, i.e., at least one server. According to the PCM, the server’s processing resources are divided into CPU, HDD, and networking and its behavior has to be specified in order to be able to simulate a given system. In our simulation model, a server object is created for each resource container. The processing resources are then added to these servers.

For scheduling processes we have designed a flexible framework. Each processing resource has its own scheduler, which is created when the resource itself is instantiated. A class diagram of the hardware model is shown in Figure 4.2.

A process to be scheduled is assigned to a processing resource by adding it to the resource’s queue. The scheduler then decides which process is scheduled next and how much processing time is given to that process before it is interrupted. Then it schedules an event using the `TickEventGenerator`, marking the interruption of the process and continues with the next process. If a process is finished, the internal action which it belongs to is informed so it can react accordingly, i.e., wait for other processes to return or inform the `CallHandler` to proceed with the next event in the control flow (see also Section 4.2.2).

We implemented a simple processor sharing scheduler [Tan07] for CPU scheduling and a first-come-first-served scheduler for I/O tasks as proofs of concept.

As resource containers can be allocated and deallocated during runtime (see Section 4.3) but the pool of resource containers is assumed to be static, we have to be able to mark the resource containers to either being available or unavailable. Marking a resource container to be unavailable can only happen if there is no component’s instance allocated to it. This will be assured by the `AllocationController` (see Section 4.1.4).

### 4.1.4 Allocation

The PCM allocation model maps assembly contexts to resource containers using allocation contexts. Thus, in our simulation model, we have to capture this mapping but also have to keep in mind that a replication, dereplication or migration operation can change this mapping during runtime.

In our allocation model we store which assembly context is mapped to which servers. For this purpose, a hash table that maps a given assembly context’s identifier to a set of resource containers, which must not be empty is used. This
4.1 Simulation Model

Figure 4.2: A class diagram of the hardware model including schedulers
way we can assure that an assembly context is mapped to at least one server node. We also store the opposite mapping, i.e., which assembly contexts are mapped to a specific server. This information is needed when a deallocation of server node is requested by a reconfiguration plan as a server can only be deallocated if there are no component’s instances mapped to it.

If there exist multiple mappings of one Assembly Context to different hardware resources we have to decide which instance is actually called. To meet this requirement we implemented a load balancing framework and implemented a random strategy, i.e., a random instance is chosen when querying the balancer. This behavior can easily be changed by extending the LoadBalancer class.

In order to be able to dereplicate a component we also have to be able to count users of a given allocated component. Dereplicating a component’s instance shall only occur if there is no running transactions using a service of the instance that is to be dereplicated. Therefore, if a component is to be dereplicated, we mark it blocked. If an instance is blocked the LoadBalancer must not dispatch any further service requests to that instance.

### 4.2 Simulation

For driving the simulation, we implement the IKiekerRecordConsumer interface and use a blocking queue to manage the producer-consumer problem.

The records are then filtered and queued if one of the following two conditions hold. Otherwise the LogReplayer is blocked.

1. The estimated return time of the first element in the queue is larger than the element that is to be added

2. The size of the queue exceeds a certain constant (PRE_BUFFER that to be specified in advance in the Constants class

The estimated return time is extracted from the incoming workloads return times. The PRE_BUFFER value must not be less than 1. This way we can assure that the simulation does not stop due to missing workload. The log reader can also send a special termination record to mark that there is no more workload available from the current source. This will set a termination flag in the external call queue and will not block fetching operations anymore. Rather it returns null to signalize the simulation engine that there no call to schedule anymore. This will cause the simulator to wait for the remaining calls to return and afterwards finish the simulation.
4.2 Simulation

4.2.1 Handling Calls and Control Flows

External calls from the input workload are handled by a dedicated class, the CallHandler. Initially, when the simulation is started the complete queue is scheduled at once, while it is not filled in the meantime. Scheduling a call is done in a small sequence of steps. First the service and its component are determined. A stack frame is pushed to a new stack. Then the control flow is evaluated and a list of events is stored. At last the simulation time at which the entry level system call has to occur is calculated and the first event of the control flow is scheduled for simulation.

The control flow only consists of internal actions, external calls and external call returns. These three types and their event routines are described in the Sections 4.2.1.1-4.2.2.

4.2.1.1 External Call

An external call event is partitioned into five basic steps.

1. The load balancer is queried for a server the service runs on. This involves the AllocationController, as this component stores, which components, or rather assembly contexts, reside on which server.

2. The AllocationController is queried again to add a user to the called component's instance.

3. The current simulation time and the execution order index is saved.

4. A stack frame is pushed for the call, containing information about the server, the service, the component’s instance and the entry time.

5. The CallHandler is told to schedule the next event. The time at which the event is scheduled is the current simulation time.

4.2.1.2 External Call Return

A return from an external call is much simpler than its entry. It involves two steps.

1. Save the current simulation time as the return time.

2. Pop the stack frame and generate a monitoring record.

4.2.1.3 Control Flow

As the control flow is evaluated on arrival of a system call event, it is not necessary to evaluate it during simulation time. The same applies to loop iterations, as loops are unfolded on control flow evaluation.
4.2.2 Simulation of Internal Actions

The simulation of internal actions is a crucial task for SLAStic.SIM as the simulated system’s performance behavior emerges from the resource usage simulation, which is defined by internal actions. The evaluation of an internal action is done using the following steps.

1. Evaluate the action’s resource demands (e.g., CPU cycles needed).
2. Determine current the allocation context from trace’s stack peek (i.e., the server the resources are demanded on).
3. For each resource demand:
   a) Create schedulable process, our internal data structure.
   b) Assign process to resource by adding it to the processor’s queue.
4. Wait for all processes to return.

The processes are then handled by the resource’s scheduler, which is responsible for simulating the processing of the processes. In the following, we describe how a CPU process is scheduled, using our implementation of the processor sharing scheduler. The activity diagram for the processor sharing scheduler is given in Figure 4.3 The basic algorithm works as follows:

1. Fetch process from the scheduler’s process queue.
2. If there is no process available:
   a) Wait for a process to arrive.
3. Else:
   a) Calculate the remaining processing time from its resource capacity (i.e., cycles per second) and the cycles that are needed to complete the processing of the process.
4. If the calculated processing time is greater than the scheduler’s time slice:
   a) Schedule tick event to be triggered in the time slice of the scheduler, marking the interruption of the process.
5. Else:
   a) Schedule tick event at calculated processing time.
6. Mark process active.
7. Wait for tick event.
8. Subtract cycles from process.
9. If the process is still incomplete:
4.3 Reconfiguration

In this section we describe how the reconfiguration plans and operations are implemented in SLAstic.SIM.

4.3.1 Handling Reconfiguration Plans

In order to handle reconfiguration plans correctly, we implemented a dedicated component for this task, the ReconfigurationController. Reconfiguration plans can be sent to SLAstic.SIM using the reconfigure(ReconfigurationPlan) method. It handles one plan at a time, i.e., any new plans arriving are discarded. To provide an infrastructure to check whether a reconfiguration plan is active, the ReconfigurationController implements the listener pattern. Listeners can subscribe via the addReconfigurationEventListener(ReconfEventListener) method. Listeners are notified on one of the following conditions:

a) Re-queue it to the process queue.
10. Else:
   a) Notify the internal action the process belongs to.
Chapter 4 Simulation

- Successful completion of a reconfiguration operation from the active plan
- Error because of pre- or post condition mismatch for a given reconfiguration operation, which will also discard the rest of the active plan
- Successful completion of the whole active reconfiguration plan

The ReconfigurationController serves as an interface to the simulation maintaining the correctness of the time (i.e., the synchronization of simulation time to the simulated time) and the sanity of the simulation model. The sanity of the model is guaranteed by the specification of suitable pre- and post conditions of the reconfiguration operations.

4.3.2 Replicate Component

The replication of a component means that a new instance of a given allocation context is spawned from a given assembly context and a resource container’s id. We do not allow for duplicate allocations of the same assembly context to one server. The allocation is done in several steps, all of which have to be successful for a successful replication of the given assembly context. An activity diagram of this operation is given in Figure 4.4.

1. Determine if the given server is available
2. Determine if the given assembly context is not allocated to the given server
3. Add a new allocation context to the system using the AllocationController

![Figure 4.4: Activity diagram of the replicate component operation](image)

The AllocationController has to maintain the consistency of the hash tables, i.e., add the assembly context to the values of the given server’s component and add the server to the values to the look up table of the given assembly context.
4.3.3 Dereplicate Component

The dereplication of a component means, that a given allocation context will be removed in a consistent manner from the system and the given assembly context will eventually no longer be allocated to the given server. As we forbid for duplicate assembly contexts allocated to the same server, the given allocation context can be unambiguously identified in the simulation model if it exists. The consistency conditions for this operation are: No transactions will be interrupted, i.e., we can only dereplicate unused assembly contexts and at least one assembly context of the given type has to remain in the system to guarantee the system’s functionality. The algorithm dereplicating an assembly context works as follows. An activity diagram of the dereplication operation is shown in Figure 4.5.

1. Check for existence of the given allocation.
2. Check if at least one allocation context will be available in the system after dereplication.
3. Mark the allocation context blocked, i.e., no new transactions will be accepted.
4. Wait for all calls to the allocation context to return.
5. Delete the allocation context from the system.

![Figure 4.5: Activity diagram of the dereplication operation](image)

The blocking of the assembly context is done by the AllocationController and involves the LoadBalancer. The LoadBalancer does not dispatch calls to the blocked assembly context. The final deletion of the allocation is then guaranteed to meet the consistency condition specified above.
4.3.4 Migrate Component

The migration of an assembly context from one resource container to another means that the component will simply be moved in a consistent manner. Consistent means that no transaction are broken by the migration. Upon completion of the operation, the given assembly context will be non-existent on the originating resource container and allocated to another, i.e., the number of allocations is not influenced by this operation. We see this as a post condition rather than an invariant to meet the consistency specifications. Also this enables us to model the migration as a composition of a replication and a dereplication operation in two steps. An activity diagram of the migration operation is given in Figure 4.6.

1. Replicate assembly context to the target resource container
2. Dereplicate assembly context on the originating resource container

![Figure 4.6: An activity diagram of the migration operation](image)

Each of these steps have to be successful for a successful migration of a component. In particular this also means, if the dereplication of the originating assembly context fails for some reason, we have to dereplicate the assembly context created in step 1. Otherwise the number of assembly contexts would have increased and thus the system’s state would have changed in an unintended way.

4.3.5 Allocate Resource Container

Allocation of a resource container means that a given unavailable resource container will be available after the operation took place. We do this by simply marking a Server in the simulation model available by setting a boolean flag.
4.3 Reconfiguration

Afterwards the resource container is empty and available, hence replication or migration operations can target it. We implemented this operation in three steps. An activity diagram of the allocation operation is given in Figure 4.7.

1. Check for existence of the given resource container
2. Check if the resource container is unavailable
3. Mark the resource container available

![Activity diagram of the allocate resource operation](image)

**Figure 4.7:** Activity diagram of the allocate resource operation

The HardwareController is responsible for maintaining the availability of resource containers. As before, all the steps have to be successful for a successful completion of the allocation operation.

### 4.3.6 Deallocate Resource Container

Deallocation of a resource container means that a given resource container has to be unavailable, i.e., cannot be used for deployment, on a successful completion of the operation. So we have to mark the given resource container unavailable in a consistent manner. Consistent here means that the system’s functionality must not be affected, for instance marking a resource container unavailable that has assembly contexts allocated to it, thus rendering these assembly contexts unavailable in an unintended way. To avoid scenarios like this, we only deallocate empty resource containers. The operation is divided into three steps. An activity diagram of the deallocation operation is given in Figure 4.8.

1. Check for existence of the given resource container
2. Check if no assembly contexts are allocated on the given resource container
3. Mark resource container unavailable

![Activity diagram of the deallocate resource operation](image)

**Figure 4.8:** Activity diagram of the deallocate resource operation
Again the HardwareController has to maintain the availability of the resource container. The AllocationController is queried for the emptiness check in step 2. If all operations finish successfully the deallocation itself was successful, too.

4.4 Monitoring

In order to evaluate the performance of the simulated system models, it has to be monitored. Monitoring the simulated system is done using the Kieker monitoring framework. As mentioned in Section 4.1.1, ExternalCallEvents are subject to monitoring. For each ExternalCallEvent we generate a monitoring record on its return from the stack frame that is pushed when entering the call. The stack frame contains information about the called service, the component the called service belongs to, the server the called service’s component resides on, the entry time, and the execution order index. On return the information is completed by the return time and the stack depth. The execution order index is the index of the external call in the control flow regarding only ExternalCallEvents.

The generated KiekerExecutionRecord is then passed to the TpmonController, which is responsible to persist the record.

4.5 Workload

Workload is input to the simulator by reading Kieker logs. As these logs can contain whole control flow traces and just a user’s service demand, we have to filter the input for external service calls, which can be seen as user interaction. This is done by just using the calls that mark the beginning of a trace (i.e., eoi→
4.5 Workload

\( == 0 \&\& \text{ess} == 0 \), see also Figure 4.9, where eoi is the execution order index starting at 0 and ess is the execution stack size).

These calls serve as input for the CallHandler, which is responsible for scheduling them. If there is no call available in the input queue the simulation is paused, if there are calls to be read, or a shutdown marker is set, which causes the simulator to shutdown after the return of the last simulated call.
Chapter 5

Evaluation

In this chapter we evaluate the quality of SLAStic.SIM. At first we define our evaluation goals. Then we describe our simulation environment, i.e., the hardware and software setup. After that we present our evaluation model, which will be used to evaluate SLAStic.SIM against our evaluation goals. Finally, we evaluate SLAStic.SIM using various workload and reconfiguration scenarios.

5.1 Evaluation Goals

In this section we define different quality metrics for evaluating SLAStic.SIM. These goals are evaluated using different experiments (see Section 5.5–5.7).

5.1.1 Validation

A simulation is valid if a simulated call sequence produces performance data equal to the performance data of a real system. Due to abstraction, we demand that the performance results have to be similar to each other. This is often referred to as credibility [Ban98]. The validity of SLAStic.SIM is evaluated

1. by comparison with a real system in Section 5.5
2. by comparison with SimuCom in Section 5.6
3. by comparisons with our expectations in Section 5.7

5.1.2 Verification

Verification means that the entity flow in the simulation model is equal to the entity flow in the real system [Ban98]. This means that calls in the real system have to produce the same control flow sequences as a simulated call in SLAStic.SIM.

Due to the lack of a mathematical transformation and specification of the simulation model, we are not able to verify the simulation formally. Instead we chose to compare the generated control flow chain of the implemented system with the one generated by SLAStic.SIM (see Section 5.5).
5.1.3 Performance

SLAstic.SIM’s performance is crucial to its usability. If the simulation was slower than the execution of the modeled system, simulation would not be feasible for performance evaluation. We are only interested in the computation performance, omitting the memory overhead. This goal is analytically investigated in Section 5.2.

5.2 Performance and Scalability

In this Section we investigate the performance goal analytically. As mentioned in Section 5.1.3, one of our goals is that the simulated time is not advancing slower than the real world’s time which would be needed to actually run the system. However, this goal is not achievable for all systems. Imagine a large scale system, running on many servers, using load balancing and component replication. As long as we simulate only one service request, the simulation can be faster. But when simulating a huge number of transactions, each user can have an own server to calculate her service request, in contrast to SLAstic.SIM, which runs only one process, regardless how many transactions are actually simulated.

Let $p_t(e)$ be the processing time of an event $e$ including the overhead of the DesmoJ event scheduling algorithm, $n$ the number of simulated user’s service requests and $m_i$ with $1 \leq i \leq n$ the number of events, the service request with index $i$ produces. Then the overall simulation time $st$ is given by:

$$st = \sum_{i=1}^{n} \sum_{j=1}^{m_i} p_t(e_{ji})$$

To analyze the complexity class of the simulation, we assume, there is only one service. This simplifies the above equation to the following.

$$st' = n \cdot \sum_{j=1}^{m} p_t(e_j)$$

Now we assume all events to have the same computation time. This leads to a computation time of

$$st'' = n \cdot m \cdot p_t(e)$$

So our system scales linearly to the numbers of simulated transactions, as the number of produced events is bounded and can thus be regarded as a constant in this analysis.

For a real world system we have $p_t(c_k)$ as the processing time of a call on a server $k$, again $n$ as the number of transactions, $m_i$ with $1 \leq i \leq n$ is again the number of calls that are produced by the service request $i$. Let now $s$ be the
number of servers, then the following equation will be the overall computation time $ct$.

$$ct = \left\lceil \sum_{i=1}^{s} \sum_{j=1}^{m_i} pt(c_{m_j}) \right\rceil$$

With $s = n$, which would mean that for each transaction in the system, there is one computing node available, we get the following formula.

$$ct' = \sum_{j=1}^{m} pt(c_{m_j})$$

Assuming again that the computation time of all calls is the same, the overall computation time is.

$$ct'' = m \cdot pt(c)$$

As in the simulated system, the number of calls produced by an entry level system call is bounded, i.e., $m_{\text{max}} = \max\{m_i\}$, with $1 \leq i \leq n$. This means, that in our best case calculations for the real system, it can be replaced by $m_{\text{max}}$, yielding a constant calculation time in the best case.

Nevertheless, as we simulate gray-box models and it is nearly impossible to have one available computation node for larger number of transactions, it is unlikely that a simulation of a system needs more computation time than the execution of the real system (see Section 5.6 and Section 5.7 in which we evaluated the performance of SLAStic.SIM using resource intensive transactions).

5.3 Simulation Setup

In this section we describe the setup of our simulation environment.

5.3.1 Hardware

We used the hardware given in Table 5.1 to evaluate SLAStic.SIM. In this table we omit unimportant components like the graphics interface or network connectivity.

<table>
<thead>
<tr>
<th>Component</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel® Core™ 2 Quad Q8200 @ 2.33GHz (throttling disabled)</td>
</tr>
<tr>
<td>RAM</td>
<td>4GB Corsair DDR2 @666MHz</td>
</tr>
</tbody>
</table>

Table 5.1: Evaluation Hardware
5.3.2 Software

The simulation hardware runs a Gentoo Linux distribution with a preemptive low latency symmetric multiprocessing kernel version 2.6.28-gentoo-r5. All experiments are run from an xterm instance using ant (for SLAstic.SIM) or an Eclipse instance (for SimuCom). The Java runtime environment used is Java(TM) SE Runtime Environment (build 1.6.0_17-b04).

In all experiments we used an overall heap size of 2048 Mega Bytes for the java virtual machine (−Xmx2048M) running either SLAstic.SIM or the Eclipse instance used to start SimuCom.

5.4 Bookstore Example

In this section we evaluate SLAstic.SIM using a simple software architecture, the bookstore model as described by Stöver [Stö09]. At first we describe the model and the hardware used for evaluation. Then we evaluate SLAstic.SIM using different workload scenarios and references using the defined quality metrics.

5.4.1 Repository

The bookstore’s repository contains three interfaces and three components, each of them providing an interface. It is depicted in Figure 5.1. The interface IBookstore declares the service searchBook(). It is implemented in the component Bookstore. The implementation requires services declared by the interfaces ICatalog and ICRM respectively. The interface ICRM declares the service getOffers() and is implemented by the CRM component which also requires the services declared by the interface ICatalog. This interface is provided by the Catalog component and declares the service getBook().

The RDSEFFs of the components’ implementations of the declared services are shown in Figure 5.2-5.4.

In Figure 5.2 the service searchBook() is shown. It contains a probabilistic loop which loops 15 times through the loop’s body. During the loop’s execution, at first the service getBook() from the required interface ICatalog is called. After the call returns, another external call to the service getOffers(), declared in the interface ICRM, is done. Finally an internal action occurs to model some CPU usage (1000 cycles) of the service searchBook. In this model, it is assumed that calling a service does not need any resources itself.

In Figure 5.3 the service getOffers() is depicted. It only calls the external service getBook() and does not feature any resource demanding internal actions.

In Figure 5.4 the service getBook() is shown. It consists only of an internal action, which models the resource usage of this service. The resource demand is
Figure 5.1: The Bookstore Repository
Figure 5.2: The service `searchBook()` as implemented by the component Bookstore.

Figure 5.3: The service `getOffers()` of the component CRM.
specified to be a CPU usage of 100 cycles. After simulating the internal action, the service returns to its caller.

![Diagram of CPU resource demand](image)

**Figure 5.4**: The service `getBook()` of the component `Catalog`  

### 5.4.2 System Model

In Figure 5.5 the system model is given. The only externally callable service is `searchBook`. Internally the components are instantiated and connected to each other according to their usage specification from the repository.

![Diagram of Bookstore system model](image)

**Figure 5.5**: The Bookstore system model
5.4.3 Resource Environment

The resource environment used in this model consists of two resource containers, Server1 and Server2 which are equal to each other regarding their processing resources. As the RDSEFFs do not describe any HDD usage, it is sufficient to model only the CPUs. Each resource container features one CPU with a capacity of 1000 units per simulated second. As a scheduling strategy we chose processor sharing as it is implemented in both, SimuCom and SLAStic.SIM.

5.4.4 Initial Allocation

The Initial allocation of the PCM bookstore model is given in Figure 5.6. All assembly contexts shown in Figure 5.5 are allocated to Server2. This has the effect that Server1 is not allocated initially in the simulation model of SLAStic.SIM.

![Figure 5.6: The initial allocation model of the Bookstore](image)

5.5 Single Call Experiment

The goal of this experiment is to verify the simulation model. This is done by comparison of the simulated call sequence to the real system’s call sequence.

5.5.1 Workload

The workload for this experiment is a single call to the searchBook() service of the Bookstore component.

The input workload has been previously recorded by monitoring a simple implementation of the model. In that implementation, resource usage was modeled
by using the standard library method `Thread.sleep()`. Unfortunately the sleep of the calls is parallelized, hence it is not possible to evaluate scheduling using this system. Therefore, we use this scenario only for validating a single call to the service `searchBook`.

### 5.5.2 Verification

As we did not specify an algebraic transformation between real world systems, Palladio models and our simulation model, it is not possible to verify SLAStic.SIM mathematically. Hence we have to compare the call sequences of generated by the real system to the ones simulated by SLAStic.SIM. From the sequence diagrams shown in Figure 5.7, we can see that they are equal. Both consist of a call to `searchBook()` and generating the same call sequence. The left object in the sequence diagram represent the actors. They call the `searchBook()` method on the `Bookstore` component, which is depicted as the second left life line. From there, `getBook()` in the `Catalog` component is called (second life line from the right). After return, the `getOffers()` service is invoked, which is provided by the `CRM` component (right life line).

### 5.5.3 Response Time Comparison to Real System

The response time to the entry level system call in the bookstore implementation was measured to be approximately 3.05 seconds.

The simulated response time of a single call to `searchBook()` was 3.015 seconds. This is close to the time in the implementation and thus a credible result [Ban98].

### 5.5.4 Performance

The bookstore implementation needed around 3.2 seconds to execute. The execution time is measured from the entry of the `main` method to its return, hence not including static overhead like class loading and AspectJ load time weaving.

SLAStic.SIM needed only around 0.032 seconds to simulate the bookstore example model. This time is measured from the start to the end of the simulation. So it does not include static overhead like model parsing and transformation either. Comparing the two execution times we can see that simulating in this case needs less computation time than executing the system. To be more exact, the speedup is around 100 for this scenario. Unfortunately these results are not generalizable.
Chapter 5 Evaluation

(a) Sequence diagram of a single simulated call
(b) Sequence diagram of implemented bookstore system

Figure 5.7: Two sequence diagrams generated with Kieker.Tpan
5.6 Multi Call Experiment

The goal of this experiment is to compare SLAStic.SIM to SimuCom.

5.6.1 Workload

The input workload for SLAStic.SIM has been previously generated using the generator script given in Appendix A. It features 700 calls with an interarrival time of one simulated second. For SimuCom we used the open workload scenario given in Figure 5.8.

![Workload scenario used to drive SimuCom](image)

To preserve the validity of the results generated by SimuCom, we chose to configure the SimuCom run at 700 simulated seconds. Simulating more than 750 seconds using the setup described in Section 5.3 caused a `TooManyUsersSpawnedException` to be thrown or SimuCom froze completely.

5.6.2 Validity

In order to validate the performance results of SLAStic.SIM, we generated a time series diagram (given in Figure 5.9) and compare it to the time series plot of SimuBench (given in Figure 5.10). Unfortunately, SimuBench does not allow for plotting response times by entry time. Also the response times are measured in simulated seconds, in contrast to SLAStic.SIM’s plots, which give response times in milli-seconds. Anyhow, as the intervals between two arriving calls is 1 simulated second in both workloads, the results are still comparable.

As we can see from these plots, SimuCom and SLAStic.SIM’s measurements are very similar. This result and the result from Section 5.5 means that we have met the requirement of implementing a valid simulator for static PCM models.
Chapter 5 Evaluation

Figure 5.9: Time series plot of response times of the service `searchBook` for the multi call experiment performed with SLAStic.SIM

Figure 5.10: Time series plot of response times of the service `searchBook` for the multi call experiment performed with SimuCom
5.6.3 Performance

The performance of the simulator is measured in nanoseconds between the start and the end of the simulation omitting static overhead like model parsing, code generation and compilation. SLAStic.SIM needed 0.98 seconds to simulate 700 seconds of the bookstore example. Compared to SimuCom, which needed 2.68 seconds we achieved a speedup of approximately factor 2.5.

5.6.4 Comparisons to SimuCom

In this Section we discuss problems that arose during our evaluation against SimuCom.

5.6.4.1 Workload

In PCM, it is not possible to use previously generated workload. This would not be a problem, if it was possible to define workload as functions of time. Even closed workload scenarios are not feasible to model workload scenarios like ours, as SimuCom spawns all users at once. This means that the response times are constant if the scenario is deterministic.

5.6.4.2 User Capacity

Unfortunately, it was not even possible to simulate the described system using open workload with an arrival rate of 1 simulated second. SimuCom freezes in our setup at 750 simulated seconds, i.e., 750 entry level system calls. As modern software systems are often designed to serve more than 750 users over time, SimuCom might have to be adopted to serve this requirement in the future.

5.6.5 Conclusion

SLAStic.SIM serves a performant simulator for static PCM models. The experiment yielded the result that SLAStic.SIM is more performant than SimuCom in the sense, that it is faster. We did not evaluate memory performance. As we will see in Section 5.7 SLAStic.SIM outperforms SimuCom in the possible numbers of parallel users in the simulated system.

5.7 Multi Call Experiment with Reconfiguration

In this experiment we show the impact of a possible online reconfiguration. This is done by adding a reconfiguration plan, which consists of a series of replication and dereplication operations. We chose to present a representative run and omit
Chapter 5 Evaluation

the statistical analysis (i.e., standard deviation, confidence intervals, correlation, etc.).

5.7.1 Workload and Reconfiguration

For this experiment, we generated workload using a simple script (see Appendix A). We generated workload data for 11000 transactions, calling the `searchBook()` service at intervals of one second.

Reconfiguration operations were statically scheduled in our simulation model’s `doInitialSchedules()` method. In order to show the impact of reconfiguration, we replicated and dereplicated the assembly context `Assembly_Catalog <Catalog>`. This assembly component contains the `Catalog` component, which defines the service `getBook()`. This service is defined to demand much CPU time, thus replication makes perfect sense. We scheduled a series of replication and dereplication operations. The times at which they were scheduled are given in Table 5.2.

<table>
<thead>
<tr>
<th>Replicate</th>
<th>1500</th>
<th>4500</th>
<th>7500</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dereplicate</td>
<td>3000</td>
<td>6000</td>
<td>9000</td>
</tr>
</tbody>
</table>

Table 5.2: Reconfiguration operations and the simulation time at which they occur

5.7.2 Comparison to Multi Call without Reconfiguration

By reconfiguring the system, we were able to reduce the response times dramatically. In Figure 5.11 we see a time series of response times without replication. If we compare this to the time series depicted in Figure 5.12, we can see a reduction of the maximum response times by approximately factor 5.7. Additionally, we can see, after replication, the response times are decreasing slowly. This is due to the random load balancing strategy we use. This strategy does not use any information about the current load of a server. The response times of the calls to the service `getBook` are given in Figure 5.13. Here we can see that the response time decreases upon scheduling the calls to the replicated components as well.

Anyhow, taking a look at the maximum of concurrent transactions in the system, we can see that we decreased this number by approximately factor 5.3, from 6461 to 1210.

In this experiment we also compare the weighted dependency graphs generated using Kieker.Tpan. The graph for the multi call experiment is given in Figure 5.14. From the graph we can see how often each service is called on each allocation context. Of course, without reconfiguration there is only one node
5.7 Multi Call Experiment with Reconfiguration

![Figure 5.11](image1.png)

**Figure 5.11:** Time series plot of the response time to the `Catalog.searchBook` service without replication

![Figure 5.12](image2.png)

**Figure 5.12:** Time series plot of the response time to the `Catalog.searchBook` service with replication and dereplication occurring at the time listed in Table 5.2
Chapter 5 Evaluation

Figure 5.13: Time series plot of the response time to the Catalog.getBook service with replication and dereplication occurring at the time listed in Table 5.2.

involved. In Figure 5.15 we give the dependency graph for the reconfigured system. In this graph we can see that the assembly context containing the Catalog is replicated to another node. This leads to the effect that roughly 29% of all calls to the service getBook are distributed to the replicate.

Figure 5.14: Dependency graph for the multi call experiment without reconfiguration.

5.7.3 Performance

To simulate the described scenario, we needed approximately 39 seconds (measured using time) and produced 506000 monitoring records. Simulating the same workload in a static system needs 39 seconds and produced 506000 records, too. This validates our calculations in Section 5.2 and shows that SLAStic.SIM has met the goal of fast simulation.
5.7.4 Conclusion

In this experiment we have shown that SLAStic.SIM is able to handle reconfiguration operation. We saw that replication is feasible for reducing response time of operations and that SLAStic.SIM handles it as expected. The results presented in this experiment setup differ from run to run, due to the probabilistic load balancing algorithm.

5.8 Simulating a Workload Curve

In this Section we simulate the bookstore example using a varying workload given by van Hoorn et.al. [vHRH08]. This experiment will show if SLAStic.SIM is capable of simulating the varying number of transactions in a credible way and how replication can be used in a system with varying workloads. Again we present a representative run and omit the statistical analysis.

5.8.1 Workload

We use a curve formed workload in this experiment. The workload curve has been originally used to generate markov4jMeter workload [vHRH08] and is depicted in Figure 5.16.

It is defined as the following function of time in minutes:

\[
numUsers(t) = 1.8 \cdot f(t) + 3.5 \cdot h(t) + 0.8 \cdot g(t) - 110
\]
where

\[ f(t) = 25 \cdot \sin\left(\frac{t-13}{3.2}\right) + 10 \]
\[ g(t) = 23 \cdot \sqrt{x + 2} \]
\[ h(t) = 10 \cdot \sin\left(\frac{x-1}{6}\right) + 30 \]

We interpret this function as the arrival rate sampled in minutes and implemented a small generator script for transforming its output into Kieker execution records (see Appendix A). Using this script we generated 30 minutes of workload for driving the SLAStic.SIM experiment.

### 5.8.2 Response Times

A time series of the response times of the service `searchBook` is given in Figure 5.17. These response times emerge from the call to the service `getBook`, for
which a time series is given in Figure 5.18. From this graph can see that the derivation is varying according to the workload curve. From a certain simulated time on, the response times are not decreasing anymore as too many transactions accumulate in the system and processing the accumulated service requests takes so much time time that it causes the system to accumulate even more and more transactions.

![Response time of the service getBook under varying workload](image)

**Figure 5.18:** Response time of the service `getBook` under varying workload

To overcome the accumulation we replicated the `Catalog` component at the simulation times 600, 800, and 1000 seconds. This yields much smaller response times as we can see from Figure 5.19.

Again for better comparability of the results we show a plot of the response times of the service `getBook` in Figure 5.20. We can see that system is still accumulating transactions but much less dramatically than without replication which results in much smaller response times of the service `getBook`, which also decreases the responses times of the service `searchBook`.

Figure 5.21 shows the operation dependency graph without replication. We can see that the curve produces 2628 calls to the service `searchBook` in is executed using only one server node. In Figure 5.22 the dependency graph for the reconfiguration experiment is given. Of course, the number of entry level system calls to `searchBook` are equal. Opposed to the non reconfigured system, we can see from the graph that the calls to the internal service `getBook` is well distributed to all four instances, hence decreasing the response times.
Chapter 5 Evaluation

Figure 5.19: Response times of the service \texttt{searchBook} under varying workload with replication.

Figure 5.20: Response times of the service \texttt{getBook} under varying workload with replication.
5.8 Simulating a Workload Curve

**Figure 5.21:** Operation dependency graph of the workload curve experiment without reconfiguration

**Figure 5.22:** Operation dependency graph of the workload curve experiment with reconfiguration
Chapter 6

Related Work

In this chapter we present work that is related to the topic of this thesis. We start with SimuCom, which is a simulator for Palladio Component Models distributed by the Palladio group. Then we introduce Java Modeling Tools (JMT), a tool suite for developing and simulating queuing networks. This is followed by an overview of ArgoSPE, which is a tool for Software Performance Engineering based on the UML SPT [OMG05]. Afterwards, we present an approach for performance modeling and simulation of service oriented architectures. Finally, we introduce MOSES, a methodology for modeling and simulation of UML 2 architectural software and platform models.

6.1 SimuCom

SimuCom is a simulator for Palladio Component Models [BKR09]. The simulation is actually generated using an openArchitectureware [EFH+08] model to code transformation from a given PCM model to Java code. The resulting simulation is built as an Eclipse OSGi Plugin which can then be started via the SimuBench. This gives SimuCom the advantage of being directly integrated into the modeling environment. A controller GUI starts the simulation after compilation and also cleans up the system after finishing. The simulator’s core is based on the Desmo-J discrete event simulation framework.

SimuCom served as a reference implementation for SLAstic.SIM. The target was to achieve a simulation that is semantically equivalent to the generated SimuCom simulation. This means that the performance results should not differ between a system simulated using SLAstic.SIM and SimuCom.

As one of our main goals was to implement a simulator for runtime reconfigurable software architectures, the generative approach was not feasible. Also extending the SimuCom framework by reconfiguration operations would have meant to regenerate the simulation code during simulation, saving the current state (i.e., user transactions, scheduler queue and simulation time) and then start the new simulation, which would have to be initialized with the aforementioned state. This means that is is not possible to simulate the transition between two
Table 6.1: Comparison between SLAstic.SIM and SimuCom

<table>
<thead>
<tr>
<th>Modeled systems</th>
<th>SLAstic.SIM</th>
<th>PCM and SimuCom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modeling</td>
<td>PCM + reconfiguration</td>
<td>PCM</td>
</tr>
<tr>
<td>Workload</td>
<td>Recorded or generated</td>
<td>Modeled</td>
</tr>
<tr>
<td>Output</td>
<td>Service response times</td>
<td>Response times, resource usage, throughput</td>
</tr>
</tbody>
</table>

configurations, which is not feasible for correct simulation of runtime reconfiguration.

Additionally, SLAstic.SIM should also be usable online, e.g., for evaluating different architectural configurations. Hence using an Eclipse GUI Plug-in for driving the simulation is not a viable solution. We decoupled the simulation of a given PCM model from the graphical user interface.

In contrast to feeding the simulator with recorded workload, one has to model workload using PCM workload models. It is possible to model different workload scenarios, but the only one of them can be used to drive the simulation. Currently SimuCom supports open and closed workload. These scenarios serve as WorkloadDriver and drive the simulation.

In Table 6.4 a rough comparison of some attributes is given.

### 6.2 Java Modeling Tools

The Java Modeling Tools (JMT) is a suite of loosely coupled graphical applications for performance evaluation [BCS09]. One of their methodologies for performance evaluation is discrete event simulation of queuing networks. In addition JMT features several analytic methods, e.g., for identifying bottle neck resources.

In JMT system’s are modeled as queuing networks using a graphical user interface (JSIMGRAPH) and wizards (JSIMWIZ for simplification of the modeling task. The models are persisted in XML. SLAstic.SIM’s simulation models, in contrast to JMT models, are generated from PCM models and focus on the architectural models of software systems. Also PCM uses different viewpoints to model a complete system, giving the advantage of the separation of concerns as each viewpoint focuses on a special domain.

JMT models can then be simulated by JSIMENGINE, which is a discrete event simulator. It supports several inputs for probabilistic workload distributions,
6.3 ArgoSPE

ArgoSPE is a tool-suite implementing the software performance engineering [WFP07] methods. It derives stochastic Petri nets (SPNs) models from models specified in ArgoUML [arg] using the the UML profile for Schedulability, Performance and Time (UMP SPT) [OMG05]. Models are serialized using XMI. Using the UML SPT profile, it is possible to formulate several so called performance queries on two different viewpoints of the model, state machine diagrams and collaboration and deployment diagrams [GMM06].

For state machines the possible queries are state population, which is defined to be the percentage of objects residing in a specific state, stay time, which is the mean time an object is in a specific state, and message delay, which is the defined to be the interval between sending a message and its receive. The message delay also involves the network delay in its computation.

For collaboration and deployment diagrams we can query for network delay, which computes the bit rate between two non adjacent hardware nodes, and the

<table>
<thead>
<tr>
<th>Modeled systems</th>
<th>Component based software architectures</th>
<th>Hardware or small software systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modeling</td>
<td>Graphical, different viewpoints</td>
<td>Graphical, one viewpoint</td>
</tr>
<tr>
<td>Workload</td>
<td>Recorded</td>
<td>Recorded or stochastic</td>
</tr>
<tr>
<td>Output</td>
<td>Response times</td>
<td>Response times, throughput, resource usage, etc.</td>
</tr>
</tbody>
</table>

Table 6.2: Comparison between SLAStic.SIM and JMT

which are implemented using the Mersenne Twister Engine. In addition to probabilistic workload, it is also possible to read previously generated or recorded workload from log files. The output of JSIMENGINE features several performance measures on the queuing network. Among them are utilization of resources, queue lengths, response times, and jobs in the system. JSIMENGINE provides more output than SLAStic.SIM does at the moment. But due to the high flexibility and extendability of Kieker, SLAStic.SIM can easily be extended to provide more detailed information about the simulated system, which is also discussed in Section 7.3.3. Like SLAStic.SIM, it is possible to use the simulation engine within external applications. In Table 6.2 we give a rough comparison between JMT and SLAStic.SIM.
Table 6.3: Comparison between SLAstic.SIM and ArgoSPE

<table>
<thead>
<tr>
<th>Modeled systems</th>
<th>SLAstic.SIM</th>
<th>ArgoSPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modeling</td>
<td>PCM</td>
<td>UML SPT</td>
</tr>
<tr>
<td>Workload</td>
<td>Recorded or generated</td>
<td>No workload, fixed UML SPT queries</td>
</tr>
<tr>
<td>Output</td>
<td>Response times</td>
<td>Results of UML SPT queries</td>
</tr>
</tbody>
</table>

response time for service requests. The response time here is the mean response time for a given service request, in contrast to SLAstic.SIM, where every single response time for each service request is monitored which makes it possible to calculate a mean, if it is desired.

For analyzing a given model, ArgoUML first creates a parametrized XMI file, which is then passed to the Model Configurer [GMM06]. The Model Configurer analyzes the parametrized XMI model and takes a configuration file to actually evaluate variables. The configuration files are written in Perl, thus making it possible to add very complex computations for any variable used in the performance annotations. The configured model is then analyzed by the model processor, which converts the model into a generalized stochastic Petri net (GPSN). The GPSN is evaluated and the results to the UML SPT queries are returned. In Table 6.3 we give a rough comparison between ArgoSPE and SLAstic.SIM.

6.4 Simulating Service Oriented Architectures with OMNeT++

Bause et al. [BBKV08] proposed a framework for simulating model of service oriented architectures (SOA) focusing on the quality of service of SOA systems. In their approach a model has to be built using two viewpoints. On the SOA level, ProC/B [BBF02] is used, which enables the modeler to define process chains for services and service orchestration. Also performance annotations for services are modeled here. For modeling the resources used by the SOA, OMNeT++ [omn] is used. Remote service calls in the ProC/B model are assigned to message transfers in OMNeT++.

For performance evaluation of the specified SOA, the model is transformed into a OMNeT++ simulation model which is then simulated by OMNeT++. It is also possible to visualize the simulation process, enabling a modeler to find
bugs in a simulation model using a graphical interface. The simulator outputs are response times of service requests and visualization of traces. The simulation is driven using a ProC/B model of the user input.

### 6.5 MOSES

MOSES is a methodology for performance evaluation of UML models. It is an abbreviation for "MOdeling Software and platform architecture in UML 2 for Simulation-based performance analysis" [CPSV08]. Models are defined using the UML 2, for software architectural models and platform models. The general methodology of MOSES is to build such models, merge them to obtain an integrated architectural model, which is then annotated with performance data and finally simulated [CG04; CPSV08].

In contrast to PCM, platform models for MOSES have to be built up by interconnecting platform components, such as dispatchers and CPUs (including schedulers). I.e., the deployment task is a bit more complicated than with PCM.

For simulating the models, the MOSES developers chose to use the UML 2 modeling and simulation tool Telelogic TAU G2 (now Rational Tau). TAU G2 is able to output sequence diagrams and execution logs and supports step by step simulation and visualization on state machines. The simulation engine, like SLAStic.SIM, uses discrete event simulation.

MOSES is only proposing tools, no dedicated applications have been developed so far. Thus, in order to use it, expensive third party software like Tau G2 has to be used for simulation and modeling.

In Table 6.4 we give an overview over several attributes of MOSES and SLAStic.SIM.

<table>
<thead>
<tr>
<th>Modeled systems</th>
<th>SLAStic.SIM</th>
<th>ProC/B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modeling</td>
<td>Graphical, different viewpoints</td>
<td>Graphical for process Chains, textual (NED language) for OMNeT++ resource models</td>
</tr>
<tr>
<td>Workload</td>
<td>Recorded or generated</td>
<td>ProC/B process chain</td>
</tr>
<tr>
<td>Output</td>
<td>Response times</td>
<td>Response times and traces</td>
</tr>
</tbody>
</table>

Table 6.4: Comparison between SLAStic.SIM and ProC/B
<table>
<thead>
<tr>
<th></th>
<th>SLAStic.SIM</th>
<th>MOSES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modeled systems</td>
<td>PCM</td>
<td>Arbitrary UML 2 models</td>
</tr>
<tr>
<td>Modeling</td>
<td>Graphical, different viewpoints</td>
<td>UML, different viewpoints</td>
</tr>
<tr>
<td>Workload</td>
<td>Recorded or generated</td>
<td>no information</td>
</tr>
<tr>
<td>Output</td>
<td>Response times</td>
<td>no information</td>
</tr>
</tbody>
</table>

**Table 6.5:** Comparison between SLAStic.SIM and MOSES
Chapter 7

Conclusions and Future Work

In this chapter we conclude our work and give possible points of extensions to SLAStic.SIM.

7.1 Summary

The foundations required for this work, including the Eclipse Modeling Framework, the Palladio Component Model, discrete event simulation, the Kieker monitoring framework, and the SLAStic approach for self-adaptive online capacity management have been stated at the beginning of the thesis. We implemented a performance simulator for runtime reconfigurable software systems, SLAStic.SIM, and described its architecture and integration into the SLAStic framework and designed and implemented the simulator itself. Furthermore, we evaluated SLAStic.SIM using various workload and reconfiguration scenarios.

7.2 Discussion

SLAStic.SIM is capable of simulating PCM models. It takes previously recorded or generated workload in the form of Kieker execution records and reconfiguration plans as input. Our evaluation shows that SLAStic.SIM reacts in the expected manner. It produces monitoring data via Kieker. The architecture is extendable at several points and integrates well into the SLAStic framework for self-adaptive online capacity management. Overall we achieved the goals given in Section 1.2. Our evaluation yields that with SLAStic.SIM we present a fast and valid simulator for complex models of hardware/software systems. With SLAStic.SIM it is possible to simulate static and dynamic (i.e., runtime reconfigurable) PCM models. For static PCM models, SLAStic.SIM performs well compared to SimuCom, which served as a reference implementation to our work, regarding user capacity and the computation time for a simulation run. Another advantage of SLAStic.SIM is that we can simulate a system using arbitrary workload input.
Reviewing the decision of implementing a new simulator from scratch in contrast to extending the implementation of SimuCom, we can say that we achieved some positive results regarding the flexibility and extendability of the simulator. Our evaluation showed that interpreting PCM models dynamically can be done in a performant way. In addition, for reconfiguration of a system, we do not have to regenerate and compile new simulation code, like in SimuCom.

Currently we support previously generated or recorded workload. This makes it possible to simulate systems using real-world workload. SLAStic.SIM is also embeddable into an online SLAStic framework, simulating a model of a deployed system to evaluate different reconfiguration plans in advance. In this case, previously recorded workload is very useful, as it can be seen as a realistic input to the system.

7.3 Future Work

In this section we propose possible extensions to SLAStic.SIM.

7.3.1 Load Balancing

In SLAStic.SIM we implemented a random load balancing strategy. This strategy is very simple and not used in real-world web servers like Apache [apa]. Instead of using our example balancer, which does not use any information about server load or traffic load, more high-level balancing algorithms could be used. As an example, we could implement the Apache load balancing algorithms in SLAStic.SIM by extending the `LoadBalancer` class and provide a simple configuration file for choosing among the available balancer implementations. Apache’s request counting algorithms can be easily implemented by modifying the random balancer strategy. For implementing the weighted traffic counting algorithm, we would need to provide traffic information to the load balancer, which is currently not implemented. The pending request counting algorithm keeps track of requests that have already been redirected to the servers.

7.3.2 Explicit Connector Model

In PCM, connectors between assembly contexts are modeled in a very simple way, i.e., it is only specified that two assembly contexts are connected, but not how. An approach has been described by Becker [Bec08]. As an extension for PCM in regard to more flexible simulation, we propose to implement explicit modeling of typed connectors. Especially when it comes to load balancing, we are currently missing the possibility to specify load balancing strategies. The future extension could be to annotate the connectors with load balancing strategies.
7.3.3 Kieker Probes

Currently we instrumented the simulated system with only one monitoring probe type, which generates KiekerExecutionRecords. Possible future extensions might include but are not limited to server load probes or reconfiguration event probes. Implementing these probes hopefully yields more detailed information about the simulated system and especially better evaluation methods against possible cost models (see Section 7.3.4).

7.3.4 Cost Model for Reconfiguration Operations

At the moment, SLAstatic.SIM’s implementations of the reconfiguration operations are not consuming any simulation time. Of course, this is an idealized view and is not a credible model. Especially the allocation and deallocation of computing nodes are time consuming operations. Hence future extensions to the simulation of reconfiguration operations might be time consuming reconfiguration operations. This could also lead to tighter preconditions and extensions in to PCM. Resource container’s have to be annotated with their allocation or deallocation times and components by their code size, which makes it possible to estimate the time needed to instantiate a component on a given server. The latter case is also resource demanding, i.e., it might be possible to implement a general parametric resource demanding behavior, describing the instantiation of a given component.

7.3.5 Component Replacement Operation

Currently SLAstatic.SIM can only handle reconfiguration operations that focus on performance improvement. Bunge and Matevska proposed another possible reconfiguration operation [Bun08, Mat09]. This operation replaces a components implementation during runtime, thus making online updates possible. Another similar reconfiguration operation was proposed by Diaconescu et al. [DM05]. Implementing these operation consistently in SLAstatic.SIM would yield information about their usability and performance in static or even dynamic allocation environments.

7.3.6 Other Workload Models

At the moment we only allow for previously recorded Kieker workload. This is a limitation, as it is not possible to use real open or closed workload models. Thus we cannot use existing PCM workload scenarios to drive SLAstatic.SIM. SLAstatic.SIM hence might be extended by abstracting the workload input and use a plugin based system, similar to the SimuCom WorkloadDrivers and actually
7.3.7 Simulation Time Evaluation of the Control Flow

As another improvement regarding memory usage, evaluating the control flow traces during simulation time could be implemented. Currently, when allocating a heap size of 2048 Mega Bytes we are able to hold approximately 40,000 control flow traces of the bookstore example presented in Section 5.4. As these traces are rather small, reduction of memory usage is desirable. Evaluating the control flow during simulation time gives another advantage. Currently SLAStic.SIM does not support parametric resource demands, as they need a variable evaluation framework that evaluates parameters during simulation time. So evaluating the variables should be done in the same step as evaluating the control flow.

7.3.8 Evaluation of Stochastic Expressions

In SLAStic.SIM we do not evaluate stochastic expressions used in the PCM input. We omitted the possibility of using SimuCom’s evaluator as this would have yielded dependencies to the whole SimuCom framework and thus also the Eclipse framework. We plan to implement a simple evaluation framework using the Desmo-J random number generators. In contrast to SimuCom’s evaluator, we would then be able to use different distributions. To specify which distribution is to be used for evaluating a stochastic expression, the PCM Stoex meta model for stochastic expressions has to be extended.

7.3.9 Formal Specification and Verification of Reconfiguration Operations

Currently the reconfiguration operations are not formally specified. In order to verify that reconfiguration of a system will never lead to inconsistency, we need a mathematical model for pre- and post conditions and the operation itself. Then we can apply formal methods to prove that the system will always stay consistent. We propose to use a temporal logic, like for instance, the Temporal Logic of Actions (TLA+) [Lam02].
Appendix A

Workload Generation Script

```perl
#!/usr/bin/perl

# read first command line argument, assumed to be an integer
$i = int @ARGV[0] if scalar(@ARGV) or die "usage: perl −e gen.pl <number of requests>";

for(0 .. $i){
    # scale to nano seconds
    $_ *= 1000000000;
    # print record to standard out
    print "\$2;$_;0;org.trustsoft.slastic.tests.bookstoreDifferentRecordTypes.Bookstore.searchBook();NULL;$_;$_;pc−vanhoorn;0;0\n"
}
```

Listing A.1: A simple script for workload generation
#!/usr/bin/perl

# read first command line argument, assumed to be the minutes of samples
$i = int @ARGV[0] if scalar(@ARGV) or die "usage: perl gencurve.pl < number_of
minutes > ";

for ($_ = 0; $_ < $i; $_ += 1) {
  # determine number of users $n arriving in minute 
  $n = numUsers($_);
  # let one user arrive each 1/$n minutes
  for ($b (1 .. $n) {
    $a = int (((($b/$n) + _) * 100000000) * 60;
    print "$2;$a;0;org.trustsoft.slastic.tests.
bookstoreDifferentRecordTypes.Bookstore.searchBook();
NULL;$a$b;$a;pc-vanhoorn;0;0
"
  }
}

# workload function
sub f {
  my $x = shift @_; 
  return 25 * sin((($x - 13)/3.2)+10;
}
sub g {
  my $x = shift @_; 
  return sqrt ($x^2)+23;
}
sub h {
  my $x = shift @_; 
  return 10 * sin((($x - 7)/6)+30;
}
sub numUsers {
  my $x = shift @_; 
  return (1.8*f($x) + 3.5 *h($x) + .8*g($x) -110);
}

Listing A.2: A simple script for workload generating a workload curve
Bibliography


[EFH+08] Sven Efftinge, Peter Friese, Arno Haase, Dennis Hübner, Clemens Kadura, Bernd Kolb, Jan Köhnlein, Dieter Moroff, Karsten Thoms, Markus Völter, Patrick Schönbach, Moritz Eysholdt, Dennis Hübner, and Steven Reinisch. openArchitectureWare User Guide, 12 2008.


[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1995.


Bibliography


Declaration

This thesis is my own work and contains no material that has been accepted for the award of any other degree or diploma in any university.

To the best of my knowledge and belief, this thesis contains no material previously published by any other person except where due to acknowledgments has been made.

Oldenburg, 1.4.2010

______________________________________________________
Robert von Massow