IDP Documentation

Lucas Köhler

Development of a Java-based service-oriented middleware for the dynamic reconfiguration of production systems‘ PLC-programs

Advisor:	Dr.-Ing. Daniel Schütz
Start:	01.10.2016
Submission:	13.03.2017

Lehrstuhl für
Automatisierung und
Informationssysteme
Prof. Dr.-Ing. B. Vogel-Heuser
Technische Universität München
Boltzmannstraße 15 - Geb. 1
85748 Garching bei München

Telefon 089 / 289 – 164 00
Telefax 089 / 289 – 164 10
http://www.ais.mw.tum.de

1 Project Description

This chapter consists of three sections. The first one contains the original project description. The second section redefines parts of the project's goals in order to better suit the project stakeholders' interests. In the third section, the functional and non-functional requirements of the developed software are described.

1.1 Original Description

The goal of the project is the dynamic reconfiguration of a production facility's PLC program. Thereby, the used production facility is the extended pick and place unit (XPPU) in laboratory scale. The production facility's control program is executed in real time on the PLC. Thereby, there already is an ISA-88 model that describes the facility's modules and functions. Furthermore, there is a Java library that allows the setting and reading of variable values on the PLC during runtime.
To achieve the desired functionality, the project is split into two parts. The first part consists of the development of a middleware in Java. This middleware will run on a Raspberry Pi. The middleware allows to access the PLC's variables. Thereby, the middleware allows to set information regarding operations or sequences thereof on the PLC. This allows to execute the specified operations on the PLC without the need to change the PLC's program code. To allow this, a suitable control program for the PLC is implemented in a parallel project. In order to get a suitable middleware, the ISA-88 model of the production facility is used to automatically generate a use case specific instance of the middleware. The model is provided as an Eclipse Modeling Framework (EMF) model. Therefore, EMF is used as a basis for the instance generation.
The second part of the project consists of the development of an Android app. This app uses the middleware to allow a user the controlling of the facility in two different modes. In an automatic mode, the user can define a sequence of operations that are executed chronologically by the PLC. The second mode allows the execution of single operations in a manual operating mode. During the implementation of the app, it is important to carefully separate the user interface and the business logic in order to allow a reuse of the logic for other applications in the future.
During the implementation of the project, the code is documented in a complete and clean way in order to simplify code reuse in the future.

1.2 Webservice Instead of Android App

In the planning phase of the project it became clear that the Android app is not needed except for simple show purposes. Instead the need for a REST webservice arose for mainly three reasons. First, a REST webservice allows to integrate security features such as authentication in the future. This is an already planned step that will be done after this project's finalization. Second, the AIS chair cooperates closely with the “Karlsruher Institut für Technologie” (KIT) and for them a REST webservice offers excellent possibilities to access the middleware and thereby the PLC remotely, as well as integrate the middleware's functionality in their own systems. Third, a REST webservice offers a platform and technology independent interface to use the middleware's functionality. This allows the future development of all kinds of tools for various platforms and use cases.
Therefore, it was decided to omit the Android app and use the freed-up time to implement a REST webservice that allows to access the middleware's functionality by using HTTP requests. The webservice is a part of the middleware and runs on the Raspberry Pi, too.

1.3 Requirements

In this section, the functional and non-functional requirements of the developed middleware are stated and explained.

1.3.1 Functional Requirements

This subsection describes the functional requirements of the middleware.

Generate runtime instance

The middleware is able to generate a runtime instance for a specified PLC. Therefore, the middleware uses two input files. The first one is the production facility modeled according to the ISA-88 standard. The model is provided as an EMF model. The second file is the production facility’s operation list. This list contains all available operations that the user can execute on the PLC. The operation list describes where the operation is located in the provided ISA-88 model. In combination with the ISA-88 standard, this allows to define operations once in a repository in the ISA-88 model and then use them for multiple suited units in the production facility. Thereby, every operation usage in the operation list is unambiguously described by a unique id, a name, and an unambiguous path which describes how and where the operation is used in the facility’s model. The unique id is used to execute the operation on the PLC: By telling the PLC the id it knows exactly which operation needs to be executed at which part of the production facility. The path is resolved against the provided model to get a human readable and understandable path.
The runtime instance is generated from the files during the start up of the middleware. This allows to use the middleware for different production facilities without any changes to the

middleware's source code. Only a suitable model and operation list need to be provided and the middleware needs to be restarted to be compatible with a different facility. Furthermore, the ISA-88 model is kept available during runtime and can be requested by the user. The available operations are kept as suitable objects to allow usage at runtime by the middleware as well as making a list of all operations available to the user.

Manual mode

The middleware implements a manual execution mode which allows the user to start operations in manual operating mode on the PLC. Thereby, the operation's unique id is set on the PLC and a flag is triggered that starts the execution if the production facility is in manual execution mode. Otherwise, the operation cannot be executed and the requesting user is notified about this. Furthermore, the manual mode supports to hold, restart, and abort currently executing respectively holding operations by setting the appropriate variables on the PLC. Thereby, start, hold, restart, and abort correspond to the equally named state transitions in the ISA-88 standard.

Automatic mode

The middleware implements an automatic execution mode which allows the user to start a batch (sequence) of operations in their given order. A given batch contains 1 to 20 (both inclusive) operations. The middleware writes the given unique operations ids to an array on the PLC and triggers the flag that starts the execution if the production facility is in automatic mode. Otherwise, the batch cannot be executed and the requesting user is notified about this.
The execution of the batch works automatically without any further input of the user besides the initial configuration: Whenever a batch operation finishes successfully, the next operation starts automatically. When the last operation has finished successfully, the batch execution was successful. If the execution of a batch operation fails, the following operations will not be executed and the batch execution has failed.
Similarly to the manual mode, the automatic mode, too, supports to hold, restart, and abort a batch. Thereby, the middleware only needs to tell the PLC that the current batch needs to be held, unheld, or aborted. The current executing operation does not need to be specified by the user but is determined automatically. Thereby, start, hold, restart, and abort correspond to the equally named state transitions in the ISA-88 standard. If an operation execution of a batch is aborted, the whole batch counts as aborted. From that it follows that the operations following the canceled operation cannot be executed without starting a new batch.

Emergency Stop

For safety reasons, every production facility must be capable of being instantly stopped with an emergency stop. This behavior must also be represented in the middleware. If an emergency stop occurs, no more operation execution related requests can be processed until the emergency stop is resolved physically at the production facility. Concretely, all requests regarding start, hold, restart, abort, and execution mode switches are denied. If an emergency stop occurs while an operation is executed, its execution is aborted and cannot be restarted after the emergency stop was resolved. Therefore, if the operation was part of a batch, the whole batch is aborted. Also, emergency stops and their resolutions are registered in the history and their occurrences made known to the user.

Check pre- and post-conditions

Whenever an operation is executed its pre-conditions have to be checked before the execution is started: The production facility must be in such a state that all pre-conditions of the operation are met. If this is not the case, the operation execution must not be started and the execution counts as failed.
After the execution was finished, the operation's post-conditions are checked on the production facility. Only if all conditions are met, does the operation execution finish successfully. Otherwise the execution has failed.
An operation's pre- and post-conditions are specified in the ISA-88 model describing the production facility.

Well-defined OPC UA interface to the PLC

There is a clearly defined interface of OPC UA variables that is used by the middleware to communicate with the PLC controlling the production facility. This interface must provide all functionality needed to properly control the production facility in manual and automatic mode (see former requirements). Furthermore, the interface must offer the possibility to get information about the execution results of operations executed on the production facility. The detailed implementation of this interface can be found in section 2.3.

Receive execution feedback from the PLC

For both execution modes, the middleware needs to be informed of the operation execution results. For every operation executed on the PLC, the middleware subscribes to a result variable belonging to the operation. In the manual mode, this equals exactly one subscription to the manual operation's result variable. In the automatic mode, the middleware subscribes to one result variable for every operation in the batch. The result variables are also contained in the batch array. Whenever a result variable of a currently executing operation is changed, the middleware is notified of the new value and writes the appropriate entry to the history.

REST webservice

The middleware offers a REST webservice to make its functionality available to the user. All methods are called by simple HTTP POST or GET calls with the required parameters as URL parameters (e.g. the operation id in manual mode to execute an operation). Answers contain relevant data in the HTTP response's body. The webservice offers methods that offer the following functionality:

- Get operation list
- Set execution mode to manual
- Set execution mode to automatic
- Start batch execution (automatic mode)
- Start operation execution (manual mode)
- Hold execution (both execution modes)
- Restart execution (both execution modes)
- Abort execution (both execution modes)
- Get ISA-88 instance model
- Get complete execution history

- Get execution history filtered by execution id
- Get execution history for entries newer than a given timestamp
- Get execution history filtered by operation id
- Get execution history filtered by module name that the operation belongs to

History

The middleware keeps a history of all execution related events. This means that for every start, hold, restart, abort, or mode switch command a history entry is created. Whenever the result of an operation execution changes on the PLC and the middleware is notified, a history entry is created to indicate the (un)successful finish of the execution operation. The start and finish of an operation batch are indicated with separate entries (additional to the entries for every operation inside the batch). Furthermore, emergency stops on the PLC are tracked in the history. Every history entry contains at least a timestamp and the executed command. The history reflects the state of the production facility and is used to determine whether a user specified command can be executed or not. If it cannot be executed, the history specifies the reason.

Execution ids

For every execution of an operation (manual mode) or batch of operations (automatic mode) a unique execution id is generated. Its purpose is to be able to tell which history entries belong to an execution cycle of an operation or batch. For example, an operation is first started, then held, then restarted, and finally finishes successfully. The four resulting history entries have the same execution id to indicate that they belong to the same execution.
Furthermore, when the user wants to hold, restart or abort an operation or batch, he must provide the correct execution id. This makes sure that only the intended operation's or batch's execution state is changed.

Operation list

The middleware's webservice offers a method that delivers a list of all available operations of the current instance. The list is delivered in the JSON data format and contains for every operation:

- The operation's unique id
- The operation's name
- The operation's resolved usage path in the instance model

1.3.2 Non-functional Requirements

This subsection describes the middleware's non-functional requirements.

Implementation in Java

The middleware is implemented in Java 8 and runs on a JVM compatible with the Raspbian Linux operating system running on the Raspberry Pi.

Use OPC UA as communication protocol to the PLC

The middleware uses the OPC Unified Architecture (OPC UA) communication protocol to set and read variables on the PLC.

Use Prosys OPC UA Java SDK

The middleware uses the Prosys OPC UA Java SDK [1] to write and read variables on the PLC.

Extensibility: Webservice security

The middleware must provide the possibility to be extended by security features, such as authentication and integrity, with reasonable effort.

Flexible logging

The middleware logs important events such as errors, commands by the user, internal problems, etc. Thereby, the logging's detail level is configurable without changing the middleware's source code.

Proper code documentation

The source code is commented in a way that other developers understand the code and can perform further development of the middleware in the future.

2 Developer Documentation

The developer documentation contains information needed by developers to understand the system and further develop the middleware.

2.1 Software Architecture

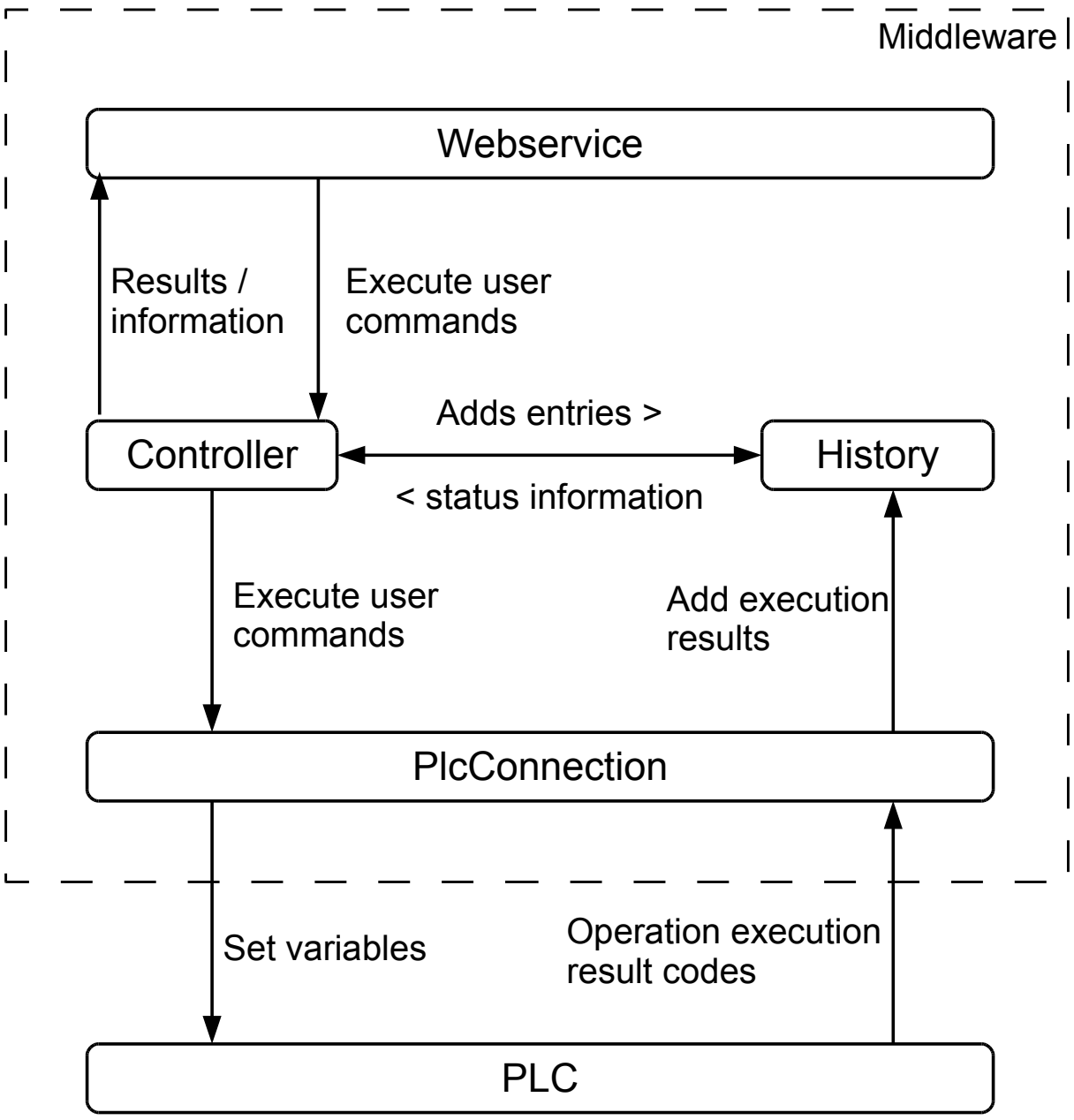

Figure 1: Middleware Software Architecture

The concept of the middleware's software architecture can be seen in Figure 1. The broken line shows the system border of the middleware. This shows that the PLC is not part of the middleware but the PLC is shown in the diagram to indicate the middleware's interactions with the PLC. The middleware uses a layered system which consists of three layers. The highest layer, the service layer, only contains the *Webservice*. It makes the middleware's functionality available to users. The webservice accepts the users' HTTP requests and sends back the appropriate HTTP responses after the requests have been processed by the lower layers.
Therefore, the *Webservice* forwards incoming requests to the *Controller*. Together with the *History* it forms the business logic layer. This layer contains the logic to determine how incoming requests of users are processed. Thereby, the *Controller* reads the content of the HTTP requests and determines how to proceed. In case of operation execution related requests, the Controller asks the *History* whether the requested action is possible. If it is, a new entry is added to the *History* and the *Controller* calls the appropriate method of the *PlcConnection*. For other requests (such as the operation list or the history content) the *Controller* gathers the information and composes a response. In both cases a HTTP request is generated which is returned to the *Webservice*.
The middleware's lowest layer, the communication layer, consists of the *PlcConnection* component. It knows the communication protocol to the PLC and sets the appropriate variables to execute the users' requests. Furthermore, it subscribes to the result variables and is notified about their changes by the PLC. Incoming result codes resulting from the users' operations are written to the *History*.

2.2 Software Design

This section describes the software design of the middleware, thereby, being more detailed than the higher level architecture description in section 2.1.
Figure 2 shows the class diagram describing the middleware's software design. The diagram shows the most important classes and interfaces with their most important attributes and methods. Details which are not needed to understand the design are omitted in the diagram because a complete diagram with all classes, interfaces, attributes, and methods would be confusing to look at and would not help anymore with understanding the design. A complete reference of all classes and their public methods is available on the attached CD of this documentation as Javadoc.
An important design goal during the middleware's development was *loose coupling*. To achieve this, components of the middleware only have dependencies on interfaces of other components instead of the implementation classes. For example, the *ControllerImpl* class has a dependency on the *History* interface but not the *HistoryImpl* class. Only at runtime, the *ControllerImpl* works with an instance of *HistoryImpl*. This improves extensibility and maintainability of the middleware because the concrete implementation of a component can be changed or replaced without the need to change any classes which depend on that component.
The following paragraphs describe the components of the class diagram from top to bottom. The *App* class has three important responsibilities. First, it uses the *Configuration* to load the

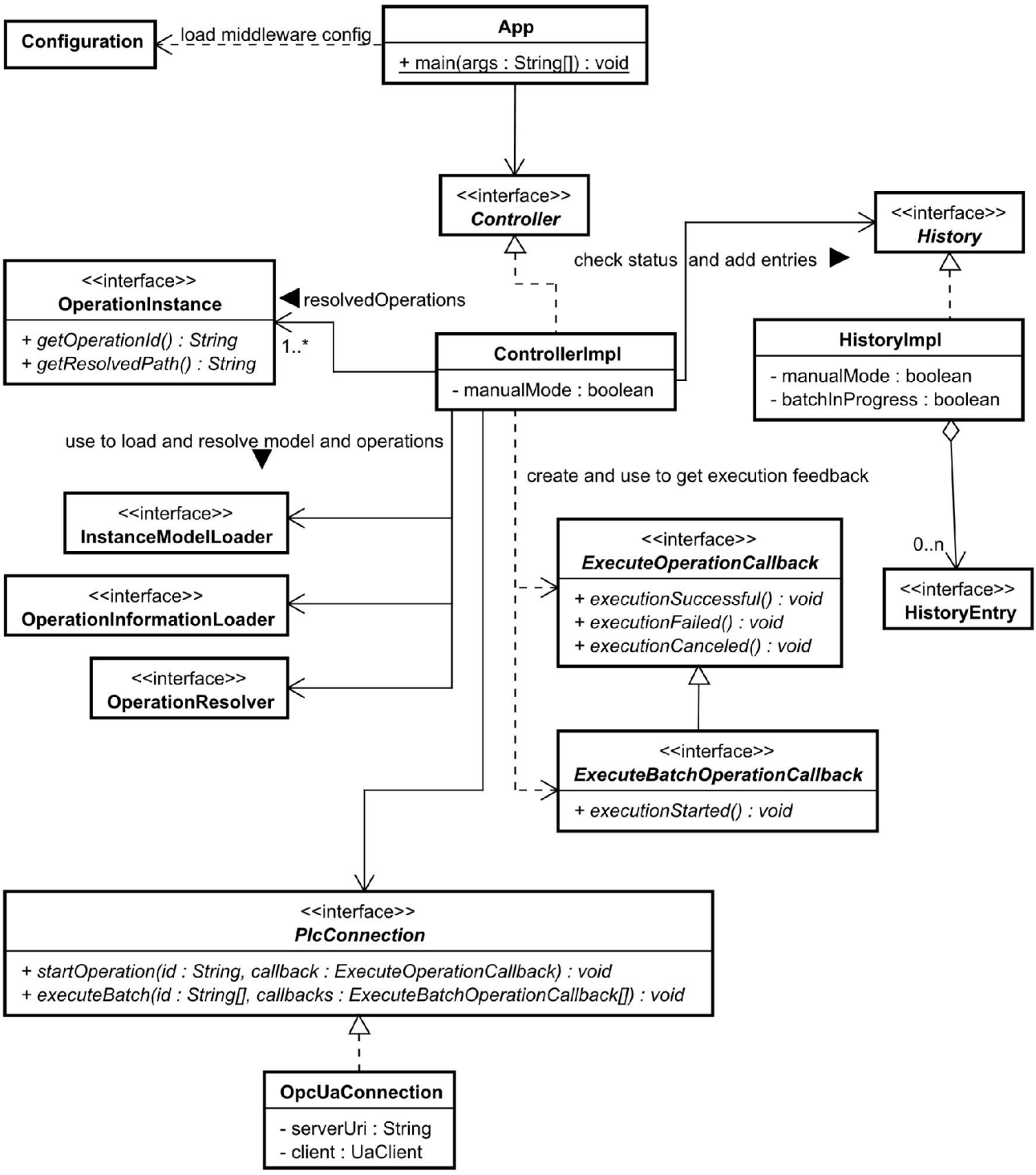

Figure 2: System Design Class Diagram

middleware's configuration file. Second, the class creates needed instances to run the program (e.g. the *Controller* and *History* instances). Finally, the *App* uses the micro framework Spark [2] to set up and configure the webservice by defining its routes and linking them to the corresponding methods of the *Controller*.

The *ControllerImpl* is "the heart" of the middleware. It uses instances of the shown interfaces *InstanceModelLoader*, *OperationInformationLoader*, and *OperationResolver* to load the ISA-88 model, and the available operations and resolves the operations against the model. The

resolved operations are kept as *OperationInstances*. The *ControllerImpl* uses the middleware's *History* instance to check whether user requests which influence the execution of operations are possible and adds the corresponding entries to the history. Furthermore, the *ControllerImpl* gathers and processes the history's contents for user requests regarding the history's content. The *ControllerImpl's* private field *manualMode* is used to keep track whether the facility is currently running in manual or automatic mode. To execute operation execution related requests on the PLC, the *ControllerImpl* uses an instance of *PlcConnection*.
The *HistoryImpl* contains the *n* newest *HistoryEntries* where *n* is the history's maximum capacity which is defined in the middleware's configuration file. The manualMode flag has the same purpose as the one in the *ControllerImpl*. The *batchInProgress* flag is used to keep track whether a sequence of operations is currently in progress. Consequently, this is only relevant while the facility is in automatic mode.
The *OpcUaConnection* class is the implementation of the *PlcConnection* interface. The *PlcConnection* defines all methods necessary to execute, hold, restart, and abort operations and operation batches on the PLC. The most important methods are *startOperation* and *executeBatch*. These are used to execute operations in manual respectively automatic mode. Both methods accept suitable callbacks as parameters. These callbacks, defined in the *ExecuteOperationCallback* and *ExecuteBatchOperationCallback* interfaces, are notified when the *PlcConnection* is notified by the PLC with an execution result code. The *OpcUaConnection* uses an instance of *UaClient* from the Prosys OPC UA SDK to write and subscribe to variables on the PLC using the OPC UA protocol.

2.3 OPC UA Interface

In this section, the variables which are used to communicate with the PLC are described. The variables are made available over OPC UA. Therefore, they are identified by a unique identifier within a given namespace. The namespace index for all used variables is **4**. Because all variables are part of the PLC's Global Variable List (GVL), all variable identifiers start with "**GVL.**". To achieve better readability of the identifiers, this prefix will not be included in the following documentation but must be added to address the variables on the PLC. The variables are documented in the format **<identifier> : <datatype>** with the variable's explanation following in the next line(s).

2.3.1 General Variables

EmergencyStop : BOOL
TRUE, if the facility's emergency stop button was pressed. Returns to FALSE when the emergency stop is resolved. The middleware subscribes to this variable to get notified whenever an emergency stop occurs or is resolved. Not writable by the middleware.

AutomaticExecution : BOOL
TRUE, if the PLC is in automatic execution mode, FALSE otherwise. Writable by the middleware.

2.3.2 Manual Mode Variables

ManualOperation : S_Operation
Contains the information needed to execute the set operation in manual mode. The operation is not set directly as a S_Operation object but the variables of the S_Operation are addressed directly by the middleware.

ManualOperation.strID : STRING
The unique id identifying the operation to execute. Writable by the middleware.

ManualOperation.iStart : INT
Whether the operation should be run or held. 1 means run, 0 means hold. Writable by the middleware.

ManualOperation.strResult : STRING
Contains the execution result of the operation. Whenever a manual operation is executed, the middleware subscribes to this variable to be notified about the execution result. The middleware can write this variable but should not need to do so.

ManualOperation.bAbortOperation : BOOL
Internally set by the PLC when the currently executing manual operation should be aborted. The middleware can write this variable but should not need to do so.

ManualOperation_START_command : BOOL
If the PLC is in manual execution mode and if the PLC is ready to execute an operation, a rising edge on this variable starts the execution of the operation described in ManualOperation. Writable by the middleware.

ManualOperation_ABORT_command : BOOL
If the PLC is in manual mode, a rising edge on this variable aborts the current operation. Writable by the middleware.

ManualOperation_Ready_for_new_Operation : BOOL
If this variable is TRUE, an operation can be executed in manual mode. Not writable by the middleware.

2.3.3 Automatic Mode Variables

AutomaticOperations : ARRAY[1..20] OF S_Operation
An array of S_Operation that contains the operations' information to execute a batch of operations. This variable is not addressed directly but only the fields of the S_Operation objects.

AutomaticOperations[i].strID : STRING
The unique id of the i-th operation. Writable by the middleware.

AutomaticOperations[i] : INT
Whether the i-th operation should be run or held. 1 means run, 0 means hold. Writable by the middleware.

AutomaticOperations[i].strResult : STRING
Contains the execution result of the i-th operation. Whenever a batch is executed, the middleware subscribes to these variables to be notified about the execution result of every batch operation. The middleware can write this variable but should not need to do so.

AutomaticOperations[i].bAbortOperation : BOOL
Internally set by the PLC when the i-th executing batch operation should be aborted. The middleware can write this variable but should not need to do so.

AutomaticOperation_START_command : BOOL
If the PLC is in automatic execution mode and the PLC is ready to execute a batch, a rising edge starts the batch execution. Writable by the middleware.

AutomaticOperation_ABORT_command : BOOL
If the PLC is in automatic execution mode, a rising edge aborts the batch execution. The PLC automatically aborts the current operation without any need to provide the index of the currently executing operation. Writable by the middleware.

AutomaticOperation_HOLD_command : BOOL
If the PLC is in automatic execution mode, a rising edge holds the batch execution. The PLC automatically holds the current operation without any need to provide the index of the currently executing operation. Writable by the middleware.

AutomaticOperation_RESTART_command : BOOL
If the PLC is in automatic execution mode and the batch execution is currently held, a rising edge restarts the batch execution. The PLC automatically restarts the current operation without any need to provide the index of the current operation. Writable by the middleware.

AutomaticOperation_Ready_for_new_Operation : BOOL
If this variable is TRUE, a batch can be executed in automatic mode. Not writable by the middleware.

AutomaticOperation_Index : INT
The index of the currently executing operation inside the batch.

2.4 Build Configuration

This section describes how to configure the middleware's build in order to develop it further. The code is organized in an Eclipse project and the middleware is build using Apache Maven [2]. The build is configured in the **pom.xml** file in the Eclipse projects root folder. Therefore, most dependencies can be resolved automatically. Exceptions are the Prosys SDK, the OPC UA Stack, and some artifacts of EMF [3].
The Maven artifacts for the Prosys SDK and the OPC UA Stack are already provided in a local maven repository in the folder local-maven-repo in the middleware's GIT repository. Therefore, as long as the Prosys SDK does not need to be changed to another version, no further configuration is necessary. In case the Prosys SDK needs an update, the files local-repo-opc-ua-sdk.txt and local-repo-opc-ua-stack.txt in the folder /setup in the GIT repository explain how the new artifacts can be generated.
To generate the needed Eclipse artifacts, two things are needed. First, you need to have Maven installed and added to your system's path (so you can directly use the mvn command on the command line everywhere). Second, you need an Eclipse installation that has EMF in version **2.12.0** installed in it. Now, the necessary artifacts can be generated to the local P2 repository by using the following command:

mvn eclipse:make-artifacts -DstripQualifier=true -DeclipseDir=<eclipse-install-path>

<eclipse-install-path> = the relative path to the eclipse installation which has EMF 2.12.0 installed in it.
This allows to use maven to build an executable JAR file of the middleware. This can be done by using Maven on the command line or directly in the Eclipse IDE. The second option is more comfortable if you use the M2Eclipse [4] Maven integration for Eclipse and is explained here. After you have imported the middleware's Eclipse project into your Eclipse workspace and M2Eclipse is installed in your Eclipse installation, simply follow these steps to build the middleware in Eclipse:

1. Right-click on the **pom.xml** file in the project's root folder
2. Select **Run As → 2 Maven Build…**

3. In the **Goals** field write: **clean install**
4. Optional: If you want to skip executing the Unit tests, check the **Skip Tests** checkbox
5. Click **Run**
6. The building process can be observed on the console.
7. The build results (including the JARs) can be found in the project's **target** folder:
    a. **log4j.properties**: a default logging configuration
    b. **xppu-middleware-4.0.0-SNAPSHOT.jar**: The jar archive containing only the built middleware but no dependencies → This is not runnable by itself
    c. **xppu-middleware-4.0.0-SNAPSHOT-jar-with-dependencies.jar**: The jar archive containing the built middleware and all dependencies → Runnable on every Java 8 runtime → Use this jar to use the middleware.

2.5 Further Development

In this section, some possibilities for further development of the middleware are explained.

2.5.1 REST Callbacks

A possibility to improve the comfort of receiving feedback for executed operations are REST callbacks. With the current implementation of the middleware, a user who requested to execute an operation (or batch of operations) on the PLC needs to periodically request the history and check for the corresponding completion entry. This can be avoided with REST callbacks. Thereby, a user provides a callback URL with the execution HTTP request. When the middleware recognizes the operation execution's completion, the callback URL is called with an agreed-upon HTTP request (most likely PUT or POST) [2]. Consequently, the user does not have to worry about finding out when an operation is finished but is notified instead. Additionally, this could reduce the load on the middleware's server because users' do not need to request the history periodically.

2.5.2 Batch Ids

Currently, only execution ids are used to identify the execution of an operation (manual mode) or a batch of operations (automatic mode). For the manual mode, this is sufficient because one execution id is unique for the execution of exactly one operation. In automatic mode however, the execution id is the same for every operation's execution of the batch. Therefore, in automatic mode an execution id does not clearly identify an operation execution.
To solve this problem, a batch id can be introduced for batch executions. In automatic mode, the batch id takes over the execution id's function: It is the same for all history entries that concern one batch. This allows to use a unique execution id for every operation execution inside a batch. Therefore, the tracking of single operations inside a batch is greatly improved.

2.5.3 Webservice Security Features

So far, the REST webservice does not provide any security features like authentication, authorization or encryption. Consequently, everyone that knows the IP address and the possible HTTP requests of the middleware may execute operations on the PLC. Obviously, this entails the risk that unauthorized persons execute operations on the PLC.
To make sure that only authorized persons are able to execute operations, users need to authenticate themselves. This could be done by creating user accounts with a password. A user would then provide the account name and the hashed password for every request. This check could automatically be executed before every request by using the *before filter* of Spark. Spark is the framework used for the webservice and the filter's documentation can be found at [3]. The configuration could be added in the *App* class (see Figure 2).
Another issue is the confidentiality of the users' requests and the corresponding responses. So far, all data is sent unencrypted and, therefore, could be intercepted and read easily. This can be improved by using HTTPS/SSL encrypted connections for the webservice requests and responses. This also can be done with help of the used Spark framework in the *App* class. For documentation on how to use HTTP/SSL with Spark look at [4].

2.5.4 History Persistence

Currently, the history entries are only kept in memory while the middleware is running. This means that once the histories capacity is reached and a new entry is added, the oldest entry is deleted and cannot be accessed anymore. If no one requested the history before that happens, the entry is lost forever. Furthermore, a crash or restart of the middleware wipes the current history.
To avoid these problems, the history could be persisted in a database or file system, e.g. in JSON format. This would allow to restore old entries after a crash or restart. Furthermore, the webservice could be extended to allow access to the older entries.

3 User Documentation

The user documentation describes how to run the middleware as well as how to use the middleware's webservice.

3.1 Middleware Start Up

The middleware is provided as an executable Java Archive (JAR). To run the middleware, Java 8 is required. Furthermore, the log4j.properties file should be provided in the same folder as the middleware jar file (see section 3.4). If Java is part of the system's path, the middleware can be run with the following command on the command line, provided the command line is navigated to the folder that contains the middleware JAR:

java -jar <middleware-jar> -c <configuration-file>

<middleware- jar> is the filename of the middleware jar file.
<configuration-file> is the relative path to the configuration file of the middleware. A configuration file must be provided to run the middleware.

Example:
The middleware jar is named middleware.jar and the configuration file lies in the same folder and is named config.ini. then the command line call is:

java -jar middleware.jar -c config.ini

3.2 Configuration File

Using the configuration file, the middleware can be adapted to the needs of the PLC it is run for. Parameters are configured as key value pairs in the INI format (see [5]). Thereby, the parameters are separated in the two sections INSTANCE and PLC. Table 1 shows the parameters and explains them.

Parameter Key	Value Description
[INSTANCE]	
history_capacity	Number of entries that are stored in the history. Optional, default value: 50
instance_model	Relative path to the isa88 model that describes the production facility. Mandatory.
operation_list	Relative path to the instance model's corresponding operation list. Mandatory.
[PLC]	
flag_switch_delay	When the middleware triggers rising/falling edges, this is the time in milliseconds that is waited between changing the variable's value. Should be greater than the PLC's cycle duration. Optional, default value: 25.
namespace_index	The namespace index of the PLC's OPC UA variables. Should be 4. Mandatory, except if namespace_uri is provided.
namespace_uri	The namespace uri of the PLC's OPC UA variables. Should not be needed as long as the standard namespace with index 4 is provided. Optional, except if no namespace_index is provided
server_uri	The uri of the PLC's OPC UA server. Mandatory.

Table 1: Configuration File Parameters

3.3 Webservice

This section describes all webservice methods and their JSON return data structures. To get the full URL to call a webservice method the method's path must be added to the middleware's IP address. Also, for some methods, one or parameters are needed. There are two kinds of parameters: path parameters and query parameters. Path parameters are part of the method path. Query parameters are key value pairs. These are added after the method path. This results in the following general format of a webservice request URL:

<Middleware-IP>:4567/<method-path>?<query-parameters>

For the following descriptions, the middleware's IP and port will not be stated explicitly again. Descriptions contain the method path, the parameters if available, the HTTP request method (GET or POST) and an explanation what the method does and what it returns. In sub sections 3.3.2, 3.3.3, and 3.3.4 the method path appears as the sub headline of its method.

3.3.1 JSON Return Datatypes

The webservice uses two types of JSON objects to return operations respectively history entries to the requesting user. These JSON objects consist of multiple key value pairs where the key name represents the represented attribute's name.

Operation Instance

An operation instance represents a resolved operation that can be executed on the PLC. Resolved means that the operation's path was successfully resolved against the ISA-88 model defining the middleware's instance by iterating over the path and finding the concrete, executable operation in the model. Table 2 shows the operation instance's attributes.

Key	Value Type	Value Explanation
operationId	string	The unique id identifying the operation
path	string	The operation's unresolved usage path as taken from the operation list
name	string	The operation's name which is not necessarily unique
resolvedPath	string	The operation's path which was resolved against the middleware's ISA-88 model

Table 2: Operation Instance JSON Attributes

History Entry

A history entry represents one event the middleware registered and logged in the history. The history saves the history entries in chronological order. Analyzing the middleware's history entries allows to determine in what state the middleware and the PLC are. Table 3 shows the history entry's attributes.

Key	Value Type	Value Explanation
operationId	string	The unique id identifying the operation
executionId	string	The unique id identifying one operation execution process or one batch execution.
resolvedOperationPath	string	The operation's path which was resolved against the middleware's ISA-88 model
timestamp	string	The point in time when the history entry was created in UTC → Does not adhere to the timezone that the middleware runs in. The ISO-8601 format, typically with millisecond precision, is used. Example: "2017-03-03T15:26:53.904Z"
action	string	The action that caused this history entry. All possible values are explained in Table 4.
resultCode	number	If applicable, the result code gives additional information for the action. A negative result code is not relevant. 100 means an operation was executed successfully. 80 means an operation was started. Other positive result codes indicate an error.

Table 3: History Entry JSON Attributes

Action	Explanation
ABORT	The execution of an operation was canceled (Doesn't matter whether it was running or paused before).
BATCH_COMPLETE	A batch execution was finished.
BATCH_START	A batch execution was started.
COMPLETE	The execution of an operation was finished and a result code was returned.
HOLD	The execution of a running operation was held.
RESET	An emergency stop was resolved. Actions can be executed again.
RESTART	The execution of a held operation was continued.
SET_AUTOMATIC_MODE	The facility's execution mode was set to automatic (batch) execution.
SET_MANUAL_MODE	The facility's execution mode was set to manual (single operation) execution.
START	The execution of an operation was started.
STOP	An emergency stop occurred. No actions can be executed until the emergency stop is resolved.

Table 4: Overview Over All History Actions - Sorted Alphabetically

3.3.2 Instance Methods

This sub section describes all webservice methods related to the facility instance that the middleware was generated for.

instance

Query parameters: None
Path parameters: None
HTTP method: GET
Returns: The ISA-88 EMF model in XMI format

operations

Query parameters: None
Path parameters: None
HTTP method: GET
Returns: A list of all known operations using the operation instance JSON format

operations/<operationId>

Query parameters: None
Path parameters: <operationId> = The operation id whose operation instance is needed
HTTP method: GET
Returns: The operation instance in JSON format for the requested operation id.
If no operation exists for the given id, an error message is returned.

3.3.3 History Methods

This sub section describes all webservice methods related to retrieving the history.

history/complete

Query parameters: None
Path parameters: None
HTTP method: GET
Returns: The complete history as a list of history entries in JSON format

history/<executionid>

Query parameters: None
Path parameters: <executionid> = the execution id of the execution process for which all history entries should be returned
HTTP method: GET
Returns: All history entries who belong to the execution process defined by the given execution id as a list of history entries in JSON format.

history/module/<name>

Query parameters: None
Path parameters: <name> = the name of the module for which all history entries should be returned
HTTP method: GET
Returns: All history entries whose resolved operation path attribute contains the given module name as a list of history entries in JSON format.

history/operationid/<operationid>

Query parameters: None
Path parameters: <operationid> = the operation id for which all history entries should be returned
HTTP method: GET
Returns: All history entries that are related to the operation with the given id as a list of history entries in JSON format

history/timestamp/<timestamp>

Query parameters: None
Path parameters: <timestamp> = the cut off timestamp in ISO-8601 format (this is the same as the format for the *timestamp* key in Table 3)
HTTP method: GET
Returns: All history entries that have the same or a newer timestamp as the given one as a list of history entries in JSON format

3.3.4 Operation Execution Methods

This sub section describes all methods directly related to the execution of operations.

abort

Query parameters: executionid = the execution id of the operation execution process that should be aborted
Path parameters: None
HTTP method: POST
Returns: The history entry resulting from the operation's abort as a history entry in JSON format. Returns an error message if no operation execution can be aborted for the given execution id. This happens if either the execution id does not match the current execution or if an abort is illegal (for example because there is no active operation execution process)

hold

Query parameters: executionid = the execution id of the operation execution process that should be held
Path parameters: None
HTTP method: POST
Returns: The history entry resulting from the operation's hold as a history entry in JSON format. Returns an error message if no operation execution can be held for the given execution id. This happens if either the execution id does not match the current execution or if an abort is illegal because the operation is not currently running.

mode/automatic

Query parameters: None
Path parameters: None
HTTP method: POST
Returns: The history entry resulting from setting the facility to automatic execution mode as a history entry in JSON format.
If the facility already is in automatic mode, HTTP response code 204 with an empty body is returned.
Returns an error message if the execution mode cannot be changed. This happens if the facility is busy with an execution or an emergency stop occurred.

mode/manual

Query parameters: None
Path parameters: None
HTTP method: POST
Returns: The history entry resulting from setting the facility to manual execution mode as a history entry in JSON format.
If the facility already is in manual mode, HTTP response code 204 with an empty body is returned.
Returns an error message if the execution mode cannot be changed. This happens if the facility is busy with an execution or an emergency stop occurred.

restart

Query parameters: executionid = the execution id of the operation execution process that should be restarted
Path parameters: None
HTTP method: POST
Returns: The history entry resulting from the operation's restart as a history entry in JSON format. Returns an error message if no operation execution can be restarted for the given execution id. This happens if either the execution id does not match the current execution or if a restart is illegal (for example because there is no held operation that can be restarted)

start/<operationid>

Query parameters: None
Path parameters: <operationid> = The operation id identifying the operation that should be executed in manual mode
HTTP method: POST
Returns: The history entry resulting from the operation's execution start as a history entry in JSON format. Returns an error message if the operation cannot be started. This happens if one of the following conditions applies: the facility is busy with another execution, an unresolved emergency stop occurred, the operation id doesn't represent a valid operation or the facility is in automatic mode

startbatch

Query parameters: operationids = The list of operations that should be executed as a batch. Possible batch size: 1 to 20. Operation ids are separated by semicolons (;)
Path parameters: None
HTTP method: POST
Returns: The history entry resulting from the batch's execution start as a history entry in JSON format. Returns an error message if the batch cannot be started. This happens if one of the following conditions applies: the facility is busy with another execution, an unresolved emergency stop occurred, no operation id was provided, more than 20 operation ids were provided, or the facility is in manual mode

3.4 Logging

The middleware uses the Log4J [6] framework for logging. This allows to configure the middleware's logging behavior in a file with the name log4j.properties that lies in the same folder as the middleware's JAR archive while the middleware is executed. The advantage of such a log file is that the logging behavior can be changed without changing the middleware's source code.
A sample configuration that logs all events with the level DEBUG or more important to the command line and a log file is available in the middleware's git repository. This should be sufficient for the usage of the middleware. If an individualized logging format is needed, lots of tutorials can be accessed for free on the internet.

4 Conclusion and Future Work

The goal of the IDP was to develop a middleware in Java that allows to dynamically reconfigure an automated production system. Therefore, the middleware communicates with a PLC over a well-defined OPC UA interface (see section 2.3). The interface was developed in collaboration with the implementers of the software running on the PLC.
An instance of the middleware needs to be compatible with the production facility and its PLC for which the middleware is run. This is achieved by generating a runtime instance from two input files. The ISA-88 model describes the production facility. This model is defined as an EMF model. The second file is an operation list that defines which operations in the model are available for execution. The generation is executed during the start-up of the middleware. This allows to use the middleware for any kind of production facility for which the two previously mentioned files are provided.
The middleware allows the user to execute single operations in a manual execution mode and the fully automatic execution of operation batches in an automatic execution mode. Furthermore, single operations and batches can be held, restarted, and aborted according to the ISA-88 standard. Execution results are registered by the middleware and all events are logged in a history. The history can be requested by users and reflects the state of the production facility.
The middleware offers a HTTP REST webservice (see section 3.3) to make its functionality available to users. Users can send HTTP requests to defined URLs based on the IP of the server that the middleware runs on. Offered methods include operation executions, execution mode switches, getting the middleware's history, getting the available operations, and getting the ISA-88 instance model defining the middleware. Every HTTP request results in a HTTP response that contains the created history entry for execution related requests and the requested information for the other requests. In case of an error, the response contains an error message describing the problem.
The middleware was developed with a three-layer architecture (see section 2.1). The separation into service, business logic, and communication layer promotes loose coupling and makes it easier to understand the code. Furthermore, the maintainability and extensibility are increased because different responsibilities inside the middleware are separated. The software design (see section 2.2) also supports loose coupling by using dependencies against interfaces rather than concrete implementations. This allows to replace or modify single components without the need to adjust all other components depending on the changed one.
There are multiple possibilities to further improve the middleware in the future. First, the introduction of REST callbacks allows users to automatically get notified about execution results instead of continuously needing to request the history. Second, the introduction of batch ids allows the proper identification of operation executions inside a batch. Third, the webservice should be extended by security features to keep the users' privacy and prevent unauthorized users from executing operations on the production facility. Finally, the introduction of history persistence allows to access older history entries and avoids information loss when the middleware is restarted, crashes, or no user requests the history before old entries are deleted. More details about these development options are explained in section 2.5.

List of Figures

List of Tables

Acronyms

HTTP *hyper text transfer protocol*
IDE *integrated development environment*
GVL *global variable list*
PLC *programmable logic controller*
REST *representational state transfer*
URI *unique resource identifier*
XPPU *extended Pick and Place Unit*

Bibliography

[1] Prosys, "Prosys OPC UA Java SDK - Prosys OPC," [Online]. Available: https://www.prosysopc.com/products/opc-ua-java-sdk/. [Accessed 22 February 2017].

[2] "Spark Framework - A tiny Java web framework," [Online]. Available: http://sparkjava.com/. [Accessed 05 March 2017].

[3] "Maven – Welcome to Apache Maven," 01 March 2017. [Online]. Available: https://maven.apache.org/. [Accessed 03 March 2017].

[4] „Eclipse Modeling Project,“ [Online]. Available: https://eclipse.org/modeling/emf/. [Zugriff am 03 March 2017].

[5] „M2Eclipse | M2Eclipse,“ [Online]. Available: http://www.eclipse.org/m2e/. [Zugriff am 03 March 2017].

[6] G. Sironi, "REST callbacks - DZone Web Dev," 24 November 2013. [Online]. Available: https://dzone.com/articles/rest-callbacks. [Accessed 01 March 2017].

[7] "Filters - Spark Framework - Documentation," [Online]. Available: http://sparkjava.com/documentation.html#filters. [Accessed 02 March 2017].

[8] "How do I enable SSL/HTTPS? - Spark Framework - Documentation," [Online]. Available: http://sparkjava.com/documentation.html#enable-ssl. [Accessed 02 March 2017].

[9] "INI file - Wikipedia," 16 February 2017. [Online]. Available: https://en.wikipedia.org/wiki/INI_file. [Accessed 02 March 2017].

[10] „Apache log4j 1.2,“ [Online]. Available: https://logging.apache.org/log4j/1.2/. [Zugriff am 02 March 2017].

[11] "Service-oriented architecture - Wikipedia," 8 February 2017. [Online]. Available: https://en.wikipedia.org/wiki/Service-oriented_architecture. [Accessed 23 February 2017].

[12] "Multitier architecture - Wikipedia," 17 January 2017. [Online]. Available: https://en.wikipedia.org/wiki/Multitier_architecture. [Accessed 23 February 2017].

Content of the Data CD

File/Folder	Description
javadoc	Contains the middleware's javadoc code documentation
xPPU40WebInterface	Contents of the middleware's git repository
documentation.docx	The IDP documentation's source document
documentation.pdf	The IDP documentation as a printable PDF file
documentation_extract.pdf	The IDP documentation formatted for easy extraction of single sections
presentation.pdf	The IDP presentation as a PDF file
presentation.pptx	The IDP presentation's source Powerpoint file
xppu-middleware-4.0.0-evaluation.jar	The middleware's executable JAR file using the evaluation version of the Prosys OPC UA Java SDK
xppu-middleware-4.0.0-full.jar	The middleware's executable JAR file using the licensed full version of the Prosys OPC UA Java SDK

Table 5: Content of the Data CD

Eidesstattliche Erklärung

Hiermit erkläre ich, Lucas Köhler, geboren am 29.04.1993 in Aachen-Kornelimünster, dass die vorgelegte Arbeit mit dem Titel „Development of a Java-based service-oriented middleware for the dynamic reconfiguration of production systems‘ PLC-programs“ durch mich selbstständig verfasst wurde.
Ich habe keine anderen als die angegebenen Quellen sowie Hilfsmittel benutzt. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden sind als solche gekennzeichnet.
Die Diplomarbeit wurde nicht bereits in derselben oder einen ähnlichen Fassung an einer anderen Fakultät oder einem anderen Fachbereich im In- und Ausland zur Erlangung eines akademischen Grades eingereicht.
Diese IDP Dokumentation entstand unter wissenschaftlicher und inhaltlicher Anleitung durch meinen Betreuer, Daniel Schütz. Zentrale Ideen und Konzepte wurden gemeinschaftlich mit ihm erarbeitet.

Putzbrunn, den 13. März 2017

