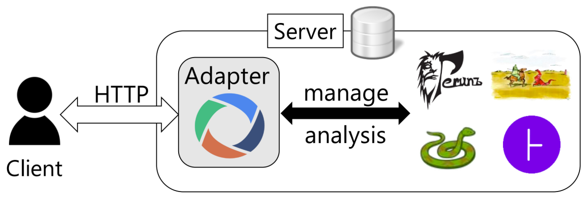
BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2021
http://excel.fit.vutbr.cz

OSLC Adapter for Software Analysis

Ondřej Vašíček, Bc.*



Abstract

The goal of this work is to provide an easy way of adding an OSLC compliant interface to an analysis tool. Such interface allows tools to be easily integrated with other tools or systems; allows them to be used remotely due to its web based nature; and allows them to be easily connected with a database for persistency and queries. In this paper, we present an OSLC adapter which can be used to easily add an OSLC interface to most analysis tools. The adapter was designed and created using Eclipse Lyo and is universal enough to accommodate functionality of most analysis tools using the OSLC Automation domain interface by leveraging their current command-line interfaces. In this paper, we see the universality from the point of view of interoperability and development stage (compile-time and run-time). This work provides a very brief introduction to OSLC and Eclipse Lyo; defines requirements and differences of analysis tools; covers the design of the adapter; and presents a working implementation of the adapter. The most important evaluation indicator is that the current working version of the adapter is already being used in practice in four use cases, i.e. integration of tools ANaConDA, Perun, Spectra (all three developed by VeriFIT); and HiLiTE (Honeywell).

Keywords: OSLC, OSLC Adapter, OSLC Provider, OSLC Server, OSLC Consumer, OSLC Client, OSLC Automation, Eclipse Lyo, tool integration, software analysis and verification, ANaConDA, Facebook Infer, Perun, Valgrind

Supplementary Material: GitLab Repository

*xvasic25@fit.vut.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Most analysis tools have different purpose-specific interfaces which focus on efficiency and providing functionality with less attention to usability, quick learning curve, and integrability into toolchains as parts of the whole SW development lifecycle. Many analysis tools are used through command-line interfaces or as services using their own interface incompatible with other tools which makes integrating them into toolchains complicated. Standardized interfaces make tool integration into toolchains much easier allowing a diverse range of tools and systems to cooperate and provide complex functionality seamlessly to users from the comfort of a single familiar system. OSLC one of the standards which aim to unify tool usage and automation of processes throughout the whole SW lifecycle. OSLC support specifically is valued in international research projects like AuFoVer [1] and Arrowhead [2].

[1]https://www.vutbr.cz/en/rad/projects/detail/29833
[2]https://www.fit.vut.cz/research/project/1299/.en

The way to add an OSLC interface to a tool which is natively not a web application is by creating an adapter. This is the case for all the analysis tools considered in this work. However, adding OSLC support to a tool can be a demanding process with a steep learning curve[3] due to prerequisite knowledge required to understand it. An appropriate OSLC domain needs to be picked for the particular integration scenario, and then the tool's interface needs to be translated to resources defined by the domain. In order to make OSLC wide spread, the process of adopting it needs to be made as easy as possible.

OSLC is already supported by a number of tools; a list of them was published in [1]. Eclipse Lyo [2] is an open project focused on making the process of adopting OSLC easier by providing tools to accommodate it. It consists mainly of the OSLC4J SDK, Lyo Designer and Code Generator [3], and many other components, such as Lyo Store, reference implementations, or a test suite for specification compliance. The tools provided are only for Java with SDK alternatives for other languages, such as .NET (OSLC4Net) and javascript (OSLC4JS), being developed. The Eclipse Lyo project can take care of a large part of an OSLC adopter's work especially thanks to Lyo Designer which can generate an OSLC compliant web application based on models of the domain and required capabilities. However, creating an OSLC adapter still requires knowledge of the available tools and a good understanding of OSLC.

This work focuses on adding OSLC interfaces to analysis tools, such as ANaConDA, Facebook Infer, or Valgrind. These tools are similar in their usage through a command-line which can be seen as a standard interface for this domain of tools. This work presents an implemented OSLC adapter that adapts the command-line interface of analysis tools to the OSLC Automation domain. Such an adapter should be usable with any analysis tool making it an universal adapter for analysis tools.

The OSLC Universal Analysis Adapter is configurable to adapt to any command-line analysis tool. The tools tested so far include: ANaConDA [4], Facebook Infer [5], Valgrind [6], Perun [7], grep, HiLiTE, and Spectra [8]. The current working version of the adapter is already being used in practice both by researchers at our university (FIT BUT - VeriFIT) and by our industry partners (Honeywell).

[3]A good place to start: https://oslc.github.io/developing-oslc-applications

[4]https://www.fit.vutbr.cz/research/groups/verifit/tools/anaconda/

[5]https://fbinfer.com

[6]https://valgrind.org

[7]https://github.com/tfiedor/perun

[8]https://www.fit.vutbr.cz/research/groups/verifit/tools/testos-spectra

2. Background

This section briefly introduces OSLC, the OSLC Automation domain, and Eclipse Lyo. A similar introduction in more detail can be found in [4] or [5].

2.1 Open Services for Lifecycle Collaboration

OSLC [6] is an OASIS Open Project focused on interoperability of tools throughout the whole software development lifecycle. OSLC uses self-describing *RESTful APIs*, serialized data representation, and the concept of Linked Data. Each artifact in OSLC is a HTTP *resource* identified by a URI that can be interacted with using *CRUD* HTTP requests (Create, Read, Update, Delete) or used to link multiple resources together via *relations*. Resources are produced and managed by OSLC servers and consumed by OSLC clients [9].

OSLC consists of a number of specifications defining resources and resource shapes for different application domains, such as *Quality Management*, *Architecture Management*, *Change Management*, or *Requirements Management*. All the application domain specifications are based on top of the *Core domain* [7]. Figure 1 shows the layered architecture of OSLC.

2.2 OSLC Automation Domain

The OSLC Automation domain [8] focuses on integration scenarios involving tools like analysis tools, build tools, or deployment tools. The basic concept of automation is a tool that is capable of performing a certain action. A user of that tool then wants to request an execution of that action and subsequently get the results of the execution. Figure 2 shows resources defined by the domain.

Automation Plans represent the units of automation available for execution. Their main role is to define input parameters for Automation Requests using *Parameter Definitions* with properties such as *name*, *value type, allowed values*, *occurrence*, or *default values*.

Automation Requests are what OSLC clients create to request execution of an Automation Plan. When creating a request, the OSLC client needs to choose an Automation Plan, and provide input parameters for the Automation Request based on parameter definitions of the Automation Plan. The client then polls the created Automation Request to check its state which indicates whether its execution has finished or not. The most

[9]Formerly called providers and consumers

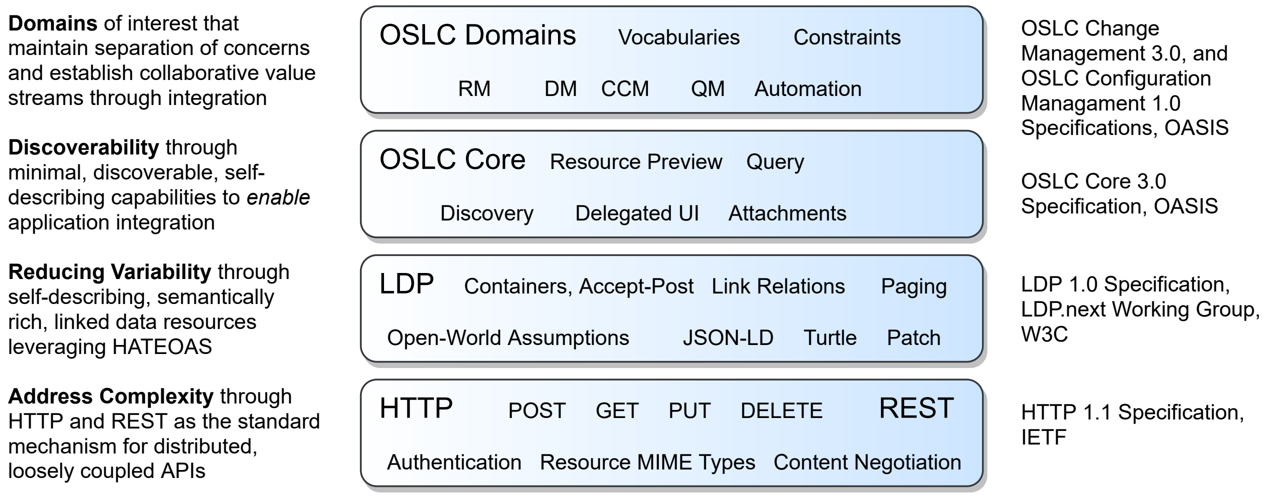

Domains of interest that maintain separation of concerns and establish collaborative value streams through integration
Discoverability through minimal, discoverable, self-describing capabilities to *enable* application integration
Reducing Variability through self-describing, semantically rich, linked data resources leveraging HATEOAS
Address Complexity through HTTP and REST as the standard mechanism for distributed, loosely coupled APIs
OSLC Domains
Vocabularies
Constraints
RM
DM
CCM
QM
Automation
OSLC Core
Resource Preview
Query
Discovery
Delegated UI
Attachments
LDP
Containers, Accept-Post
Link Relations
Paging
Open-World Assumptions
JSON-LD
Turtle
Patch
HTTP
POST
GET
PUT
DELETE
REST
Authentication
Resource MIME Types
Content Negotiation
OSLC Change Management 3.0, and OSLC Configuration Managament 1.0 Specifications, OASIS
OSLC Core 3.0 Specification, OASIS
LDP 1.0 Specification, LDP.next Working Group, W3C
HTTP 1.1 Specification, IETF

Figure 1. OSLC layered architecture (source: [7], remade)

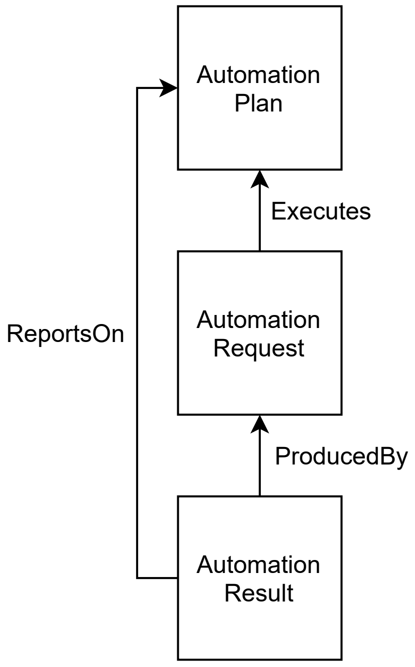


Figure 2. OSLC Automation resources [8]

important properties of a request are *input parameters*, *state*, *desired state*, and a link to the *executed Automation Plan*. Input parameters are represented as *ParameterInstance* resources with properties such as *name* and *value*.

Automation Results are produced by the OSLC server based on an Automation Request. The most important part of a result are its *contribution* properties which reference various artifacts that were produced by the Automation Request execution with properties such as *title*, *type*, and *value* (the contribution itself). Other important properties of an Automation Result are *verdict*, *state*, and a link to the *associated Automation Request*. An Automation Result contains the same input parameters as the originating Automation Request, and in addition contains output parameters which represent parameters that changed or were added during execution.

2.3 Eclipse Lyo

Eclipse Lyo [2] is an eclipse project focused on making the process of adopting OSLC easier. It consists of the OSLC4J SDK, Lyo Designer, and many other components, such as Lyo Store, reference implementations, or a test suite for specification compliance.

OSLC4J is a Java toolkit for developing OSLC servers and clients. It contains Java object annotations for OSLC attributes and UI previews, support for service provider and resource shape documents, libraries for developing servers and clients, sample applications, and a test suite.

Lyo Designer is an Eclipse plugin that consists of two parts: Lyo Toolchain Designer and Lyo Code Generator. *Lyo Toolchain Designer* is a graphical modeling tool capable of modeling OSLC domains, OSLC vocabularies, OSLC toolchains, and adapter interfaces. *Lyo Code Generator* can generate code using OSLC4J based on the models created with the designer. The generator generates annotated classes for the modeled domain and all the required logic to run a working Maven Web application for the modeled adapters.

Lyo Domains is a repository containing models and generated classes of all OSLC application domains.

Lyo Store is a library for storing OSLC resources in a SPARQL triplestore offering persistency and query functionality based on the OSLC Query Syntax.

3. Analysis Tools Requirements

In order to design an universal adapter, usage requirements of different analysis tools need to be examined.

Tools used as representatives for the requirements study and testing include, ANaConDA (dynamic analysis), Valgrind (dynamic analysis), Perun (dynamic analysis and Git integration), Facebook Infer (static analysis), grep (simplest form of static analysis, and UNIX utility representative), and HiLiTE (test case generation). These tools were used to define what functionality needs to be provided by the adapter. Figure 3 shows the defined requirements and a brief summary of how the OSLC Universal Analysis Adapter accommodates them. These requirements translate either to design decisions, features provided by the adapter, or directly to the OSLC Automation domain resources and actions. For more in-depth descriptions refer to [4][10].

4. Design and Implementation

The adapter produced in this work was designed to meet analysis tool requirements from Section 3 using the OSLC Automation domain, Lyo Designer and Code Generator, Lyo Domains and Lyo Store. Base code was generated using Lyo Designer as a Java Maven web application, packaged with an Apache Fuseki SPARQL triplestore [12], and is ready to run on both Linux and Windows.

The OSLC Automation domain was chosen for the interface because its resources directly translate actions performed and artifacts used when running an analysis tool through the command-line, see Figure 4.

4.1 Architecture — Analysis and Compilation

Due to compilation and analysis being two different integration scenarios, it was decided to split the designed adapter into two sub-adapters - two OSLC servers with separate interfaces forming a toolchain. Having two separate interfaces is useful because it avoids mixing two unrelated usage scenarios in one place, and having a standalone compilation server allows it to potentially be replaced by another compilation and deployment system or to be used on its own in a separate use case. Figure 5 shows interfaces of the two sub-adapters and their interaction.

The two sub-adapter's need to run on the same server so that the Compilation adapter can be used to create SUT's and the Analysis adapter can then access them directly to execute analysis.

The domain model of the adapter was created using Eclipse Lyo. All the Automation domain resources have been imported from Lyo Domains with small modifications to fit our use case. An entirely new resource *fit:SUT* was crated in our own domain namespace *VeriFit Universal Analysis* to represent SUT resources created by the Compilation adapter and used by the Analysis adapter.

4.2 Configuration an Execution

The adapter allows users to define their own Automation Plans using configuration files. Each Automation Plan represents an analysis tool available to be executed on the server and its input parameters. Input parameters are divided into command-line arguments for the tool and functional parameters for the adapter which control features such as execution timeout or matching of execution outputs. Command-line arguments passed as input parameters on Automation Request creation by a client are then put together based on their positions and prefixed by the tool launch command to form a string to be executed in the native shell of the server by the adapter.

Files produced by the execution are then added as Automation Result Contributions along with contributions created by the adapter, such as the total execution time, standard outputs, or status messages. Users can also define their own plug-in filters to process Contributions produced by an Automation Request which allows them to modify them in any way to control what is stored in the database as results.

5. Evaluation and Experiments

The adapter's functionality was verified by an automated test suite using Postman [13] that contains over 700 HTTP requests. The test suite contains system tests that cover the core functionality and error handling of the adapter, features required to accommodate requirements defined in Section 3, and basic usage scenarios of all the tested analysis tools.

Further manual experiments were performed by executing analysis using ANaConDA, Perun, Valgrind, Grep, and Facebook-Infer to make sure these tools can be used through the adapter.

More importantly, the adapter is currently deployed in practice with four different analysis tools. Honeywell uses the adapter for automated test case generation and other analyses using their tool, HiLiTE, through a web client and as part of their requirements verification tool. And the VeriFIT research group from BUT FIT uses the adapter to provide an analysis server running ANaConDA, Spectra, Perun, and more in the future. Furthermore, an Eclipse plugin for executing

[10]Or my upcoming diploma thesis named *OSLC Adapter for Software Analysis*

[11]System-Under-Test

[12]`https://jena.apache.org/documentation/fuseki2`

[13]`https://www.postman.com`

1. **General Requirements for Analysis Execution**
   (a) **Initial Setup** - Installing the analysis tool and configuring the adapter
   (b) **Pre-Analysis Execution**
      i. **Transfer the SUT [11] to the server** - Creating an SUT resource
      ii. **Compile the SUT** - Executing a build command on the SUT resource
   (c) **On Analysis Initiation**
      i. **Configure the analysis tool** - Configuration files or directories as input parameters of Automation Requests
      ii. **Discover the analysis tool's parameters** - Listed in an Automation Plan defined for the tool
      iii. **Specify analysis input parameters** - Automation Request input parameters
      iv. **Execute analysis tool** - Directly executing the tool using its launch command
   (d) **During Analysis Execution**
      i. **Monitor analysis status** - State property of the Automation Request and Result
      ii. **Cancel execution** - Updating the desired state of the Automation Request
   (e) **Post-Analysis Execution**
      i. **Get analysis outputs** - Add files modified during analysis as Automation Result contributions
      ii. **Information about execution run** - Automation Result contributions with execution time etc.
      iii. **Persist analysis outputs** - SPARQL triplestore database with optional persistency
      iv. **Examine analysis outputs** - Contributions can be downloaded and queried
      v. **Run follow up analysis** - SUT resources as workspaces (modifications stay after analysis)
      vi. **Clean up** - Restoring the SUT to its initial state or creating a fresh one
2. **Tool Type Specific Requirements**
   (a) **Dynamic Analysis** (ANaConDA, Valgrind)
      i. **Build the SUT** - Basic function of the Compilation adapter
      ii. **Launch the SUT** - Fetch the launch command from the SUT resource
   (b) **Static Analysis** (Facebook Infer, grep)
      i. **Knowledge of the SUT build command** - Fetch the build command from the SUT resource
      ii. **Building the SUT might not be required** - Compilation toggle as an input parameter
   (c) **Stateful Analysis or Combination of Tools** (Perun)
      i. **Keeping SUT context for multiple analysis executions** - SUTs as workspaces
      ii. **Modifying files created by analysis execution** - Update capability for Contributions
   (d) **Model Based / Test Case Generation** (HiLiTE)
      i. **Generic artifact as the analysis input** - SUTs with optional properties and compilation

Figure 3. A brief summary of analysis tool usage requirements (in bold) and an outline of how they are accommodated by the adapter.

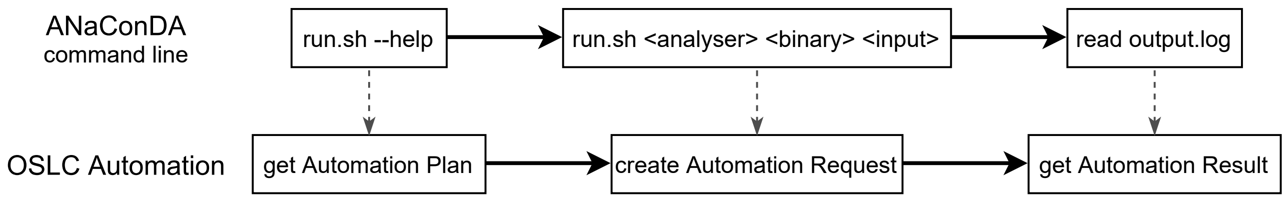


Figure 4. Mapping actions performed when using an analysis tool to OSLC Automation domain resources.

analysis from the IDE using the adapter is currently being developed by a VeriFIT member.

The adapter in its current state covers almost all of the requirements described in Section 3. Due to allowing clients to execute almost any command on the server based on their custom Automation Plans with custom input parameters, the adapter should be able to provide functionality of any unix style command-line

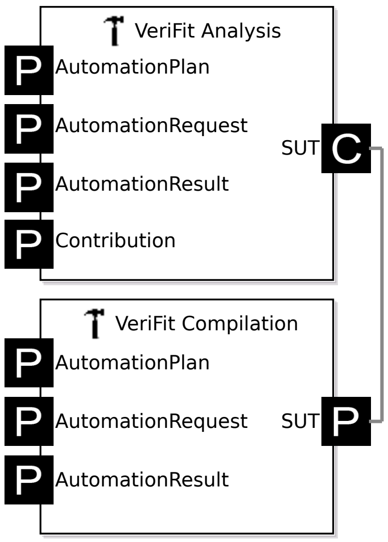


Figure 5. Toolchain diagram of the Universal Analysis Adapter showing its two sub-adapters. Made with Eclipse Lyo

utility. There is, however, currently no way of cleaning an SUT after an analysis has been executed on it other then creating a new SUT resource.

5.1 Case Study — ANaConDA

To demonstrate the usefulness and functionality of the adapter we present a case study of running ANaConDA through the adapter. Unfortunately, there is not enough space to include examples of actual RDF resources used to communicate with the adapter or to configure it.

Command-line interface of ANaConDA for executing anlaysis looks like this: `run.sh [optParams] <analyser> <SUT> <inputs>`. Mandatory parameters are used to select an analyser to use, to launch an SUT, and to pass input parameters to the SUT respectively. A notable optional parameter for our case is `--config <dir>` which instructs ANaConDA to use a custom configuration directory.

To run analysis using ANaConDA through the adapter, we need to configure it by creating an Automation Plan and a properties file. The properties file will most importantly specify a path to the ANaConDA `run.sh` script. Most important properties of the Automation Plan will be an `identifier` to form its unique URI, and parameter definitions to define ANaConDA's interface. The `analyser` parameter will be represented by a parameter definition with the same name, allowed values matching available analysers, and command-line position 2. The `SUT` parameter will be represented using a `launchSUT` parameter definition with a default value of `true` which is a special parameter which instructs the adapter to insert the SUT launch command at command-line position 3. The `inputs` parameter will have no restrictions and command-line position 4. All mandatory parameters will have occurrence set to `exactly-one`. And finally, the `--config <dir>` parameter will have command-line position 1, value prefix "`--config `", and occurrence `zero-or-one`. After the adapter loads the new configuration, it will add a number of common input parameters to the new Automation Plan which include a link to the SUT to be analysed and all the adapter features like creating configuration files or specifying a timeout.

Then, before executing analysis, an SUT needs to be created using the Compilation sub-adapter. This is done by sending a POST request containing an Automation Request to a creation factory URI in the Compilation sub-adapter. Input parameters of the Automation Request will be `sourceGit` with a Git repository as its value; `launchCommand` specifying the SUT launch command property; and `buildCommand` specifying the SUT build command. A default value of `true` will be used for the `compile` parameter which is an optional parameter used to toggle compilation. The adapter will respond with a created version of the Automation Request containing a link to an Automation Result. Once SUT creation is finished, signified by the `state` property of the Automation Result, the Automation Result will contain a link to the newly created SUT resource along with compilation outputs and other properties.

The SUT resource URI can then be used as one of the input parameters of requesting analysis in the Analysis sub-adapter. Analysis can be requested by sending a POST request containing an Automation Request to a creation factory URI in the Analysis sub-adapter. Input parameters submitted in the request will contain a mix of ANaConDA interface parameters and common adapter parameters. ANaConDA input parameters will match the configuration created earlier to select an analyser (e.g. atomrace, a data race[14] analyser), specify inputs for the analysed SUT, and to specify a configuration directory. These will form a string to execute together with the ANaConDA launch script and the SUT launch command. The common adapter parameters will include a `confDir` parameter whose value holds a base64 encoded zip archive of a configuration directory filled with ANaConDA configuration files; and a `outputFilter` parameter which selects an output filter to use when processing Contributions

[14]data race - two or more concurrent accesses to the same memory location with at least one being a write

produced by the analysis (defined at the end of the case study). Before analysis is executed by the adapter, the configuration directory will be placed into the SUT directory to be usable by ANaConDA. Result of the analysis can be retrieved as an Automation Result resource in the same way as for SUT creation. The executed string will be: `run.sh --config *dir* atomrace *launchSUT* *inputs*`. The Automation Result will contain Contribution resources which will contain standard outputs of the analysis, its return code, execution time, adapter status messages, and others.

The output filter for ANaConDA was defined as a plug-in for the adapter consisting of a class implementing a `filter` method. The filter searches standard output for any data race reports by ANaConDA and creates a boolean Contribution resource which says whether a data race was found or not. This new Contribution can then be queried by clients to see the actual outcome of the analysis.

All resources created during the whole process described above can be persistently stored and queried if desired. Multiple Automation Plans can be defined for integrating multiple tools into the adapter. The main advantage of running analysis in this way is that the communication process for all analysis tools uses standard OSLC Automation with standard syntax and semantics of actions.

6. Conclusions

This paper provided a basic introduction to OSLC and its Automation domain, and listed useful development tools for OSLC available in Eclipse Lyo. An OSLC Universal Analysis Adapter was designed an implemented using Eclipse Lyo to allow most analysis tools with a command-line interface to be extended with an OSLC Automation interface.

The adapter covers almost all of the defined analysis tool requirements and is already being used in three different use cases both by researches and by our industry partners. It allows tools to be used through OSLC and remotely as servers with little effort and no manual coding required both on Linux and Windows.

There is still quite a bit of work on finishing the adapter and adding useful features in the future. Proper thought needs to go into security concerns of letting clients execute any SUT and any commands on the analysis server should the adapter be used in an untrusted environment. And there are plans for creating a coordinator adapter which could aggregate multiple servers running different analysis tools using instances of the Universal Analysis Adapter to provide load balancing, fault tolerance, and access to multiple systems in one place.

Acknowledgements

I would like to thank my supervisor Aleš Smrčka, Ing., Ph.D, my consultants Jan Fiedor, Ing., Ph.D., Prof. Tomáš Vojnar, Ing., Ph.D.; Bohuslav Křena, Ing., Ph.D, and Mgr. Tomáš Kratochvíla.

This work was supported by the AuFoVer project[15] and the Arrowhead project[16].

References

[1] Jad El-khoury. An analysis of the oasis oslc integration standard, for a cross-disciplinary integrated development environment : Analysis of market penetration, performance and prospects. Technical Report 978-91-7873-525-9, KTH, Mechatronics, 2020. QC 20200430.

[2] Eclipse lyo. https://www.eclipse.org/lyo/. Accessed: 2021-03-27.

[3] Jad El-khoury. Lyo code generator: A model-based code generator for the development of oslc-compliant tool interfaces. *SoftwareX*, 2016.

[4] Ondřej Vašíček. Oslc adapter for software analysis. FIT VUT v Brně, 2021. Term Project, Brno.

[5] Ondřej Vašíček. Oslc adapter for anaconda framework. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Aleš Smrčka, Ph.D., 2019. Bachelor's thesis.

[6] Open services for lifecycle collaboration. https://open-services.net/. Accessed: 2021-03-27.

[7] Oslc core version 3.0. part 1: Overview. http://docs.oasis-open.org/oslc-core/oslc-core/v3.0/csprd03/part1-overview/oslc-core-v3.0-csprd03-part1-overview.html. Edited by Jim Amsden. 31 May 2018. OASIS Committee Specification Draft 03 / Public Review Draft 03. Accessed: 2021-03-27.

[8] Oslc automation version 2.1 part 1: Specification. https://rawgit.com/oasis-tcs/oslc-domains/master/auto/automation-spec.html. Edited by Fabio Ribeiro. 03 March 2019. OASIS Working Draft 01. Accessed: 2021-03-27.

[15] https://www.vutbr.cz/en/rad/projects/detail/29833

[16] https://www.fit.vut.cz/research/project/1299/.en