

OpenXL: An Adaptable Web-Service Middleware

Ulrich Dinger
TU Dresden & Siemens AG
Hans-Grundig-Strasse 25
01062 Dresden, Germany
dinger@rn.inf.tu-dresden.de

Donald Kossmann
ETH Zurich
Universitätsstrasse 6
8092 Zürich, Switzerland
kossmann@inf.ethz.ch

Christian Reichel
ETH Zurich & Siemens AG
Universitätsstrasse 6
8092 Zürich, Switzerland
christian.reichel@inf.ethz.ch

ABSTRACT

The inherent complexity of current software systems has reached a critical level. This affects the general deployment complexity in the form of a huge amount of varying non-functional requirements (NFRs). With regard to flexibility and maintenance, changes to software systems need to be automated in order to deploy different configurations of a system based on a given set of NFRs. This paper proposes an Automated Software Engineering (ASE) approach which addresses the automated compile time/deployment time adaptation of NFRs in the context of Web Service middleware. It is shown why Web Service standards and a high-level programming language are advantageous for the NFR-optimization of a software system, and how flexible, adaptable Web Service middleware (OpenXL) can be implemented in the industry context using ASE techniques. The paper describes the architecture of OpenXL and depicts in detail how an automated middleware composition process is implemented. Additionally, it is illustrated how a single code source can be used for multiple applications which all address different NFRs. Moreover, a standardized benchmark (TPC-W [1]) is utilized in the measurements section in order to evaluate the OpenXL implementation and to draw meaningful conclusions of the accuracy of ASE. Basically, the proposed ASE approach is generally applicable to the automated NFR-adaptation of software systems.

Categories and Subject Descriptors

D.2 [SOFTWARE ENGINEERING]: Management – *Software configuration management*; D.2 [SOFTWARE ENGINEERING]: Software Architectures – *Domain-specific architectures*

General Terms

Management, Measurement, Design

Keywords

Non-Functional Requirements, NFR, Web Service, Platform, Middleware, OpenXL, Service Language Layer, SLL, Semantic

1. INTRODUCTION

Regarding complexity, the optimization of software systems for different non-functional requirements (NFRs) can be a critical task using state-of-the-art middleware and programming

languages. For example, application servers such as JBoss [2], WebSphere [3] and WebLogic [4] are responsible for most of the NFRs (size, memory footprint, throughput, response time, etc.) of the software. Basically, the non-functional properties of this middleware are rather static than adaptable which results from the fact that established middleware products are tuned for a specific scenario (often server applications) and have not been designed to be flexible in this matter. This is acceptable for many use cases, but if the middleware requires an optimization for different non-functional requirements existing implementations are not sufficient.

A representative NFR-adaptation use case is provided by Home Automation and Management Software (HAMS) which typically must run on different types of target devices (such as PDA, tablet PCs, home gateway, central server) inside a house and must additionally address the requirements of different users within heterogeneous application scenarios. The work at Siemens AG on projects related to HAMS has shown that the deployments of such systems have to take various NFRs (size, footprint, throughput, etc.) into account and that for reasons of maintainability, usability and flexibility these deployments have to be automated as much as possible. Basically, Java, XML and Web Services took important steps towards a platform-independent development and deployment of software. This paper makes use of these efforts and proposes an automated software engineering (ASE) [5] approach for the creation of a Web Service-based middleware implementation (OpenXL) in the context of HAMS which supports an automated adaptation and deployment based on different NFRs. The ASE approach which was used for the implementation of OpenXL defines the following three principles:

1. High-level programming language/model. A new abstraction layer on top of traditional programming languages (such as Java) raises the level of abstraction and avoids hard bindings on lower level frameworks and APIs via the provisioning of high-level statements. Thus, the programming logic can be separated from inherent NFRs.

2. Adaptable middleware architecture. A middleware layer is responsible for the interpretation of the new high-level programming language. In order to address a flexible adaptation, it is implemented using a pluggable skeleton which can host exchangeable implementations (modules). With regard to maintenance, the skeleton is automatically generated out of the high-level model.

3. Automated (re)configuration. Regarding an automated (re)configuration of the middleware, all middleware modules are annotated with meta-data such as dependencies, metrics and functionalities. A specific tool, the feature configurator, evaluates these data and realizes an automated deployment of modules (given set of NFRs) using semantic reasoning.

These principles are leveraged for the implementation of OpenXL. Subsequent sections of the paper are structured as follows: In the second section the general scope of the solution and related work are described. Section 3 depicts how OpenXL was implemented using ASE and what the corresponding parts of the OpenXL architecture are. Section 4 shows how functional and non-functional requirements of the OpenXL middleware can be stepwise and automatically tuned, e.g., from a Micro Edition which was optimized for handheld and embedded devices towards an Enterprise Edition platform. Moreover, the section evaluates the created Web Service middleware implementation and the accuracy of ASE using the TPC-W [1] benchmark and additionally gives a report of the use-case experiences. In the last section of this paper, conclusions and avenues for future work are provided.

2. SCOPE OF SOLUTION

The growing complexity of software systems requires optimized methodologies in order to handle the huge set of functional as well as non-functional requirements. In this context, larger software systems often have to run on a variety of different target hardware platforms (servers, gateways, PDAs, etc.). Adapting the software system to the different cost and maintenance tradeoffs and specific user requirements should not require any manual work; in particular, it should not require changing any code of the software system. This is the main objective addressed in this work.

2.1 Requirements

The introduction mentioned home automation (HAMS) as one motivating example. For HAMS, a flexible middleware is required which supports the networking and interaction of heterogeneous devices in a home environment based on Web Service standards. The devices in this scenario are provided by multiple vendors and all have different hardware properties, for instance, regarding available memory, physical size and performance. The only universally valid assumption for the hardware in the context of HAMS is that a Java Virtual Machine (JVM) is supported on all devices. Basically, the applications in the HAMS scenario (blinds controller, telecommunication system, video management, etc.) must be controllable via all devices which have a user front end.

This means that the middleware must be deployable on each device and, thus, must be able to address the varying NFRs with regard to heterogeneous hardware-, user- and application scenarios. Basically, the HAMS implementation requires flexible support with regard to the automated adaptation of the following middleware NFRs:

1. *Overall Physical Size,*
2. *Memory Footprint,*
3. *Startup Time,*
4. *Average Response Time (ART) and*
5. *Throughput measured as Web Interactions per Second (WIPS)¹.*

In general, all kinds of NFRs should be addressable. For all of the mentioned NFRs, a MINIMIZE, MAXIMIZE and concrete value (e.g., overall software size < 10 MB) optimization needs to be supported which also includes a mixture of several NFRs. Apart from that, it is specified that the middleware system must provide

mechanisms for the flexible exchange of modules (such as communication, persistence and data manipulation) which are typically implemented by different vendors. It is also a general requirement that standards are used where applicable, e.g., for the specification of module dependencies or module metrics. Moreover, the HAMS scenario requires a static (compile time) as well as dynamic (deployment time/runtime) optimization process, including a remote deployment of the middleware modules. Summarizing these requirements it can be said that HAMS needs automated support for multiple application deployments with different non-functional requirements using a single code source.

2.2 Related Work

Currently, several approaches address the management of NFRs or a platform-independent programming/deployment. Although, most of these approaches provide a solution for important sub-aspects, a more flexible and integrative solution is missing which addresses a platform- and NFR-independent programming in order to allow a flexible deployment of applications (e.g., at compile time or deployment time) given a target platform and a set of NFRs. The following state-of-the-art approaches basically address the aforementioned requirements and cover the management of NFRs:

a) Container. A container-based infrastructure already allows the deployment of standardized components/services and manages a well defined set of NFRs such as security, persistence or communication. Examples for a container-based management of NFRs are provided by existing J2EE [6] containers (application servers) or Microsoft's .Net infrastructure [7] respectively. Nevertheless, solely using a container approach, basically more advanced optimizations of NFRs (such as memory footprint, size) are often not possible or forbidden, either because the deployed component/service has a specific API dependency or because the underlying container is not designed for a fine-grained exchange of internal parts. Thus, existing container approaches do not support an automated adaptation based on the requirements of section 2.1 which limits its usage in the scenarios like HAMS.

b) AOP. Aspect Oriented Programming (AOP) handles cross cutting concerns via aspects. The underlying weaving process of state-of-the-art AOP implementations already allows to weave in specific aspects (such as security, caching, etc.) at compile time (e.g., AspectJ [8]), load time or runtime (PROSE [9]). AOP can be combined with the container approach and therefore realize more flexible adaptations with regard to NFRs. Nevertheless, specific properties (such as the size of a container) are basically not adaptable via AOP and crosscutting concerns between aspects (e.g., dependencies) are currently not sufficiently describable and manageable. As a result, an automated weaving/adaptation process of a system based on the NFRs of section 2.1 is currently not affected via AOP which, again, restricts its application in the HAMS scenario.

c) MDA/Feature Modeling. OMG's [10] MDA and other MDSD approaches support the modeling of platform independent domain knowledge and the generation of platform specific code out of these models. In this context, techniques such as model transformations and code generation, e.g. via Query View Transformations (QVT) [11], are currently under development/standardization. Languages like the Unified Modeling Language (UML) [12] and EMF Ecore [13] are utilized to describe software in a platform independent way. In general, this approach requires a manual (re)configuration of the

¹ This term is specified inside the TPC-W benchmark [1].

underlying code or generators each time the NFRs are changing. For a more flexible and automated handling of software features, generation techniques can be combined with Feature Modeling techniques which support “the conceptual abstraction and description of relationships between distinguishing characteristics of SE artifacts or assets” [5]. In this context, existing tools such as pure:variants [14] and XFeature [15] address the management of features, but for reasons of missing tooling and standards for the handling of NFRs and missing runtime adaptation support, a more automated and flexible handling of NFRs based on the requirements of section 2.1 is still limited.

d) Manual. It is also an option that the programmer manages all NFRs manually or that existing systems simply remain static (which eliminates certain use cases). This approach is widely-used, but has several limitations with regard to (re)usability, flexibility or maintenance in the HAMS scenario. Basically, this approach can not address the requirements of section 2.1 at all and therefore is not an option for the automated management of NFRs.

The OpenXL implementation provides “best of breed“ of the aforementioned MDA, AOP and Container approaches. It combines the abstraction of MDA with the practical usability of containers and if needed includes the dynamics of AOP.

3. OPENXL IMPLEMENTATION

Regarding a platform- and NFR-independent programming, the subsequent sections report the results made during an industrial strength product development carried out in a cooperation between ETH Zürich and Siemens AG. In this context, an adaptable, open-source Web Service middleware (OpenXL) using an automated software engineering (ASE) approach was created which provides several extensions with regard to traditional middleware implementations (e.g., the usage of a high-level Web Service programming language) and thus can be flexibly optimized (e.g., statically or dynamically) for varying NFRs. Within this section, the ASE process and the main components of the OpenXL middleware architecture are introduced and described.

3.1 Usage of a High-Level Programming Language and Internal Model

The usage of a high-level programming language (cp. traditional languages such as Java or C#) in combination with an XML-based data and type system on top of existing programming languages is a basic cornerstone of the OpenXL middleware implementation. Such a language is able to provide a high-level, API-independent description of source code that can be used to implement applications in a platform-independent way. For example, a high-level statement like

```
encrypted send_sync MyService::op1 ($MyXMLMessage)
```

concisely expresses that the operation `op1` of an external Web Services `MyService` is invoked using message encryption and the `MyXMLMessage` as input. Thus hard-dependencies on a specific framework/API are explicitly avoided inside the code (cp. an implementation that explicitly requires the framework Apache Axis [16] or WSIF). This approach supports a high flexibility with regard to the adaptation of NFRs, for instance, via compile-, deployment- or run-time bindings of `send_sync` statement implementations inside the middleware.

With regard to the OpenXL middleware implementation, the conceptual ideas of the *Service Language Layer (SLL)*² are utilized which were introduced in a separate paper [18] and are the basis for the flexible execution/interpretation of an arbitrary high-level Web Service program. SLL comprises an XML-based model (xSLL) for the internal representation of a Web Service program as well as a plain-text programming language (SLL_p) for the flexible development of Web Services. The conceptual idea of SLL is illustrated in Figure 1.

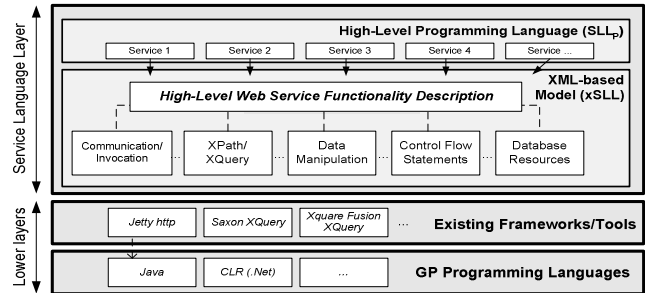


Figure 1: Service Language Layer (SLL)

As shown, SLL is a separate layer on top of general purpose (GP) programming languages and existing frameworks/tools. xSLL is a part of SLL and comprises a high-level XML-representation of functionalities such as Web Service communication, data manipulation, database resources, etc. All Web Services written in the high-level programming language (SLL_p) are convertible into the internal xSLL model and hence interpretable via any implementation of SLL. Instances of SLL_p and the internal xSLL model basically contain meta-data of a Web Service (provider, category, version, etc.) as well as concrete application logic which involves error handling code, invariants, compensation actions and operations which in turn define variables, expressions, statements (e.g., data manipulation, sync invocations), combinators (sequence, parallel, data flow), etc. In contrast to traditional programming languages, most of the xSLL model parts are on a higher level of abstraction. For example, the parallel combinator

```
{ //Block A } || { //Block B }
```

expresses that two blocks can be executed in parallel without the need to program threads. Another example was the aforementioned `send_sync` statement. Using such a high-level description, explicit dependencies on specific APIs (such as Axis or WSIF) are avoided which provides a basic ingredient for the needed separation of programming logic of source code from inherent NFRs.

Because the SLL_p programming language is a part of SLL and therewith on the same abstraction level as xSLL, a transformation between the SLL_p programming language and the xSLL model describes a one-to-one mapping between language and model constructs. Figure 2 illustrates an additional aspect of SLL which was described in [18]. As shown, programs written in traditional programming languages can be transformed into xSLL programs and vice versa. In the context of this paper and the implementation of OpenXL, this specific feature of SLL is not needed. It is expected that all programs are written in the high-level programming language SLL_p and afterwards converted to xSLL.

² A complete specification of SLL, including a XML Schema file of xSLL, is available under [18].

Nevertheless, this feature can be supported if the requirements of the HAMS scenario are changing.

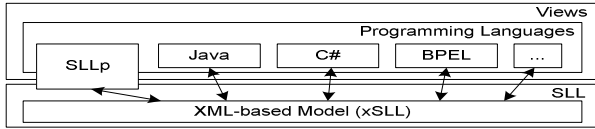


Figure 2: Service Language Layer (SLL) Cross-Compilation

SLL is naturally extensible (without invalidating existing code) by extending the XML Schema file of the xSLL model and by plugging-in new functionality modules into the OpenXL middleware (see the next section). This allows integrating new concepts and structures as the field of high-level Web Service languages matures. Concrete examples for SLL_p programs are shown in section 5, where a high-level Web Service programming language is needed in order to implement the TPC-W benchmark [1]. Therefore, SLL_p code is not part of this section.

3.2 Generation of a Modular Middleware Skeleton

The *Middleware Skeleton* is the implementation of the service language layer (SLL). The skeleton supports the deployment and execution of services written in SLL_p and provides a modular architecture to address the exchange (hot swap) of modules that implement specific SLL functionality based on different NFRs (e.g., a module which sends asynchronous messages via SOAP needs to be replaced with a JMS based implementation). With regard to interoperability, the platform skeleton is implemented in Java and is, thus, executable on all hardware platforms which provide a Java VM. This was a basic requirement of the HAMS scenario where all devices provide a Java VM (see section 2.1). In order to address maintenance aspects and general adaptation flexibility, ASE generates most of the OpenXL platform skeleton code using a generator framework called FXL [19] and the XML Schema file of the xSLL model. Basically, the FXL framework provides a way to convert an XML schema to an (E)MOF [12] representation and afterwards to generate Java classes via generator pipelines. The corresponding FXL pipeline which describes a workflow of transformations and was used for the generation of the middleware skeleton is shown in Figure 3.

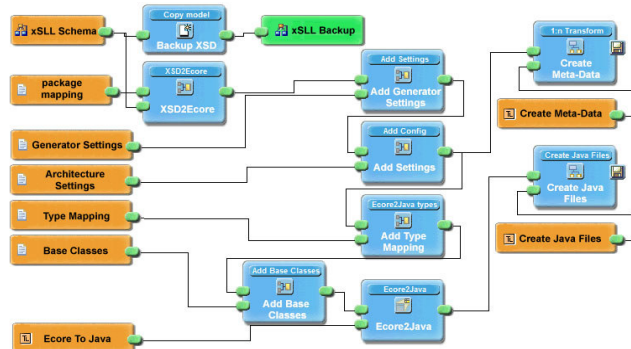


Figure 3: Middleware Skeleton Generator Pipeline

As illustrated, the pipeline accepts the XML Schema file of the xSLL model as input, converts it to Ecore, adds the package-, generator- and type-settings, converts Ecore to Java and at the end writes the OpenXL middleware skeleton to the file system. The underlying generator files (transformation) are based on existing Ecore2Java XSLT/TL transformations of the FXL project and were specifically adapted for the creation of a flexible OpenXL

middleware skeleton architecture. In this context, the main adaptation work of the transformations was related to the integration of architecture patterns as well as parametrizable transformation structures. The generated OpenXL skeleton comprises the default implementations (abstract classes) for a specific instruction of the xSLL model (including load and save functionality) as well as the infrastructure which is used to add a specific SLL functionality implementation (module). With regard to modularity, the platform skeleton can be configured as a static infrastructure (with factories for implementations of certain xSLL features) as well as a dynamic infrastructure (e.g., an OSGI-container based on the implementation). In each version (static or dynamic), the OpenXL skeleton is an adaptable system which is able to exchange and host provider-specific modules.

Figure 4 summarizes the aforementioned steps which were involved in the creation of the platform skeleton. As shown in the center of this figure, a platform skeleton provides the main parts of an xSLL interpreter. It is additionally illustrated that the provisioning and deployment of OpenXL modules has not been addressed yet.

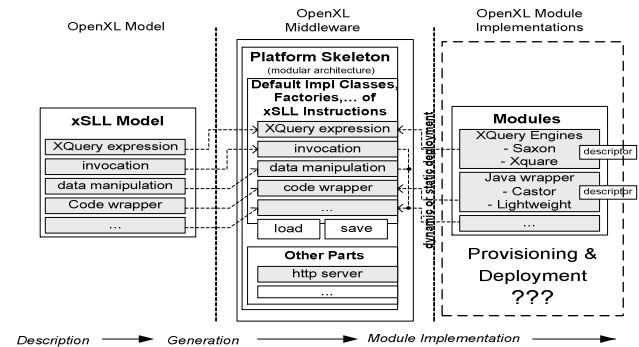


Figure 4: OpenXL Platform Creation Process

3.3 Semantically Annotated Modules as a Basis for Automated Middleware Adaptation

As mentioned before, an OpenXL module provides an implementation for a specific piece of SLL functionality (e.g., invocations via the `send_sync` statement) by superimposing the (generated) default implementation of its abstract parent instruction. In this context, the following interface method of the Instruction base class inside the middleware skeleton is the most relevant one:

```
public void execute(Context context)
throws RuntimeException, NotImplementedException;
```

Each module provides an implementation of this method for one or more xSLL instructions. Inside the method, active resources, variables, settings, etc. can be accessed and modified via the runtime context parameter. If a specific module is deployed inside a platform, its execute method is called each time the control flow of the service passes the instruction. Inside the method, each instruction analyses the active context, calls other sub-instructions if needed and implements the specific functionality. Basically, OpenXL modules for a certain piece of xSLL functionality can be provided by different vendors, e.g., a Saxon [20]-based or an XQuare Fusion [21]-based implementation for the execution of an XQuery [22] expression.

All modules are basically encapsulated in bundles and annotated with semantic information. In general, various kinds of meta-information are relevant (dependencies between modules, feature

descriptions, NFR relations, etc.) which must be described and documented for a seamless integration in an automated OpenXL middleware adaptation process based on a given set of NFRs. The annotated information goes beyond existing approaches, e.g., simple OSGI manifest files which cannot describe complex relationships of modules or the features and semantics of modules. In order to express the complex dependencies, a feature-based model (ontology³) was developed in the context of ASE which comprises pre-defined entities (e.g., XQueryEngine, Variable, Wrapper, etc.) as well as expressions for relations between entities (hasFeature, requiresFeature, prohibitFeature, etc.). Via a descriptor file written in the SLL_p language, the semantic information is made available for each module. SLL_p is also used for the semantic description of modules because it provides a unified approach for the high-level description of application logic as well as of meta-information/semantic (cp. WSDL-S [24], OWL-S [25], WSMO [26], etc.).

Listing 1 shows the SLL_p-based descriptor file of the concrete Saxon XQuery engine implementation. As illustrated, general information of the OpenXL module (version, category, etc.) as well as new features (ontology concepts) are specified in the global description part of the file. Basically, locally declared or globally available ontology concepts are needed for the modeling of relations and dependencies between modules.

```

namespace depl = "http://www.open-xl.org/plat/deploy";
namespace ont = "http://www.open-xl.org/rts-ontology";
...
service SaxonDescriptor implements depl:DeploymentPT {
  description {
    version 8.7;
    text "SAXON B XQuery Engine";
    category "ch.ethz...sll.model.expressions.xquery";
    semantic {
      <ont:feature xsi:type="ont:Functionality"
        name="SaxonXQueryEngine"
        ontologyNamespace="http://saxon.sourceforge.net">
        <isSubFunctionalityOf>
          ont#XQueryEngine</isSubFunctionalityOf>
        <hasFeatures>
          <feature name="nfr#SupportsLazyLoading"/>
          <feature name="nfr#StartupTime"
            value="619" unit="nfr#ms"/>
          <feature name="nfr#PhysicalSize"
            value="11141" unit="nfr#Byte"/>
          <feature name="nfr#ART" context="nfr#TPCW">
            <link ref="http://.../TPCW/results.xml"/>
          </feature>
          ...
        </hasFeatures>
      </ont:feature>
    }
  }
  public operation deploy($platform as ont:XLPlatform) {
    ... //see the content in listing 2
  }
}

```

Listing 1: Saxon XQueryEngine Descriptor File (1 of 2)

In the case of the SaxonDescriptor, a new SaxonXQueryEngine feature concept was specified inside the module descriptor which is of the type ont:Functionality and a sub-functionality of ont#XQueryEngine. The SaxonXQueryEngine feature itself has several features (non-functional properties) such as nfr#SupportsLazyLoading, nfr#StartupTime, nfr#PhysicalSize, etc.

As shown in Listing 2, the aforementioned SaxonXQueryEngine feature is referenced in the deploy operation of the descriptor

³ The OpenXL ontology [5] was developed using Semantic Web principles [23].

file which is a part of the OpenXL depl:DeploymentPT interface and accepts a \$openxl parameter as input. The deploy operation of the SaxonDescriptor specifies the pre-conditions, effects and post-conditions which are related to the deployment of the SaxonXQueryEngine module inside the OpenXL middleware. In this specific case, a deployment would only be possible if the target middleware is executed on a Java 1.4.x compliant VM and another XQueryEngine has not been deployed inside the middleware yet. As an effect of the deployment, the middleware afterwards comprises the #SaxonXQueryEngine feature which is specified in the semantic part of the description section⁴.

```

public operation deploy($openxl as ont:XLMiddleware) {
  semantic {
    ...
    pre_conditions {
      <ont:semantic>
        <variable qname="openxl">
          <hasFeatures>
            <feature>nfp#Java 1.4.x</feature>
          </hasFeatures>
          <prohibitsFeatures>
            <feature>ont#XQueryEngine</feature>
          </prohibitsFeatures>
        </variable>
      </ont:semantic>
    }
    post_conditions {...}
    effects {
      <ont:semantic>
        <variable qname="platform">
          <hasFeatures>
            <feature>#SaxonXQueryEngine</feature>
          </hasFeatures>
        </variable>
      </ont:semantic>
    }
  }
}

```

Listing 2: Saxon XQueryEngine Descriptor File (2 of 2)

Up to now a middleware skeleton and semantically annotated modules are available which alone cannot address an automated deployment of the OpenXL middleware based on NFRs. Therefore a new middleware part is needed (see Figure 5) which is described in the subsequent section.

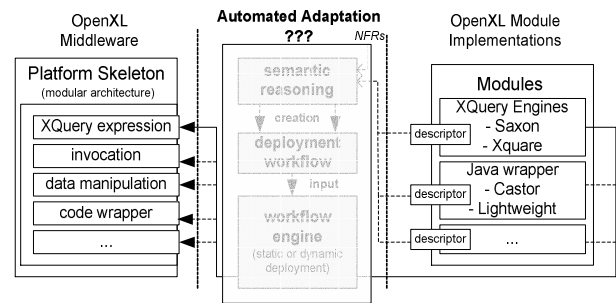


Figure 5: Automated Adaptation Still Remains an Open Issue

3.4 Usage of a Feature Configurator Wizard (FCW) for Automated Adaptations

The *Feature Configurator Wizard (FCW)* was developed to manage the automated adaptation of the middleware features based on NFRs in the context of ASE. Therefore, the FCW has

⁴ If the #SaxonXQueryEngine feature must be referenced by other module descriptors, this part must be made available via a globally available ontology file.

direct access on the deployed modules of existing middleware configurations as well as generally available modules (which are published in a global registry or stored in a module directory). As mentioned before, OpenXL modules are described using semantic descriptor files written in SLL_p. Within these files, functional as well as non-functional properties of a specific OpenXL module are modeled. Basically, the embedded reasoner of the FCW evaluates this information and uses it for the creation of the feature tree. Implementation-, complexity- and performance-aspects of the underlying reasoning process are described in a separate paper [5] and are therefore not in the scope of this paper. Basically, the paper illustrates that workflow generation is an NP-hard problem but also states that it can be sufficiently addressed via heuristics and optimized algorithms in the ASE context.

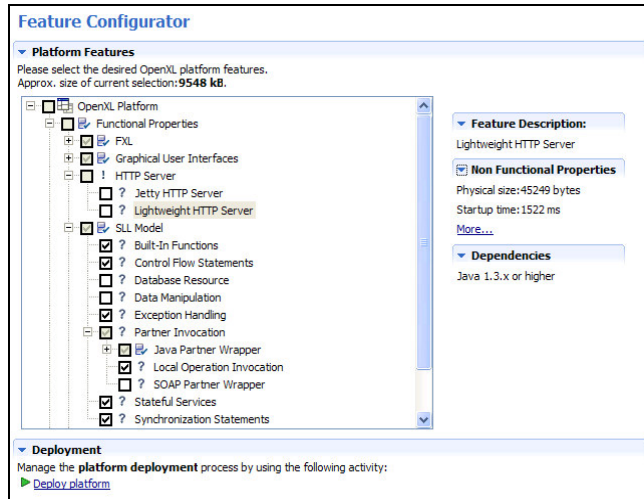


Figure 6: Feature Configurator Wizard

As illustrated on the left side of Figure 6, the features of an existing or new platform are adaptable via the FCW which provides a specific view on the functional as well as non-functional properties of all registered modules. A single feature can be activated or deactivated via its checkbox for a specific middleware configuration which means in the case of a functional feature selection that the corresponding module (which provides this feature) will be a part of the deployed middleware. If dependencies to other features or feature exclusions exist (e.g., caused by NFR decisions) a specific activation of a feature can be explicitly forbidden by the FCW. The exclamation mark on the left of a feature states that this feature or a child of this feature is required for a successful deployment whereas the question mark indicates an option. Moreover, gray checkboxes indicate that all mandatory features of this sub-tree are fulfilled. On the right of Figure 6, specific information of the selected feature, for instance, a description of the feature, non-functional properties as well as dependencies to other features are shown.

The “deploy platform” link (bottom of Figure 6) is used after the completion of the feature selection process in order to initiate the middleware deployment process. This process involves several steps, for example, the modification of the platform name, the selection of a static (Factory-based) or dynamic (OSGI-based) architecture, etc. and ends with the creation of a deployment document (SLL_p based workflow description) which is responsible for the automatic deployment of a new or an adaptation of an existing middleware. For the execution of the deployment document an existing OpenXL middleware

implementation can be bootstrapped. In the case of a dynamic platform skeleton, the deployment process simply involves the (un)deployment of OpenXL OSGI bundles (modules). In contrast to that, the static platform skeleton initiates an adaptation of the middleware skeleton (module factories, configuration files, etc.) via FXL [19] pipelines which also involves an inclusion of the corresponding modules code in the classpath of the middleware. Both architecture versions, static or dynamic, address different requirements (flexibility vs. size) and are needed in the HAMS scenario.

The main parts and steps which are involved in an automated middleware creation and adaptation process are summarized in Figure 7. What is still missing is a “real life” use case which applies the aforementioned adaptation process in order to create NFR-optimized middleware configurations. This is described and depicted in the subsequent sections.

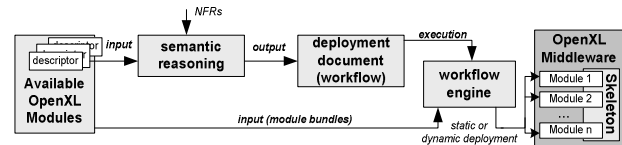


Figure 7: Automated Middleware Adaptation Process

4. MIDDLEWARE ADAPTATION

It is a basic objective of this paper to evaluate the principles of automated software engineering (ASE) [5] and the created OpenXL middleware implementation in “real-life” use cases with regard to a flexible adaptation of NFRs. In order to provide comparable results, the paper uses a representative benchmark: the TPC-W benchmark which tests ShopService implementations. The following sub-sections depict the creation of three different OpenXL middleware configurations based on a given set of NFRs which are relevant in the HAMS use case and are used as representative basis in TPC-W measurements of section 5.

4.1 The Basic NFRs of HAMS Middleware Configurations

The main requirements of the tested HAMS middleware configurations are (cp. requirements of section 2.1):

Micro Edition (ME): This configuration must run on a PDA using a Java 1.3 VM and must be automatically optimized for size and memory footprint (NFRs). Only a restricted amount of concurrent activities needs to be supported. In the context of TPC-W, not all functionalities/methods of the deployed ShopService are required, for example, with regard to the integration of existing Java legacy code.

Standard Edition (SE): This version must run on a home gateway using a Java 1.4 VM or higher. In this scenario, the memory footprint and size of the platform are not important. The platform must be automatically optimized for Average Response Time (ART) and a high amount of concurrent activities. It requires full support for the integration of Java legacy code and only support for a file or memory-based storage system.

Enterprise Edition (EE): This platform must run on a server machine and must be optimized for Web Interactions per Second (WIPS) in the higher ranges of concurrent activities (> 20 concurrent clients). It also requires a high-performance storage system for the main parts (e.g. shop items, user lists) of the TPC-W ShopService. The Java VM version can be 1.4 or higher.

4.2 Automated Configuration Process Based on the Given NFRs

Each of the aforementioned OpenXL platform configurations (ME, SE, EE) need a separate optimization process (selection of different OpenXL modules) to satisfy the given NFRs. In order to automatically determine the required OpenXL modules of each platform configuration, the FCW (which was described in section 2.4.3) is utilized. Figure 8 shows how, for instance, the maximum size of the ME platform can be restricted during the initial platform deployment process. Via a global platform size restriction (on the right side of Figure 8) several combinations of modules are automatically excluded/invalidated, for example, that the Saxon XQuery Engine and the Jetty http server [27] are used together for the ME which is forbidden because of the fact the usage of both modules can not keep the max 10MB platform size condition. In any case Saxon would not be an option for the ME platform because it requires Java 1.4.x or higher which collides with the Java 1.3.x or higher requirement. In order to effect this kind of middleware deployment support, the FCW leverages the semantic descriptor files of all registered modules. Basically, a reasoner (see section 3.4) checks the semantic information of the underlying modules and selection dependencies and forwards the results to the GUI which in turn propagates them to the user.

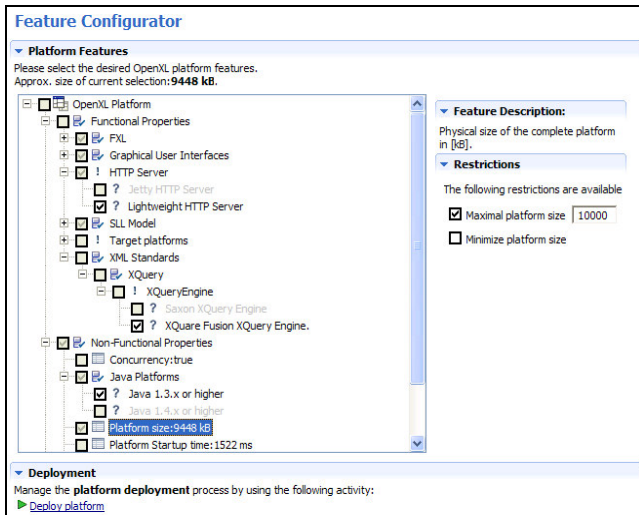


Figure 8: Maximum Platform Size NFR

The first part of Table 1 shows a representative snippet of the FCW middleware configuration results which were determined using the bold-printed NFR input of the second part of Table 1.

Table 1: OpenXL Middleware Configurations

Open Source Modules	Versio	OpenXL Middleware Configuration		
		Enterprise	Standard	Micro
HTTP Server				
- Jetty	5.1	x	x	-
- OpenXL Lightweight	1.1	-	-	x
XQueryEngine				
- Saxon	8.7.1	x	x	-
- XQuare Fusion	1.1.1	-	-	x
Java Partner Wrapper				
- OpenXL Lightweight	1.1	-	-	x
- Castor	0.9.5.4	x	x	-
Storage				
- XML-DBMS	2.0 A3	x	-	-
- MySQL/Connector/J	3.1.12	x	-	-
- Memory/File System	-	x	x	x
Other (not listed here)				

Used NFR Input				
- Overall Platform Size [MB]	-	DON'T CARE [12,22]	DON'T CARE [10,95]	<10MB [6,47]
- Platform Memory Footprint [MB] after Startup	-	DON'T CARE [0,99]	DON'T CARE [0,97]	MIN [1,16]
- Platform Startup Time [ms]	-	DON'T CARE [1578 / 2197]	DON'T CARE [1546 / 2180]	MIN [1453 / 2737]
- ART	-	DON'T CARE	MIN	DON'T CARE
- WIPS	-	MAX	DON'T CARE	DON'T CARE

For example, in the case of the Micro Edition (ME) middleware the specified NFR input was [Platform Size]=[<10MB], [Startup Time, Memory Footprint]=[MINIMIZE], [WIPS, ART]=[DON'T CARE], [Java Platforms]=[1.3.x or higher]. This correlates with the requirements of section 2.1. The results of the FCW in the specific scenario of the ME are: usage of a lightweight http server in combination with the XQuare Fusion XQuery Engine, a lightweight Java partner wrapper (which is able to integrate legacy code) as well as the integration of a memory/file based storage. This combination provides the best configuration for the aforementioned ME requirements which basically address a lightweight OpenXL platform edition. Several final metrics of the platform configurations (ME, SE, EE) are shown in square brackets in the second part of Table 1. These metrics are calculated during the platform configuration process using the measurement meta-information of a reference system (see section 3) in the underlying module descriptors. The overall platform size is basically a metric which remains static for different reference systems and is therefore only affected by the module configuration of a middleware itself. Regarding the startup time metric, the first value in the brackets states the time which is needed until the middleware has completed the startup phase on the specific reference system. This metric does not include lazy loading times of modules, e.g., the XQuery engine startup time. In contrast to that the second value comprises all lazy loading times, e.g., the startup time of the XQuery engine. The memory footprint metric (also in square brackets) specifies the memory footprint of a middleware configuration after all module initializations.

In addition to the three configurations, XL [30] which first of all provided a Web Service programming language in combination with a runtime system is included in the TPC-W benchmark evaluation process. It provides an excellent reference to compare the changing configurations (ME, SE, EE) of the OpenXL Web Service middleware against a static system which has the same focus. The first part of Table 2 shows the main components of the XL version whereupon with regard to the best comparison possibilities the database (DB) functionality of XL was deactivated and the corresponding DB libraries were not included in the XL metrics (although this would be possible).

Table 2: XL Components and Main Metrics

Libraries	XL Standard
Http Server	
- XL Lightweight	x
Java Partner Wrapper	
- Castor	x
XQuery Engine	
- BEA XQRL	x
Storage	
- Memory/File System	x
Other (not listed here)	
Static Platform NFRs	
- Overall Platform Size [MB]	10,95
- Memory Footprint [MB] after Startup	2,79
- Startup Time [ms]	3610 / 4117

In general, XL uses a lightweight http-server implementation, the BEA XQRL XQuery engine and a memory/XML-file based

storage implementation. All metrics such as size, memory footprint or the startup time are illustrated in the second part of Table 2 and were measured using the reference server system specified in section 3.

5. MIDDLEWARE MEASUREMENTS

Within this chapter a simple HAMS service scenario (BinaryLightService) and standardized benchmark (TPC-W) is applied to the static XL system as well as the three OpenXL middleware configurations. For each measurement sub-scenario, a pre-build ME configuration was dynamically adapted (at runtime) towards the SE and EE middleware configurations. For this purpose, the ME middleware skeleton is based on the Knopflerfish OSGI container [28] (see section 3.4).

5.1 General Measurement Assumptions

It is a basic objective of the underlying measurements to acquire the middleware metrics which were defined in section 2.2, to evaluate the implementations and to draw conclusions from the accuracy and exactness of the middleware deployment process. Basically, all tested configurations of this paper are relevant for ETH-Siemens AG cooperation use-cases (e.g., for HAMS). In general, all measurements were recorded using the following systems:

- **OpenXL:** P4 3GHz, 1 GB RAM, WinXP SP 2, JDK1.5.0_07
- **Remote Client Emulator (RCE):** P4 3GHz, 1 GB RAM, WinXP SP 2, JDK1.5.0_03B7

Both systems are connected via a crossed network cable (1Gbit Ethernet) in order to minimize potential side effects caused by network latency, throughput, etc. The universally valid requirements for all measurement scenarios are:

- Overall 4000 request messages are sent.
- Each Remote Client Emulator (RCE) thread sends a message, waits for the server response and starts to send a new message until the message limit will be reached.
- The ‘response time’ metric measures the time elapsed between the leaving of a message and the arrival of the response message on the client. The Average Response Time (ART) specifies the arithmetic average, where the sum of all response time values is divided by the overall message count (4000).
- The WIPS metric is a TPC-W benchmark term. It is used to refer to the average number of Web Interactions Per Second completed during the Shopping Interval.
- The term Platform Worker Threads (PWT) specifies the number of parallel threads which can process an accepted http request on the server.
- The term Client Worker Threads (CWT) states the number of parallel clients which are emulated by the RCE.

5.2 Simple Scenarios Cannot Show the Full Potential of NFR-based Adaptations

The first measurement scenario starts with a very simple service (BinaryLightService) in order to compare the measured metrics (WIPS and ART) of the different middleware configurations with the NFR input of section 4.2. This is a typical scenario where NFR-based adaptations do not much carry weight. It is expected that a more lightweight middleware configuration such as the Micro Edition (ME) can relatively keep up with the other configurations in this scenario, because performance optimized versions (such as SE, EE) lose their advantages in a scenario of

simple queries and statements as it is the case in the BinaryLight scenario. In section 5.3.1 and 5.3.2 the queries become more and more complex which gives an example for the necessity of NFR-driven adaptations and illustrates the importance of automated middleware configurations for a specific use case.

5.2.1 The BinaryLightService Measurements

As shown in Listing 3 the tested BinaryLight service is written in the SLL_p language [18] (see section 3.1) and the getStatusMessage operation returns a customized status message.

```
namespace ns = "http://www.open-xl.org/hams/devices";
...
service BinaryLightService {
  var $binaryLight as ns:BinaryLightType;
  ...
  public operation getStatusMessage()
    as ns:StatusMessage {
    return <message> The binary light is turned
      {$binaryLight/status}. </message>;
  }}
}
```

Listing 3: BinaryLightService.sll

The BinaryLight service is an excellent example for lightweight services with a restricted set of methods and functionality. In general, the following middleware metrics are measured within this section: ART, WIPS and Parallel Requests in Execution.

5.2.2 The Average Response Time (ART) Metric

The first measurement scenario shows how the Average Response Time (ART) of a middleware (configuration) is basically affected by the number of concurrent clients (CWT). For this purpose the CWT of the RCE was set to 1, 5, 10, 25, 50, 100, 200, whereas the used PWT count inside the middleware remains static (150 workers). As shown in Figure 9 and Table 3, the pre-build ME OpenXL middleware has the worst ART rates. This basically correlates with the specified NFR optimization requirements (see section 4.1 and 4.2) during the middleware configuration process whereupon only the footprint minimization of this configuration had a high priority.

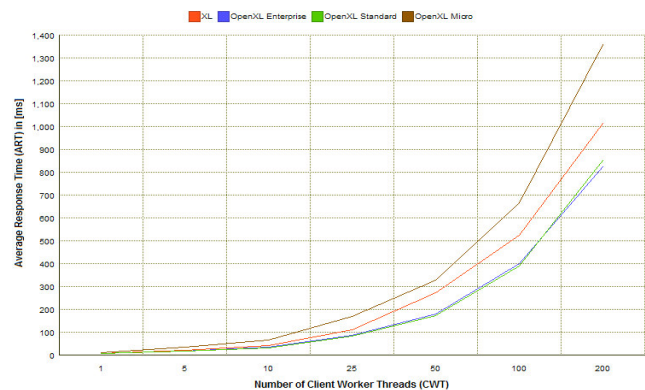


Figure 9: ART Measurements BinaryLight Scenario

The ART performance of the static XL system is below that of the ME configuration (e.g., ART 524 ms vs. 665 ms for a CWT count of 100). A stepwise reconfiguration of the ME middleware towards a SE and EE version via the FCW resulted in a general improvement of the ART rates for all numbers of CWTs. It was observed in the BinaryLight scenario that the results of the SE and EE configuration are basically comparable. A reason could be that DB support is not needed in this scenario and both use the same XQuery engine module.

Table 3: ART Measurements BinaryLight Scenario

		Response Times in [ms]																							
		1 CWT			5 CWT			10 CWT			25 CWT			50 CWT			100 CWT			200 CWT					
		avg	min	max	avg	min	max	avg	min	max	avg	min	max	avg	min	max	avg	min	max	avg	min	max	avg	min	max
XL	6	4	118	21	5	765	40	6	1697	109	6	3146	271	6	10270	524	12	16332	1013	157	20494				
EE	6	4	221	17	5	2026	34	6	384	86	7	657	180	8	655	399	13	1582	823	56	2507				
SE	6	4	228	16	5	377	32	6	221	82	8	807	173	7	700	388	8	2406	852	12	2903				
ME	10	8	277	33	12	311	66	14	455	168	16	726	329	40	1147	665	206	1941	1360	701	3609				

Nevertheless, as mentioned before, the ME configuration can relatively keep up with the other performance-optimized configurations which is due to the simplicity of the BinaryLight scenario. Comparable measurement results are observed in section 5.2.3.

5.2.3 Web Interactions per Second (WIPS)

The second measurement scenario uses the `getStatusMessage` operation of the `BinaryLightService` in order to illustrate the WIPS behavior of the middleware configurations while changing the client thread count (CWTs) as in chapter 5.2.2. As shown in Figure 10 and Table 4, the WIPS rates in this scenario are the highest for the SE and EE configurations. This are expected results because SE and EE were optimized for high performance (ART and WIPS rates) and both are using almost the same modules (except the DB module which did not come into operation in this scenario). It was observed that higher CWT counts cause an increased overhead (synchronization, etc.) inside these middleware configurations whereby the WIPS of the SE and EE configuration start to converge to the results of the static XL platform. The result of XL is in between those of the ME and the SE/EE configuration.

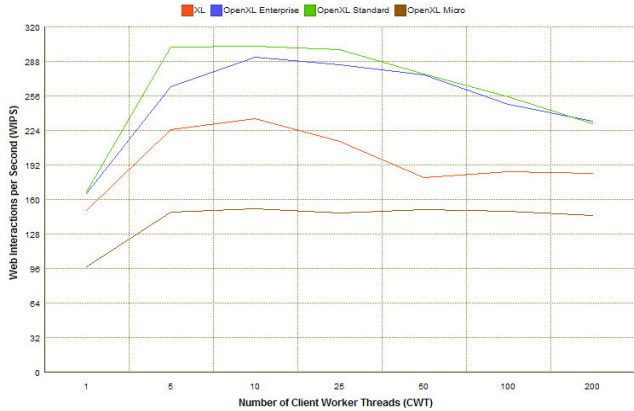


Figure 10: WIPS Measurements BinaryLight Scenario

The ME has the lowest WIPS rates in the complete CWT spectrum. This is a direct result of the NFR input [WIPS, ART] = [DON'T CARE] during the middleware configuration process which allowed the FCW a high-priority optimization with regard to memory footprint and platform size. In general, significant differences in the measurement results (larger than x2) between footprint-optimized, static and performance optimized middleware are not observable in this simple scenario.

Table 4: WIPS Measurements BinaryLight Scenario

		Web Interactions per second (WIPS)						
		1 CWT	5 CWT	10 CWT	25 CWT	50 CWT	100 CWT	200 CWT
XL		149.76	224.87	235.09	214.16	180.42	185.55	184.3
EE		165.16	264.4	291.79	284.75	275.26	248.43	232.46
SE		166.99	301.41	302.39	298.86	275.96	255.02	230.55
ME		97.42	148.26	151.29	147.61	150.9	149.11	144.88

5.2.4 Parallel Requests in Execution (PRIE)

The last sub-scenario of this section illustrates a special metric (which was not defined in section 2.2) of the middleware for a CWT count of 200 and a PWT count of 150 which can help to understand the preceding measurement results and the subsequent results of the more complex measurement scenarios. It shows how many requests are concurrently in execution on each configuration over the time axis. Please note that this metric does not include request backlogs of middleware server sockets.

As depicted in Figure 11, the ME and XL system have almost constantly high PRIE rates which is in the case of ME a bottleneck problem of the underlying XQuery engine (concurrency is not well supported). In the case of XL, it is a PWT-switching overhead problem whereas constantly high PRIE rates generally have more negative effects on the ART and WIPS values for higher CWT counts. If such a behavior of modules is well known, their specific effects on the overall metrics can be described before and considered during the deployment process of a specific middleware (e.g., which is basically a reason why the XQuery Engine is not an option for high-performance enterprise configurations as they are needed in the sub-sequent sections). In contrast to that, the SE and EE configuration have peak phases, whereby the amount of accepted requests does not reach the maximum PWT count over the whole time and the server socket backlog of the Jetty module grows if needed. As shown, the backlog and PWT strategy are also influenceable by the used http server which generally affects the thread-switching overhead of a platform (e.g., SE/EE) and therefore the overall platform performance.

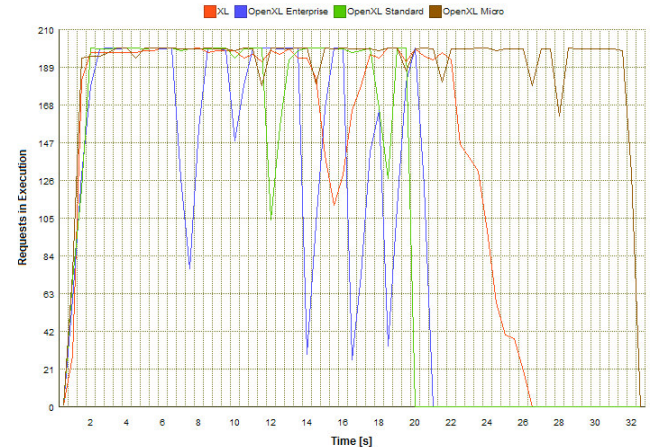


Figure 11: Parallel Requests in Execution (PRIE)

Although the BinaryLight measurements can help to determine basic directions for middleware characteristics, this scenario is not qualified for comparable and transferable measurement results in the enterprise context. In this context, a standardized benchmark is needed which better illustrates the stress-characteristics and metrics of a specific platform.

5.3 TPC-W: A Representative Measurement Scenario for Middleware Adaptations

The TPC-W Benchmark is a transactional web benchmark which uses an e-commerce application (ShopService) in order to determine the performance metrics of the underlying system. Within this paper, the TPC-W Benchmark is utilized to evaluate the OpenXL middleware. For this purpose, the required TPC-W

ShopService was implemented using the SLL_p language and afterwards deployed on all OpenXL middleware configuration (ME, SE, EE) as well as on the static XL Engine which provides a specific interface for the deployment of SLL_p files. It is important to mention that the deployment of the first TPC-W ShopService implementation failed for the OpenXL ME configuration for which a required module (the XQuare Fusion XQuery Engine) was responsible. In contrast to the BEA XQRL and Saxon XQuery engine, the current XQuare Fusion implementation (version 1.1.1) is not in a mature state. Complications existed to get the TPC-W ShopService running on this engine. Several expressions had to be rewritten, before the service was deployable. It was also a major problem that the XQuare Fusion XQuery Engine does not implement the complete XQuery specification. For example, simple expressions such as `count($cart/item)=0`, `$customers/customer[1]`, `$items/item[id=$id]`, `sum(...)`, etc. are not supported. In order to get comparable measurements, an optimized ShopService implementation which is running on all platform configurations was used.

5.3.1 The QueryItems Measurement Scenario Shows Main Effects of NFR-driven Middleware Adaptations

Within this scenario, the performance WIPS/ARTs metrics of all systems (ME, SE, EE and XL) are evaluated while changing the underlying item list size of the TPC-W ShopService. In this context, the `queryItems` operation of the shop service is tested which makes use of the storage implementation for search queries in the item list. It is expected that the EE edition has the best performance results because it was optimized for performance (especially WIPS) in combination with a database system (see section 4.2). Basically, the `QueryItems` measurement scenario tries to illustrate how the usages of specific OpenXL modules (especially of the persistence implementations) and the NFR optimization criteria affect the overall middleware metrics in a significant way. Therefore, the item list (IL) size varies from 100, 200, 400, 800, 1600, 3200 items. As mentioned before the XL, ME and SE system use the file system and memory as direct storage. In contrast to that the EE is able to use the memory in combination with a MySQL database. For all measurements within the `queryItems`-scenario the number of PWTs is 150 and the corresponding number of CWTs is 100.

5.3.1.1 Implementation Of The QueryItems Operation

As shown in the first lines of Listing 4, the ShopService (which is written in SLL_p) declares the `tpcw` namespace and an external `$items` resource. Moreover, the `queryItems` operation is illustrated which accepts a searchItem (`sItem`) parameter as input. Basically, the `queryItems` operation uses the search item in combination with a `For`, `Let`, `Where`, `Order by`, `Return` (FLOWR) expression in order to find matching shop items. Although, it would be possible that the resource declaration of the shop-item list directly points to a XML file

```
resource $dbshop as tpcw:Items = "tpcw/items.xml"
```

or a database

```
resource $dbshop as tpcw:Items = "mysql://localhost/tpcwshop"{
  "User"="shop";
  "Password"="shop"; ...
};
```

inside the service document, the TPC-W ShopService implementation uses the more flexible external declaration of resources which utilizes the late resource binding mechanism of the OpenXL middleware during the service deployment process.

With regard to the ME and SE configuration, only XML-file based resources are supported (see section 2.3) which is basically ensured via the validation of the xSLL model of each deployed service against the XML Schema file of the concrete middleware configuration⁵. In contrast to that, the EE platform supports all resource declaration types (XML-file, database, external) and does not cause deployment errors for all aforementioned resource types.

```
namespace tpcw = "http://www.open-xl.org/examples/shop";
external resource $items as tpcw:Items;
...
service ShopService {
...
public operation queryItems($sItem as tpcw:Item)
returns ($sItem as tpcw:Item) {
sItem =
for $cItem in $items/item
where (empty($sItem/title/text())
or $cItem/title = $sItem/title)
and (empty($sItem/publisher/text())
or $cItem/publisher = $sItem/publisher)
and (empty($sItem/subject/text())
or contains($cItem/subject, $sItem/subject))
and (empty($sItem/isbn) or $cItem/isbn = $sItem/isbn)
and (empty($sItem/discount/text())
or number($cItem/discount) > number($sItem/discount))
and (empty($sItem/price/text())
or number($cItem/price) < number($sItem/price))
and (empty($sItem/id/text()) or $cItem/id = $sItem/id)
and (number($cItem/stock) > 0.0)
return <item>
{ $cItem/id }
{ $cItem/title }
{ $cItem/publisher }
{ $cItem/author_id }
{ $cItem/subject }
{ $cItem/pub_date }
{ $cItem/isbn }
{ $cItem/discount }
{ $cItem/price }
</item>;
}
}
```

Listing 4: ShopService queryItems() Operation

5.3.1.2 Enormous Improvements of the Middleware Average Response Times (ART)

Within this sub-section, the average response times (ARTs) of all configurations are measured for the aforementioned `queryItems` operation of the TPC-W ShopService. Figure 12 shows the ratio of ARTs and specific item list sizes.

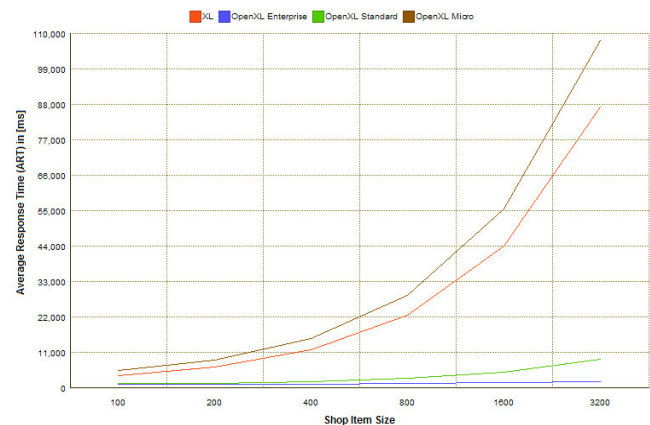


Figure 12: ART Measurements TPC-W queryItems Scenario

⁵ This XML Schema comprises rules for all allowed instructions. In the case of the XL platform the SLL_p2XL converter demanded an explicit endpoint for the external resource and ensured a successful deployment of the TPC-W ShopService.

It is obvious that the ME configuration scales significantly bad for larger item lists. The main reason for that is the implementation of the XQuare Fusion engine which can not be executed concurrently. In the current state, the XQuare Fusion implementation does not support a declaration of external query variables which requires a (de)serialization of documents each time a query is executed. As a result, a bottleneck exists during the processing of an XQuery expression which causes a substantial downgrade of the ART/WIPS metrics. The ME performance results correlate with the NFR input during the FCW process which allowed a [DON'T_CARE] optimization for performance metrics and illustrate the importance of a NFR-based optimization if performance must be explicitly high. The ART results of the static XL Platform are basically comparable with those of the ME platform. As shown in Figure 12, the reconfiguration of the ME platform as SE platform results in an enormous improvement of the overall ART rates which is basically due to the usage of the Jetty http server in combination with the Saxon XQuery engine. The additional exchange of the persistence module (usage of a database) for the EE provided the best ART rates within this measurement scenario.

Table 5 illustrates that the x32 increase of the item list (IL) size during the EE middleware measurement (from 100 to 3200 shop items) only caused a less than x2 increase (from 1117ms to 1919ms) of the ART values. Comparing the ART increase of the SE platform (from 1296ms to 8925ms which is equivalent to x8) with the results of the EE configuration, it can be reasoned that the performance advantage of x4 basically results from the usage of a database in this use case.

Table 5: ART Measurements TPC-W queryItems Scenario

	Response Times in [ms]								
	100 IL			200 IL			400 IL		
	avg	min	max	avg	min	max	avg	min	max
XL	3662	1746	26638	6423	4036	27151	11766	4755	32584
EE	1117	87	6393	1102	110	5271	1144	134	5788
SE	1296	564	3448	1457	7	2882	1943	10	3085
ME	5337	1129	7583	8574	977	10004	15357	1130	16344
	800 IL			1600 IL			3200 IL		
	avg	min	max	avg	min	max	avg	min	max
XL	22573	7220	49183	44119	14785	71423	87324	20544	124444
EE	1263	18	5791	1511	112	7737	1919	211	6557
SE	2908	873	4086	4887	1081	6048	8925	1271	10946
ME	28691	1766	30041	55565	2842	57139	107878	4962	110836

This confirms practical experiences, but additionally illustrates that an automated deployment process can profit from measurement data in order to provide a NFR based optimization, e.g., with regard to ART or WIPS.

5.3.1.3 Enormous Improvements of the Middleware Web Interactions per Second (WIPS) Metric

Figure 13 illustrates the ratio between the item list size and the WIPS of a middleware configuration in the queryItem scenario. As expected, the WIPS rates of the ME are the worst in the whole item list size spectrum. With regard to an optimization of the performance characteristics of this configuration, an ART-driven adaptation (see section 4.2) towards SE resulted in much better WIPS rates in this scenario. Basically, the exchange of the http server and the XQuery engine caused this improvement. A further reconfiguration of the SE configuration towards the EE configuration affected the underlying persistence module,

whereby the memory/XML-file based storage of the SE platform was replaced with a memory/database system.

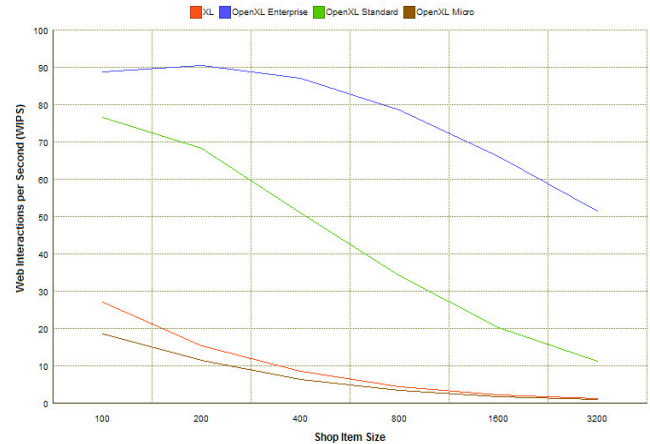


Figure 13: WIPS Measurements TPC-W QueryItems Scenario

As shown in Table 6, the WIPS rates of the EE configuration are approximately **x5** higher than those of SE for an item list size of 3200. With regard to the ME middleware configuration, EE performs **x50** better. Both comparisons show that significant performance improvements are realizable selecting the “right” modules. Speaking in other words, expensive mistakes can occur if a system is not configured well for a specific use case. While the ME configuration is the best solution for an embedded use case, its application in the enterprise use case (e.g., queryItems operation of the ShopService) can cause unacceptable bottlenecks. Basically, the queryItems scenario confirms the FCW results of section 4.2.

Table 6: WIPS Measurements TPC-W queryItems Scenario

	Web Interactions per second (WIPS)					
	100 IL	200 IL	400 IL	800 IL	1600 IL	3200 IL
XL	27.08	15.45	8.44	4.4	2.25	1.14
EE	88.67	90.44	87.16	78.52	66.02	51.55
SE	76.71	68.25	51.09	34.13	20.29	11.13
ME	18.54	11.53	6.43	3.44	1.78	0.92

5.3.2 The AddItemToCart Measurements Show the Need of NFR-driven Middleware Adaptations

The next measurement scenario illustrates the performance characteristics of all configurations using a complex TPC-W benchmark operation and a varying amount of CWTs. The addItemToCart function contains representative types of XQuery expressions, control flow statements, data manipulation statements as well as context binding statements. It is expected for this scenario that the differences between a footprint-optimized platform (ME) and performance-optimized middleware configurations (SE, EE) become significantly apparent. Additionally, the basic disadvantages of a static platform (XL) are illustrated within this scenario which basically emphasizes the necessity of software systems adaptations for varying NFRs.

5.3.2.1 Implementation of the AddItemToCart Operation

Listing 5 provides an overview of the content of addItemToCart function using the first ‘ideal’ implementation of the TPC-W ShopService which was not deployable on the ME configuration (see section 5.3).

```

namespace tpcw = "http://www.open-xl.org/examples/shop";
external resource $items as tpcw:Items;
...
service ShopService throws Exception{
...
var $order_id as xs:integer = 0;
// context which holds session/user specific data
context session {
// declare the master key
master key $sid as xs:string;
// set context timeout (10 minutes)
timeout 600000;
var $customerData as shop:customer; // customer data
var $cart as tpcw:Cart =
<cart><state>unused</state><lineitems/></cart>;
}
...
public operation addItemToCart($sid as xs:string,
    $item as tpcw:Item) as xs:string {
//tries to get a user context for the sid
get session correlation($sid with $sid) pattern NEW;
//initialized?
if($session:cart/state = "unused") {
replace $session:cart/state with <state>created</state>;
insert <total>0.0</total> into $session:cart;
...
insert <id>{ $order_id }</id> into $session:cart;
$order_id = $order_id + 1;
}
...
var $targetItem = $items/item[id = $item/id];
...
if(count($targetItem)<1) {
throw <ShopException>"Item does not exist"</ShopException>;
}
if(number($targetItem/stock) < number($item/quantity)) {
throw <ShopException>Item needs some replenishment
($targetItem/stock) lt {$item/quantity}.</ShopException>;
}
...
replace $items/item[id = $item/id]/stock with
<stock> {number($targetItem/stock) - number($item/quantity)}
</stock>;
var $subtotal as xs:double = (number($item/quantity) *
    number($targetItem/price)) * (1.0 - $discount);
var $tax as xs:double = $subtotal * 0.0825;
var $shipping as xs:double = 3.0 + (1.0 *
    number($item/quantity));
...
var $lineitem as tpcw:LineItem =
<lineitem>
{ $item/id }
...
<tax>{ $tax }</tax>
<shipping>{ $shipping }</shipping>
<total> { $subtotal + $tax + $shipping } </total>
</lineitem>;
insert $lineitem into $session:cart/lineitems;
replace $session:cart/total with
<total>{sum($session:cart/lineitems/lineitem/total)}</total>;
...
}
}

```

Listing 5: ShopService addItemToCart() Operation

For reasons of clarity several statements were omitted. As shown in Listing 5, the operation accepts an item and session id as input. If no sid is handed over, a new sid and context instance are created by the context binding statement (get-correlation). It is important to mention that, for example, the increment of the order_id after the second insert statement automatically causes an exclusive write-lock of the variable for concurrent operation calls. In the sub-sequent lines, the required amount of the line item is balanced with the stock and the total price is calculated. Afterwards, if these steps were successful, the line item is added to the shopping cart and the total shopping cart value is updated.

Generally, the following requirements were made for the AddItemToCart measurements:

- Within each cycle the client gets a new session id (sid) which opens a new context. Thus the corresponding shopping cart has an 'unused' state and is empty. During the operation call, the item with the specified id is added to the shopping cart.
- The scenario increases the count of concurrent client threads (CWT) in the following order: 1, 5, 10, 25, 50, 100, 200.

- The PWT count is 150 and the shop item list (IL) size is 400.

5.3.2.2 The Average Response Time (ART) Metric Shows Main Drawbacks of Static Middleware

This sub-scenario measures the ARTs of all middleware configurations for the addItemToCart operation of the TPC-W ShopService. It compares the static XL system with the optimized OpenXL configurations and tries to illustrate the accuracy and importance of NFR optimizations. As shown in Figure 14, the OpenXL ME configuration has the worst ARTs for all numbers of CWTs. Although, the static XL system provides much better ART results than ME, the fundamental drawbacks of XL are that it neither can be used in the PDA scenario (because its size is not under 10 MB which was evaluated in section 4.1) nor in a gateway and server scenario which require high-performance. In the latter case, the ART results of XL significantly cannot keep up with the SE and EE performance metrics which basically restricts the usage in most of the enterprise use cases. Speaking in other words, the NFR-based adaptation of the ME configuration towards SE and EE via the FCW resulted in the intended adaptation of the underlying middleware NFRs and thus enabled its usage in the enterprise scenarios, whereas XL remains static and is outperformed in each use case scenario by the adapted OpenXL middleware versions.

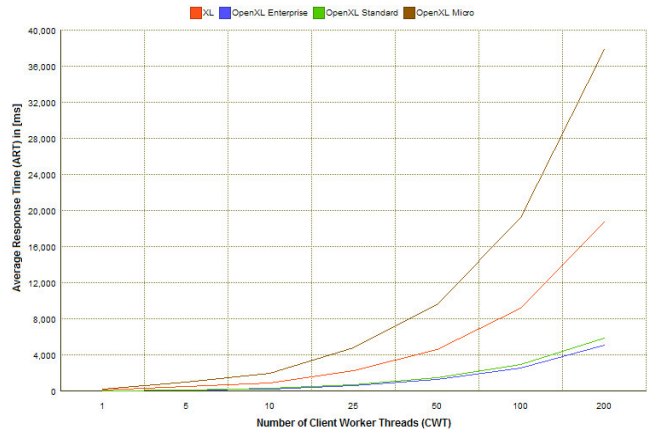


Figure 14: ART Measurements TPC-W addItemToCart

For example, as shown in Table 7 using 10 parallel CWTs the ME has an ART of 1918 ms and the static XL system has an ART of 891 ms which means that the XL ART is approximately x2 times better. In comparison to that the SE has an ART of 273 ms and the EE an ART of 240 ms which shows that both approximately perform x9 times better than the ME and x4 times better than the XL platform.

Table 7: ART Measurements TPC-W addItemToCart

	Response Times in [ms]											
	1 CWT			5 CWT			10 CWT			25 CWT		
	avg	min	max	avg	min	max	avg	min	max	avg	min	max
XL	91	84	227	451	129	3206	891	126	5532	2220	137	12467
EE	27	23	235	108	41	466	240	107	942	616	109	1911
SE	33	28	357	128	56	472	273	111	876	699	77	2610
ME	201	192	415	965	473	1558	1918	922	3650	4790	1143	9030
	50 CWT			100 CWT			200 CWT					
	avg	min	max	avg	min	max	avg	min	max	avg	min	max
	XL	4603	2150	22614	9210	4339	28165	18692	12641	36107		
EE	1236	7	3579	2512	4	5908	5058	1053	12956			
SE	1421	234	3564	2906	111	6577	5837	1430	13310			
ME	9536	1148	16616	19201	1029	34100	37876	10304	71724			

The ratio is approximately the same for all numbers of CWTs. Assuming that a typical enterprise use case requires minimum response times (e.g., under 1 second), these ratios also state the overall hardware cost factor or the number of involved system instances which are at least needed inside a clustered middleware architecture. As a result, a middleware configuration provides the best solution for a specific use case, if it was optimized for a concrete scenario and if the optimization process itself in comparison does not cause too much costs. Costs are, for example, caused by the adaptation or maintenance of middleware. Using ASE, the adaptation of a system can be automated which means that the cost factor remains minimal and an optimized middleware often outperforms static systems.

5.3.2.3 The Web Interactions per Second (WIPS) Metric Confirms the Preceding Measurements

The WIPS measurements of the addItemToCart scenario provide a second metric which is important for the NFR-based optimization of middleware (see section 2.2). In general, high WIPS rates indicate a good performance of the underlying system. As shown in Figure 15, the basic results of these measurements are comparable to those of the preceding sub-section.

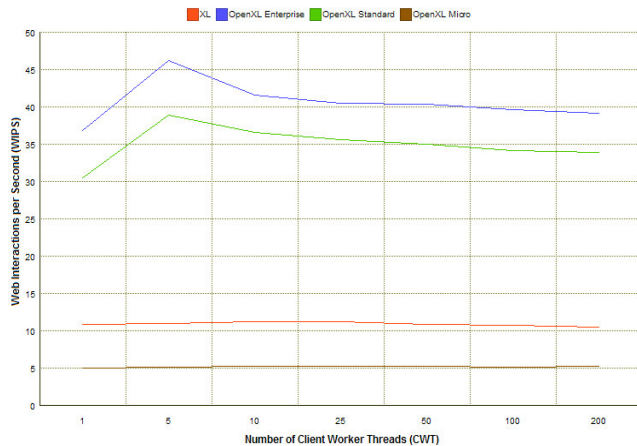


Figure 15: WIPS Measurements TPC-W addItemToCart

Basically, the ME middleware has the worst WIPS rates which is again due to the fact that it was optimized for small size instead of high-performance. The static XL system provides better results than ME, but can not keep up with the performance optimized OpenXL middleware configurations. EE has the best WIPS rates, which results from the additional usage of a database in the server scenario (cp. SE). In this context, the optimization of the ME configuration (which must run on a PDA) and the SE configuration (which is not allowed to use a database) towards EE showed again that automated adaptations are able to change the underlying non-functional properties of a middleware implementation.

With regard to Table 8, the results of the XL measurements for varying CWTs in the complex addItemToCart scenario can be seen as static baseline. In the context of 100 CWTs for example, XL reaches 10.76 WIPS. Although the ME configuration only reaches half of this value (5.18 WIPS), it alone addresses the NFRs (size, footprint) of the PDA scenario (see the metrics of section 4.2). The SE and EE are approximately x3 times (34.2 WIPS) or x4 times (39.62 WIPS) above the WIPS rates of XL. As shown a significant performance advantage was achieved which was basically controlled by the FCW-based configuration process.

Table 8: WIPS Measurements TPC-W addItemToCart

	Web Interactions per second (WIPS)						
	1 CWT	5 CWT	10 CWT	25 CWT	50 CWT	100 CWT	200 CWT
XL	10.91	11.03	11.18	11.22	10.8	10.76	10.48
OpenXL Enterprise	36.79	46.2	41.61	40.47	40.33	39.62	39.19
OpenXL Standard	30.45	38.87	36.53	35.67	35.05	34.2	33.87
OpenXL Micro	4.96	5.17	5.21	5.21	5.23	5.18	5.22

To sum it up, the automated adaptation of middleware for a given set of NFRs is advantageous in many use cases and provides the required flexibility with regard to a heterogeneous system and software landscape.

6. CONCLUSION AND FUTURE WORK

As shown, an ever-increasing number of large software systems require methodologies in order to manage the inherent complexity in a suitable way. With regard to deployment and maintenance aspects, for example, hardly manageable dependencies and inter-relationships of contained software components often exceed a limit of human reasonableness. This includes the complex deployment requirements in the form of non-functional requirements (NFRs) of a system. Basically, there is a need for automating the changes and the deployment of a system according to a given set of NFRs. The present paper addressed the aforementioned automated management of NFRs and introduced a flexible, adaptable Web Service middleware system (OpenXL) which was created (see section 3) using automated software engineering (ASE) techniques. ASE proposed a three-pronged approach which used a high-level programming language, semantically annotated software modules and an automated feature configuration process.

In order to evaluate the OpenXL implementation a standardized benchmark (TPC-W) was utilized (see section 5) which showed that OpenXL can be flexibly adapted given a set of NFRs. In this context, three platform configurations were dynamically configured which all address different use cases in a home automation and management scenario (HAMS): a *micro edition* which must run on limited devices such as a PDA, a *standard edition* which flexibly addressed the NFRs of home gateways and an *enterprise edition* which must run on a high-performance server. As depicted in the measurements section 5, the accuracy of the platform configuration via a feature configurator allowed a flexible fine-tuning with regard to NFRs such as memory footprint, size, startup time, WIPS and ART. Here, a stepwise NFR reconfiguration of the micro edition (towards SE and EE) provided significant performance improvements in the TPC-W benchmark use case which basically showed the achievement potential of an automated NFR optimization using ASE. It was also illustrated that a static system such as the XL platform has several drawbacks with regard to an application in a heterogeneous software landscape. In this context, a specifically optimized and adapted system outperformed the static system in almost every use case. In general, the promising results of this paper illustrate that an automated handling of NFRs can provide significant improvements in the context of a Web Service middleware (re)configuration and that the underlying principles are basically applicable to the NFR management of software systems.

Nevertheless, some limitations of the current OpenXL implementation exist. For example, if the annotated meta-information (feature descriptions, metrics, benchmark data) of the OpenXL modules is not exact or representative enough; as a result, NFR adaptations of a system can fail or (in the worst case) be simply wrong. Additionally, finding the best solution for a

given set of NFRs is a NP-hard problem which still remains unsolved. The handling of the complexity (e.g., via heuristics) is out of the scope⁶ of this paper and was therefore discussed in a separate paper [5]. Moreover, this paper assumed that all services which are deployed on the OpenXL platform are written in the SLL_p language. This is not a basic necessity, because the usage of a service language layer (SLL) [18] also allows, e.g., the deployment of Java or BPEL [29] code. Apart from that the usage of the XQuare Fusion XQuery engine showed that a not mature OpenXL module implementation causes restrictions in the general deployment of services. Well tested and certified implementations can help to avoid these problems.

Future work in the context of ASE will address an extension of the overall available OpenXL modules, for instance, with regard to security, peer-to-peer strategies and communication. Additionally, validation support for module descriptors, a module versioning strategy and more advanced graphical support (editors, etc.) will be provided. In the context of OpenXL, it is intended that the middleware is used for highly-distributed, self-manipulating, genetic algorithms and software systems. Moreover, the existing FXL project [19] will use OpenXL/SLL_p instead of XML pipelines for the description of pipelines/workflows, transformations and software generators. This provides a highly-distributed software engineering (SE) process which addresses the industry-driven reuse of SE components in a service oriented architecture (SOA) and allows a flexible customization of software based on existing standards.

7. ACKNOWLEDGEMENTS

We would like to thank Gustavo Alonso for his support on this paper.

8. REFERENCES

- [1] Smith, W.D., TPC-W: Benchmarking An Ecommerce Solution, Revision 1.2, http://www.tpc.org/tpcw/TPC-W_wh.pdf.
- [2] JBoss, J2EE Application Server, <http://labs.jboss.com/portal/>, Access date: August 23rd, 2006.
- [3] IBM White Paper: The IBM WebSphere software platform and patterns for e-business invaluable tools for IT architects of the new economy, 2000
- [4] BEA WebLogic Server Specifications, <http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/server/specs>, 2003
- [5] Dinger, U. et al: A Semantic Web Services Approach Towards Automated Software Engineering, International Conference on Software Engineering Advances (ICSEA) 2006, to be published
- [6] Sun Developer Network: Java Enterprise Edition, <http://java.sun.com/javaee/index.jsp/>, Access date: August 30th, 2006.
- [7] Platt, D.S.: Introducing the Microsoft.NET Platform, Microsoft Press, 2001
- [8] Kiczales, G. et al: An Overview of AspectJ, Lecture Notes in Computer Science, volume 2072, pages 327-355, 2001
- [9] Popovici, A. et al: Dynamic Weaving for Aspect Oriented Programming, 1st International Conference on Aspect-Oriented Development (AOSD), Enschede, The Netherlands, 2002
- [10] Object Management Group (OMG), <http://www.omg.org/>, Access date: August 30th, 2006.
- [11] MOF 2.0 Query / Views / Transformations RFP, <http://www.omg.org/docs/ad/02-04-10.pdf>, 2002
- [12] Unified Modeling Language (UML), Version 2.0, <http://www.omg.org/technology/documents/formal/uml.htm>, 2004.
- [13] Budinsky et al.: Eclipse Modeling Framework: A Developer's Guide. Addison-Wesley, 2003
- [14] pure::variants variant management, http://www.pure-systems.com/Variant_Management.49.0.html, Access date: August 30th, 2006.
- [15] XFeature – Feature Modeling Tool, <http://www.pnp-software.com/XFeature/>, Access date: August 23rd, 2006.
- [16] Axis Architecture Guide, Version 1.0; <http://cvs.apache.org/viewcvs.cgi/~checkout~/xmlaxis/java/docs/architecture-guide.html>, 2003
- [17] Chamberlin, D. et al: XQueryP: Programming with XQuery, XIME-P 2006
- [18] Donald Kossmann, Christian Reichel: SLL: Running My Web Services on Your WS Platform (Poster). WWW 2005. A longer version appeared in ICWS 2005 (Industry Track). Also see <http://www.open-xl.org>.
- [19] Christian Reichel, Roy Oberhauser, XML-based Programming Language Modeling: An Approach to Software Engineering, Proceedings of SEA 2004, MIT Cambridge, MA, USA, November 2004. Also see <http://www.fxl-project.com>.
- [20] Kay, M.H.: Saxon – The XSLT and XQuery Processor, <http://saxon.sourceforge.net/>, 2005
- [21] The XQuare project, <http://xquare.objectweb.org/>, Access date: August 30th, 2006.
- [22] Chamberlin, D. et al: XQuery 1.0: An XML Query Language, <http://www.w3.org/TR/xquery>, W3C Working Draft, 2001
- [23] Berners-Lee, T. et al: Semantic Web, Scientific American 284, pages 34-43, 2001
- [24] Akkiraju, R., Farrell, J., Miller, J.A., Nagarajan, M., Schmidt M-T., Sheth, A., Verma, K. Web Service Semantics - WSDL-S, Technical Note, Version 1.0, April 2005, [http://www.alphaworks.ibm.com/g/g.nsf/img/semanticdocs/\\$file/wssemantic_annotation.pdf](http://www.alphaworks.ibm.com/g/g.nsf/img/semanticdocs/$file/wssemantic_annotation.pdf)
- [25] OWL Services Coalition: OWL-S: Semantic Markup for Web Services, <http://www.daml.org/services/owl-s/1.0>, 2003
- [26] Roman, D. et al: Web service modeling ontology – standard, <http://www.wsmo.org/2004/d2/v0.3/>, 2004.
- [27] Mort Bay Consulting: Jetty-Java HTTP server and servlet container, <http://jetty.mortbay.org/>, 2002.
- [28] Knopflerfish – Open Source OSGi, <http://www.knopflerfish.org/>, Access date: August 30th, 2006.
- [29] Curbera, Y. et al: Business Process Execution Language for Web Services, Version 1.0, 2002, <http://www-106.ibm.com/developerworks/library/ws-bpel/>
- [30] Florescu, D., et al.: XL: An XML Programming Language for Web Service Specification and Composition, Computer Networks Journal, Volume 42, Issue 5, August 2003.

⁶ In the context of the OpenXL use case, the overall size of registered modules was less than 50, whereby the complexity of optimization problems basically did not reach a critical level.