BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

OSLC ADAPTER FOR ANACONDA FRAMEWORK

ADAPTÉR OSLC PRO FRAMEWORK ANACONDA

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **ONDŘEJ VAŠÍČEK**
AUTOR PRÁCE

SUPERVISOR **Ing. ALEŠ SMRČKA, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Vašíček Ondřej**
Program: Informační technologie
Název: **Adaptér OSLC pro framework ANaConDA**
OSLC Adapter for ANaConDA Framework
Kategorie: Analýza a testování softwaru

Zadání:

1. Seznamte se se standardem OSLC. Nastudujte tvorbu adaptérů OSLC pro nástroje pro testovací účely. Nastudujte framework ANaConDA pro dynamickou analýzu paralelních programů.
2. Analyzujte požadavky na automatizované spouštění dynamické analýzy prostřednictvím frameworku ANaConDA. Navrhněte architekturu systému poskytující službu automatické dynamické analýzy programů.
3. Implementujte službu pro dynamickou analýzu programů využívající framework ANaConDA. Služba bude poskytována prostřednictvím OSLC adaptéru.
4. Ověřte funkcionalitu služby pomocí automatických integračních testů.

Literatura:

- Fiedor Jan, Mužikovská Monika, Smrčka Aleš, Vašíček Ondřej a Vojnar Tomáš. Advances in the ANaConDA Framework for Dynamic Analysis and Testing of Concurrent C/C++ Programs. 2018. In: *Proceedings of ISSTA'18.* s. 356-359. doi: 10.1145/3213846.3229505
- OSLC Group. Tutorial: Integrating products with OSLC. Dostupné na URL: https://open-services.net/resources/tutorials/integrating-products-with-oslc/
- OSLC Group. Quality Management 2.0. Dostupné na URL: http://open-services.net/specifications/quality-management-2.0/

Pro udělení zápočtu za první semestr je požadováno:

- První dva body zadání.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Smrčka Aleš, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 1. listopadu 2018

Abstract

This work is a proof of concept for adding an OSLC interface to academic software tools. The process of adding OSLC support to a tool is demonstrated by creating an OSLC adapter for the ANaConDA framework using Eclipse Lyo. The goal of this work is to allow ANaConDA to integrate with other tools. An introduction to ANaConDA, OSLC, and Eclipse Lyo is provided along with an overview of the related basic concepts. Then, the design and implementation of the OSLC adapter for ANaConDA is presented. As a result of this work, the process of creating an OSLC adapter is evaluated, and a working implemented OSLC adapter for ANaConDA is presented. Finally, this work features an overview of my past ANaConDA related work which includes implementation of two data race detectors, FastTrack and Eraser.

Abstrakt

Předmětem této práce je ověření konceptu rozšiřování akademických programových nástrojů o OSLC rozhraní. Proces rozšíření nástroje o OSLC rozhraní je demonstrován vytvořením OSLC adaptéru pro prostředí ANaConDA za použití Eclipse Lyo. Cílem této práce je umožnit integraci prostředí ANaConDA s jinými programovými nástroji. Práce poskytuje základní úvod k prostředí ANaConDA, OSLC a Eclipse Lyo spolu s přehledem souvisejících konceptů. Dále je popsán návrh a implementace vytvořeného OSLC adaptéru. Výsledkem této práce je zhodnocení procesu vytváření OSLC adaptéru a představení implementovaného OSLC adaptéru pro prostředí ANaConDA. Závěrem je zmíněna má předchozí práce na rozšiřování prostředí ANaConDA ve formě implementace dvou detektorů souběhů, FastTrack a Eraser.

Keywords

OSLC, OSLC Adapter, OSLC Provider, OSLC Automation, Eclipse Lyo, tool integration, ANaConDA, dynamic analysis, concurrency

Klíčová slova

OSLC, OSLC Adapter, OSLC Provider, OSLC Automation, Eclipse Lyo, integrace nástrojů, ANaConDA, dynamická analýza, souběžnost

Reference

VAŠÍČEK, Ondřej. *OSLC Adapter for ANaConDA Framework*. Brno, 2019. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Aleš Smrčka, Ph.D.

Rozšířený abstrakt

Předmětem této práce je ověření konceptu rozšiřování akademických programových nástrojů o OSLC rozhraní. Tahle iniciativa spadá pod výzkumnou skupinu VeriFIT a její účast v projektech AQUAS a AUFOVER. Proces rozšíření nástroje o OSLC rozhraní je demonstrován vytvořením OSLC adaptéru pro prostředí ANaConDA za použití Eclipse Lyo. Hlavním cílem této práce je umožnit integraci prostředí ANaConDA s jinými programovými nástroji.

OSLC (*Open Services for Lifecycle Collaboration*) je otevřený projekt organizace OASIS, který definuje standard pro integrační rozhraní vývojových nástrojů. OSLC používá webové technologie a architekturu klient-server. Server spravuje zdroje (angl. resources), se kterými pracuje daný nástroj, a poskytuje nad nimi klientům sadu operací. Standard definuje RESTful rozhraní pro jednotlivé integrační scénáře v životním cyklu vývoje software a způsob komunikace pomocí protokolu HTTP.

Prostředí ANaConDA (*Adaptable Native-code Concurrency-focused Dynamic Analysis*), postavené nad nástrojem PIN, poskytuje podporu pro tvorbu dynamických analyzátorů a následnou dynamickou analýzu vícevláknových programů napsaných v jazyce C/C++ na binární úrovni. ANaConDA obsahuje několik předpřipravených analyzátorů a navíc podporuje techniku vkládání šumu, která zvyšuje šanci nalezení chyb.

Eclipse Lyo je sada nástrojů a knihoven, vytvořená autory OSLC, určená pro usnadnění tvorby nástrojů podporujících OSLC. Obsahuje především *Lyo Designer*, rozšíření prostředí Eclipse pro modelování OSLC rozhraní umožnující generovat základ kódu podle modelovaných zdrojů a operací nad nimi, a knihovnu *Lyo Store*, která umožňuje snadné propojení adaptéru s databází pro implementaci perzistence.

OSLC adaptér byl navržen jako server spravující rozhraní *OSLC Automation*, které je určené pro nástroje provádějící překlad nebo analýzu. Rozhraní je složené ze tří hlavních zdrojů - *Automation Plan, Automation Request, Automation Result.* Automation Plan představuje jednotky automatizace, které server poskytuje. Automation Request představuje žádosti, které vytváří klient pro exekuci některé z jednotek automatizace. A Automation Result reprezentuje výsledek jednotky automatizace provedené serverem. Pro návrh adaptéru a generování základu kódu byl použit Lyo Designer a pro implementaci perzistence byla použita knihovna Lyo Store.

Výsledkem této práce je zhodnocení procesu vytváření OSLC adaptéru a prezentace implementovaného OSLC adaptéru pro prostředí ANaConDA. Adaptér umožňuje používat prostředí ANaConDA jako analyzační nástroj skrze standardizované REST rozhraní. Adaptér navíc poskytuje perzistentní uchovávání zdrojů v SPARQL databázi a možnost se na tyto zdroje dotazovat pomocí syntaxe definované v *OSLC Query.* Splnění požadavků standardu bylo pro adaptér ověřeno pomocí OSLC testovací sady, která je součástí Eclipse Lyo, a jeho funkčnost byla ověřena testovacím nasazením v Brněnské pobočce firmy Honeywell, ze kterého byla získána velmi pozitivní zpětná vazba.

Práce na adaptéru bude dále pokračovat v rámci spolupráce výzkumné skupiny VeriFIT a firmy Honeywell v českém národním projektu AUFOVER. Mezi možná vylepšení adaptéru patří zajištění komplexnějšího kompilačního systému, který aktuálně umožňuje analyzovat pouze programy s jedním zdrojovým souborem, nebo přidaní funkcionality pro úplnou konfiguraci prostředí ANaConDA. Významným předmětem další práce je pak konverze adaptéru na generický adaptér, který by umožňoval pracovat s více analyzačními nástroji.

V závěru této práce je zmíněna má předchozí účast na rozšiřování prostředí ANaConDA ve formě implementace dvou detektorů souběhů, FastTrack a Eraser.

OSLC Adapter for ANaConDA Framework

Declaration

I declare that I have prepared this Bachelor's thesis independently, under the supervision of Ing. Aleš Smrčka, Ph.D.
Ing. Jan Fiedor, Ph.D. provided me with further information.
I listed all of the literary sources and publications that I have used.

.....
Ondřej Vašíček
July 26, 2019

Acknowledgements

I would like to thank my supervisors from the University of Malta, Christian Colombo and Mark J. Vella, for their assistance with making this work in line with the local expectations, and for their review and feedback of this work's drafts. And another big thank you goes to the University of Malta and ERASMUS+ for giving me the opportunity to spend time in Malta exploring the country and meeting new people.

Then, I would like to thank my supervisors from my home university, Aleš Smrčka and Jan Fiedor, for their assistance and guidance in selecting the topic of this work and producing the result. A special thanks goes to Jan Fiedor and Honeywell employees for deploying the ANaConDA OSLC Adapter in Honeywell and experimenting with it.

Also, I would like to thank to the creators of Eclipse Lyo, especially Jad El-Khoury, for their assistance with issues encountered during this work.

Finally, I would like to thank my girlfriend for proof reading this work and for her encouragement in finishing this work.

Contents

Chapter 1

Introduction

This chapter comprehensively presents and explains the core point of this work, the objective, how it was achieved, and what the results are. As the last point, a brief description of this work's structure and chapter contents is presented.

1.1 Motivation and Objectives

This work was initiated as a contribution to the AQUAS project [1] and the AUFOVER project [5].

OSLC seems to be becoming the preferred way of integrating development lifecycle tools. It is being adopted or experimented with by both academic and industry developers. Adding an OSLC interface to a tool means making it more accessible for potential users. This is why we want to explore the possibilities of adding an OSLC interface to a tool. If OSLC turns out to be a viable integration option, it can be added to all the tools developed at our university.

This work's objective is to demonstrate that OSLC support can be added to academic software tools. Our evaluation focuses on the following points:

1. Ability of the new OSLC interface to provide the functionality of the interfaced tool. We require the OSLC interface to be able to provide all the core functionality.

2. Difficulty of the process of adding an OSLC interface to a tool. We consider how demanding it is to get familiar with OSLC, and how complex it is to design and implement an OSLC adapter. The process should be as easy as possible. In an ideal case, we want most of the creation process to be accommodated by libraries or SDKs so that we can focus on aspects of the adapter that are specific to our tool.

3. Functionality of the new OSLC adapter which was implemented in this work. We focus on verifying its compliance with OSLC and on testing its usability in a real use case scenario. The implementation is expected to be fully compliant with OSLC, and we hope to receive positive feedback from test usage experiments.

1.2 Solution

We create an OSLC adapter for the ANaConDA framework. ANaConDA is a dynamic analysis tool that featured in academic papers and was also used in some international

projects. ANaConDA recently started to get recognized by its first potential users, some of whom are interested in integrating ANaConDA using OSLC.

OSLC adapter for a non Web based tool has to be a standalone Web application, assuming we don't want to convert our tool into a Web application. Creating an entire OSLC adapter from scratch is a complex task which requires the creator to have good knowledge in many areas. To make OSLC easier to adopt, the creators of OSLC have made Eclipse Lyo, a project which provides libraries, SDKs, and general resources for adopting OSLC. Lyo Designer, OSLC model designer and Java code generator, was used to model OSLC domains and the adapter's interface, and to generate the core adapter's code.

ANaConDA is currently used through a command line interface by requesting analysis of a program and then receiving the analysis output. This kind of usage is best suited for the OSLC Automation domain because its core elements are requests and their corresponding results. To provide persistent storage and allow more complex resource query capability, we decided to use a SPARQL triplestore database server.

1.3 Result

A working ANaConDA OSLC Adapter was created as a Java web application packaged with a standalone SPARQL triplestore, Apache Jena Fuseki. Both web applications are meant to be deployed as WAR files, and were tested using a Jetty container.

The adapter provides the core functionality of ANaConDA through a RESTful interface based on the OSLC Automation domain. Using a REST client, such as Postman, it is possible to send HTTP requests to execute analysis of a desired program, fetch the results, update the results, and query for resources. A development test suite was created in Postman to verify the adapter's functionality, and a Postman request collection with examples of usage is provided. The adapter was tested using an OSLC compliance test suite, and deployed on a server in Honeywell [11] in Brno for its employees to experiment with. Honeywell gave very positive feedback on the adapter, and recently the developers of *art2kitekt* [4] started to experiment with the adapter as well.

OSLC turned out to be ideally suited for this scenario, because the OSLC Automation domain exactly matches the usage of a tool with a command line interface like ANaConDA. The process of creating an OSLC adapter is well supported by various SDKs and libraries, which provide a lot of required functionality. However, most of the provided tools are still under development meaning they have limited functionality and may contain bugs. Also the documentation for the tools provided and for using OSLC in general is somewhat hard to find and connect together.

1.4 Document Structure

Chapter 2 introduces the background and basic concepts of ANaConDA and OSLC. Chapter 3 introduces the ANaConDA framework, how it works, and what its capabilities are. Chapter 4 introduces the core concepts of OSLC, the OSLC domains used in this work, and Eclipse Lyo. Chapter 5 presents the model and architecture of the ANaConDA OSLC Adapter. Chapter 6 covers the important parts of the adapter's implementation. Chapter B presents my prior work related to ANaConDA. Chapter 7 evaluates the results of this work. Finally, Appendix A explains how to use the adapter.

Chapter 2

Background

This chapter briefly covers the basic concepts of this work. An explanation of parallel computing and concurrency errors is given along with an explanation of dynamic analysis as an approach to detecting concurrency errors. And then, there is a short introduction to the topic of tool integration.

2.1 Paralelism

The introduction of multicore processors and multithreaded programs brought a whole new dimension of possible errors. Programs no longer consist of a deterministic sequence of commands. With multiple workers comes a need to coordinate each individual's work with the work of others, and management of shared resources is needed in order to assure data consistency. Lack of synchronization between threads can lead to nondeterministic program behavior. Every run of a multithreaded program can have a different thread-wide command execution order. This makes tracking and identification of concurrency errors very complicated because an error may, for example, occur only once in every hundred executions, or only occur when the CPU is under high stress.

The most common concurrency errors include data races, deadlocks, and order violation. A *data race* is an error leading to data inconsistency. It is an event of at least two threads accessing a shared memory location without any synchronization while at least one of the accesses is a write. A *deadlock* (or similarly livelock) is a situation when all threads are in a state of not making any progress because they are waiting for an event that could only occur if one of the waiting threads made progress. Deadlocks can cause programs to freeze and be stuck until externally terminated. *Order violation* stands for performing certain operations in an illegal order, such as writing to a file before its open, or accessing an object after it was deleted. This kind of error can cause program crashes or unexpected exceptions.

2.2 Dynamic Analysis

In order to make the process of error tracking and identification easier, various analysis algorithms were developed. Different analysers focus on detecting different errors, and have different characteristics depending on the used analysis algorithm. Usually, analysis algorithms simplify or extrapolate the program execution. A higher level of analysis extrapolation typically means less $false\ negatives$ but more $false\ positives$ and vice versa. Both dynamic and static analysis can be used to detect certain concurrency errors. This work

focuses on *dynamic analysis* on the binary level. The *SUT* (system under test) is analyzed during execution by monitoring certain event occurrences, such as specific function calls or memory accesses. The main advantage of this approach is no need for source codes, although debug symbols and information can help attributing errors to source-level statements. The disadvantages include a potential need to repeat the analysis multiple times before an error is detected, and significant execution slowdown because of the analysis overhead.

2.3 Tool Integration

Without a standardized interface the only way of integrating software tools with each other is by creating an adapter on a *tool to tool* basis. This makes it quite difficult and time consuming to integrate tools because the programmer needs to have good knowledge of the implementation of both tools being integrated, and the programmer needs to implement a specific adapter for the particular integration case, possibly even making changes to one of the tools in the process. Should the programmer decide he wants to switch the integrated tool with a different one, he would then need to create a whole new adapter specific to the new tool. To avoid this kind of specific adapter creation on a tool to tool basis, we need a standardized interface. With a standardized interface the only adapter a tool needs is the *standard interface adapter*, or the tool can even use the standard interface natively without the need for any adapter. There should be no extra work required in order to integrate tools through the standardized interface in the ideal case; however, it is typically still required to make some tool specific modifications because tools can, for example, have different execution parameters.

Chapter 3

ANaConDA Framework

The focus of this chapter is an introduction to ANaConDA intended for those not familiar with it already, or for those who would like to start using it. At first, we cover the basics of how ANaConDA works and what is it capable of. Then, a setup and usage guide is provided for the current version of ANaConDA.

3.1 Overview of ANaConDA

ANaConDA [40],[37] is a framework for dynamic analysis of concurrent C/C++ programs on the binary level and for easy creation of analysers. It was developed by the *VeriFIT* [6] research group of the *Brno University of Technology* [7]. ANaConDA uses *Intel PIN* [47] to perform dynamic instrumentation and insert monitoring probes into the analyzed binary on runtime. The probes connect to *callback functions*, which then allow analysers to monitor program events, such as memory accesses, synchronization operations, thread creation, function calls, and more.

Apart from monitoring, ANaConDA can also insert noise into the analyzed code, in order to cause rarer instruction interleavings to appear during program execution. *Noise injection* [38],[36] is performed by inserting sleeps, CPU yields, or busy-waits before or after a percentage of certain program events. Causing artificial slowdown to a thread can lead to otherwise very rare program states. This allows ANaConDA to detect errors which otherwise occur very rarely.

ANaConDA comes with several builtin analysers including ones for the detection of data races, deadlocks, and contract violations. But most importantly, ANaConDA allows for easy analyser creation by supplying the analyser with a higher level representation of the program state as presented by Intel PIN. To create an analyser, the programmer needs to declare the program events of interest by registering callback functions to be called before or after the corresponding event. Every time a monitored event occurs, the program control of the thread jumps into the body of the corresponding callback function in the analyser's code. There, the analyser has access to information about the event through callback parameters, and can perform any required actions before returning program control back to the analyzed program.

A diagram of how ANaConDA works is shown in Figure 3.1. The figure contains three main blocks: ANaConDA Framework structure (ANaConDA Framework), available analysers (Analysers), and the binary of a program under test (input program). We take the *AtomRace* analyser as an example to show the details of event monitoring. Under AtomRace

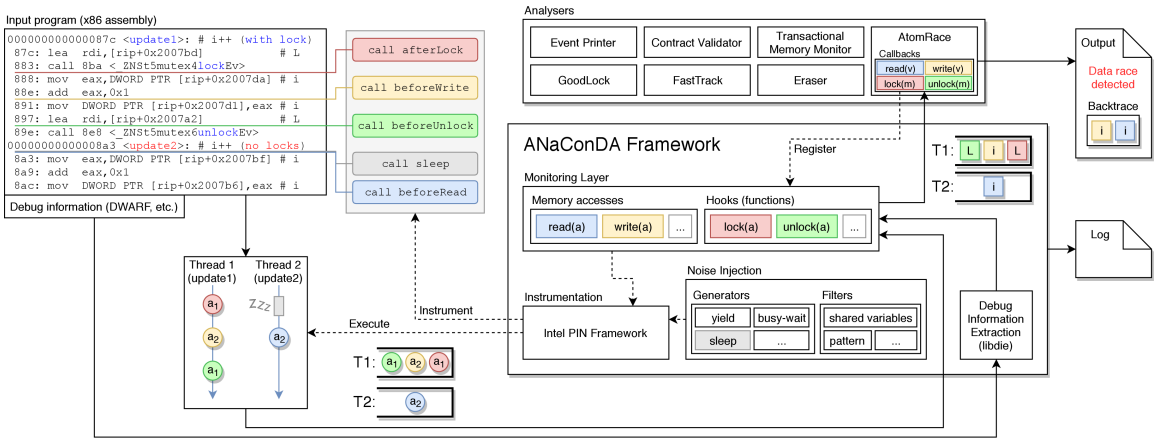

Figure 3.1: ANaConDA Framework diagram, source: [37]

in the *Analysers* section we can see that AtomRace uses four callback functions, ie. monitors four different types of events: read, write, lock, and unlock. These callback functions need to be *Registered* in the *Monitoring Layer* of the ANaConDA Framework. The Monitoring Layer then configures the *Instrumentation* based on the registered callback functions. At this point, extra instrumentation configuration can be added by the *Noise Injection* module of ANaConDA. Intel PIN *Instruments* the *input program* binary by inserting calls to the callback functions registered by the analyser, or calls to noise functions (e.g. sleep). The instrumented program is then *Executed* by Intel PIN. Monitored events can occur in different threads, meaning that callback functions of the analyser will also be executed by different threads (shown as boxes labeled *T1* and *T2*) during the program execution. The analyser then analyses the program execution based on its callback functions being called, and produces an analysis *Output*. The analyser also has access to debug information which is extracted from the input program binary by the *Debug Information Extraction* library. The ANaConDA Framework also produces a *Log* of its own.

Last year, ANaConDA received the *Best Tool Demonstration* award at *ISSTA 2018* [12], and currently takes part in the *AQUAS* project [1] and in the Czech national AUFOVER project [5].

3.2 Current installation and usage

Currently, ANaConDA can be used both on Linux and MS Windows by building it from source. This section gives instructions for the Linux version. It can be built and launched with two scripts, and configured using several configuration files. Unfortunately, in its current state ANaConDA is slightly complicated to setup and use on newer systems because it still uses an older version of Intel PIN, which requires legacy third party code. Migration to the new version is planned but will take a lot of work. Currently, it is possible to build and use ANaConDA if the system downgrades to gcc version 4.9.

Building ANaConDA is fully automated via its `build.sh` script. To set up ANaConDA, one should clone it's git repository and run the `tools/build.sh` script twice. The first run with the `--setup-environment` argument downloads all the ANaConDA dependencies, and the latter with the `all` parameter builds the ANaConDA Framework and all the included

analysers. Make sure to install gcc 4.9 and g++ 4.9, and set them as the default compilers, or the environment setup process will fail when trying to build its own version of gcc.

ANaConDA as an analysis tool is used through the command line by running the `tools/run.sh` script with at least two parameters: `analyser`, `binary_under_test`, and (optionally) arguments of the program under test. To configure noise and backtraces, modify the `framework/conf/anaconda.conf` configuration file.

To create an analyser for ANaConDA, one needs to register a new analyser in a configuration file `tools/conf/analysers/anaconda` specifying the path to its source code, and then build it using `tools/build.sh new_analyser_name`. The analyser's source code needs to have a function, in which it should provide registration of callback functions, called `PLUGIN_INIT_FUNCTION()`. The callback functions are listed in `anaconda/anaconda.h`. The recommended way of creating a new analyser is by copying the directory of one of the analysers included with ANaConDA and modifying it.

Chapter 4

OSLC - Open Services for Lifecycle collaboration

This chapter introduces OSLC (Open Services for Lifecycle collaboration) [23] and its core concepts along with the underlying technologies. The introduction focuses on the aspects of OSLC that were used in this work. For more information or a more detailed explanation refer to the OSLC Web site [23].

4.1 Foundation technologies of OSLC

OSLC is built on top of three main technologies. This section names those technologies and provides a general overview of what they are.

4.1.1 REST

REST (REpresentation State Transfer) [41] is an architectural style for Web based applications meant for communication between computer systems. REST is resource based meaning it works with things rather than actions. Resources can have different representations, typically *JSON* or *XML*. A *REST API* consists of several endpoints identified by a *URI* [34] to which it is possible to send different types of *HTTP* requests in order to perform certain actions. REST defines seven constraints: uniform interface, statelessness, client-server architecture, cacheability, layered system, and code on demand (optional). The most important one to understand for OSLC is uniform interface. REST defines four types of operations for resources: create, read, update, delete. These operations correspond to HTTP request types: `POST`, `GET`, `PUT`, `DELETE`.

4.1.2 RDF

RDF (Resource Description Framework) [30] is a data model for Web applications. Data is organized into triplets of *subject-predicate-object* creating relations between resources. RDF can have multiple representations, such as *turtle* [35], *RDF/XML*, or *JSON*. Databases for persistent storage of RDF data are called triplestores. *Triplestores* [33] are optimized for storing triplets and querying triplets using a specialized query language, *SPARQL* [44].

4.1.3 Linked Data

Linked Data [13] is a set of rules defining how to publish and connect data on the Web in order for the data to be machine readable and easily connected with its context. The ultimate goal of Linked Data is for the whole Web to be accessible as a single global database. Linked Data has four basic rules: „1) Use URIs as names for things 2) Use HTTP URIs, so that people can look up those names. 3) When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL). 4) Include links to other URIs, so that they can discover more things.“ [13]

4.2 Overview of OSLC

OSLC [23] is an *OASIS Open Project* [21] focused on interoperability and communication of software tools throughout the whole software development lifecycle. A key aspect of OSLC is specifying only the minimal amount needed for a particular integration scenario, so that the standard is easy to adopt and usable for as wide range of integration cases as possible. As a result, OSLC specifications contain minimal obligatory requirements for standard conformance, and then contain many other requirements for further optional extensions. OSLC uses self-describing *RESTful APIs* [41, Chapter 5], RDF 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` (Create, Read, Update, Delete) HTTP requests. And each resource has an RDF representation, such as *RDF/XML*, *XML*, or *JSON*. Finally, resources can be linked together by URI identified *relations*.

OSLC consists of a number of specifications defining resources and RDF resource shapes for different application domains, such as *Quality Management*, *Architecture Management*, *Change Management*, or *Requirements Management*[1]. All the application domain specifications are based on the *Core domain* [24]. Key elements of OSLC specifications are usage rules and patterns for HTTP and RDF, resource shapes and constraints, resource representation, resource operations, authentication, resource discovery, and resource querying. Figure 4.1 shows the layered architecture of OSLC.

Currently, the latest version of the OSLC specification is 3.0 but some domains are still only at version 2.0 or 2.1. This work focuses on OSLC Core 2.0 and OSLC Automation 2.1.

4.3 OSLC Specifications

OSLC is divided into work groups. Each work group focuses on a specific integration scenario referred to as a *domain*. A domain can be imagined as a standardized interface. They define resources that an interface should use as its elements which are then used to communicate between software tools. In order to keep all domains coherent there is the *OSLC Core* domain managed by the core work group on top of which all the other domains are based. In this work we have used the *OSLC Automation* domain. Specifications define requirements with three levels of importance - $MUST$, $SHOULD$, and MAY. For a tool to be compliant to the specification, it needs to satisfy all requirements marked as MUST.

[1] For OSLC domain specifications refer to [27]

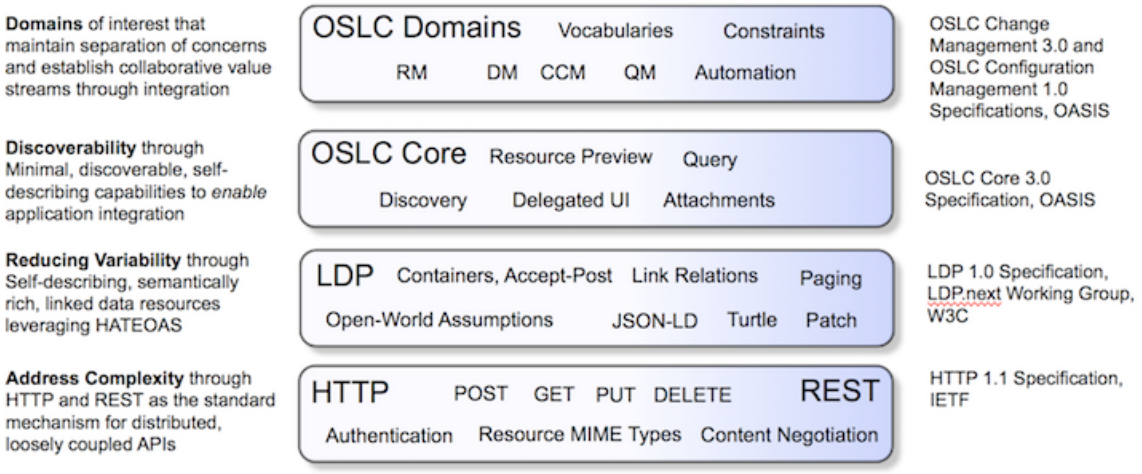

Figure 4.1: OSLC layered architecture, source: [22]

4.3.1 OSLC Core Domain

The Core domain [24] specifies the basics shared by all other specialized domains. It specifies how to use HTTP and RDF in a standardized way. OSLC is based around *Resources*. Resources are uniquely identified by a URI, and their type is specified by the URI of the type. Resource types are defined by *Resource Shapes* which are RDF resources containing a list of *Properties* and information about those properties, such as `occurrence`, `value type`, or `allowed values`. Types are organized into XML namespaces which are defined as a tuple of namespace prefix and namespace URI. Resources and properties are then identified using namespace prefixes, e.g. `oslc:properties`. Most resource properties are optional which is determined by their occurrence. Usually a resource will have just one or two properties with occurrence of `exactly-one` or `one-or-many`, and the remaining properties will have occurrence of `zero-or-one` or `zero-or-many` making them optional. The Core domain also defines error responses, resource paging, authentication, resource operations and delegated UI dialogs.

Basic Capabilities

Basic operations [24] defined for resources are CRUD operations: Create, Read, Update, Delete. Each of these operations is a separate capability, and a resource will typically support only some of them, depending on the application domain specification and the actual implementation. **Creating** a resource is performed by sending a `POST` request to a dedicated *creation factory* URI. The POST request body has to contain the resource to be created with all the required properties based on the resource's resource shape. The body representation can be RDF/XML, JSON, or XML depending on the specific implementation. **Reading** a resource is performed by sending a `GET` request to the resource's URI. The request should use the `Accept` header to specify what kind of resource representation is requested. **Updating** a resource is performed by sending a `PUT` request to the resource's URI with the changed resource in the request body, same as with resource creation POST requests. The recommended way of updating a resource is reading the resource, modifying the response body, and then sending it back in the update request. **Deleting** a resource

is performed by sending a `DELETE` request to the resource's URI. The response to any of the CRUD requests will use a HTTP status code, such as `200 OK`, `201 Created`, `400 Bad Request`, `404 Not Found`, or `405 Method Not Allowed`.

Query Capability

Query capability [24] is used to list all resources or to find a subset of resources. It has its own URI to which `GET` requests need to be sent with an `Accept` header set to determine the requested resource representation. Sending a GET request with no parameters will result in receiving a list of all resources, typically divided into pages of a smaller amount of resources to avoid oversized responses. Request parameters `page` and `limit` are used to cycle through pages or change the page size. The OSLC Query Syntax specification [50] defines query parameters that can be added when using query capabilities to filter the results. The query syntax is provided but so far is optional. The specification defines parameters such as `oslc.where`, `oslc.searchTerms`, `oslc.orderBy`, or `oslc.select`.

Delegated UI Dialogs

One of the seven constraints of RESTful APIs is the optional constraint *code on demand*. OSLC defines this functionality as two delegated UI dialogs: *selection dialog* and *creation dialog*. The purpose of these two dialogs does not need to be further explained as it is clear from their names. They should be implemented using a combination of HTML `<iframe>` and JavaScript to allow integrating tools to use the integrated tool specific UI as part of their own UI.

OSLC Provider and OSLC Consumer

There are two types of OSLC applications, clients and servers. An *OSLC Provider*[2] is a server that manages OSLC resources from at least one domain and implements a set of operations as capabilities for the managed resources. An *OSLC Consumer*[3] consumes the provider's services by manipulating the provider's resources through the provider's capabilities. A consumer can discover what resources and capabilities are provided by a provider using *Resource Discovery*. One way of implementing resource discovery is through a service provider catalogue. A *Service Provider Catalogue* is a bootstrap point for resource discovery and the only point that needs to be known to the consumer. The catalogue contains a list of service providers. A *Service Provider* contains a set of services. A *Service* is a set of capabilities and their URIs. See Figure 4.2 for a graphical representation of the above described resources and their relations.

When an OSLC consumer wants to consume services provider by an OSLC provider, all he needs to know is the URI of the service provider catalogue. From there he can navigate to any of the provided capabilities, such as resource creation factories or query capabilities. Resources can be created by sending a POST request to a creation factory with the resource to be created specified in the request's body. The body of a creation POST request can be filled in based on a resource shape which should be provided by the OSLC provider for all managed resources. Once a resource is created the consumer can read, update, or delete the resource by sending GET, PUT, or DELETE requests to the resource URI, assuming

[2]In OSLC Core 3.0 the term *Provider* was deprecated and replaced with *Server*
[3]In OSLC Core 3.0 the term *Consumer* was deprecated and replaced with *Client*

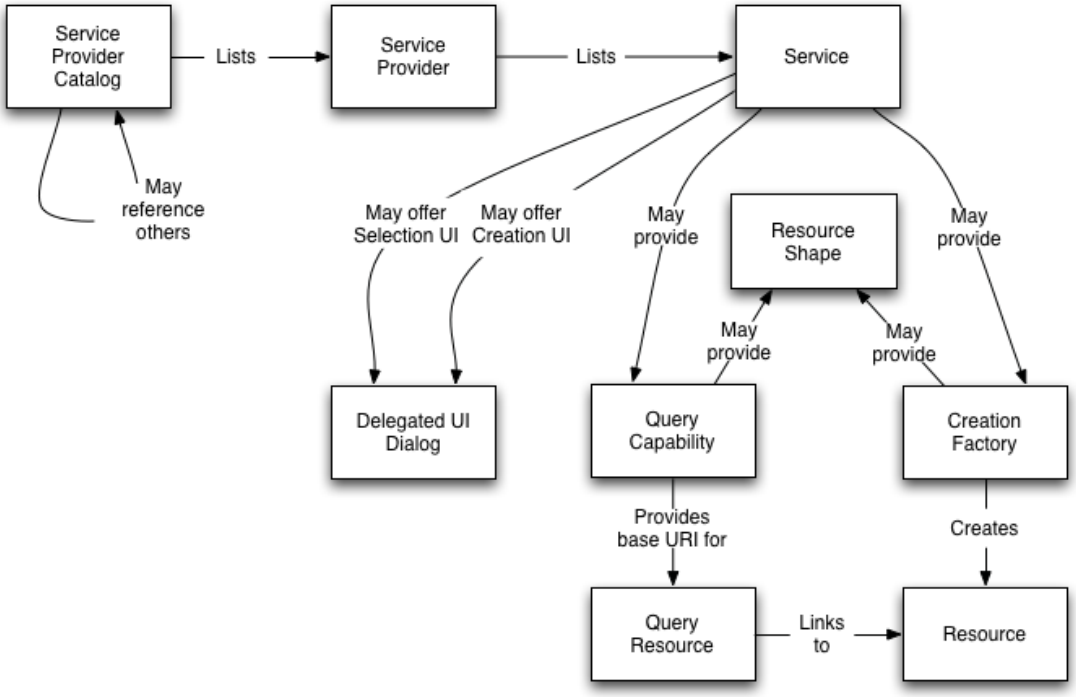

Figure 4.2: OSLC Provider elements and their relations, source: [24]

the OSLC provider supports all of those capabilities for that particular resource. Querying for resources can be done by sending a GET request to the query capability URI.

4.3.2 OSLC Automation Domain

The OSLC Automation domain [25] 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 fetch the results of the execution. The core resources of the OSLC Automation domain are *Automation Plans*, *Automation Requests* and *Automation Results*. Their relations are shown in Figure 4.3.

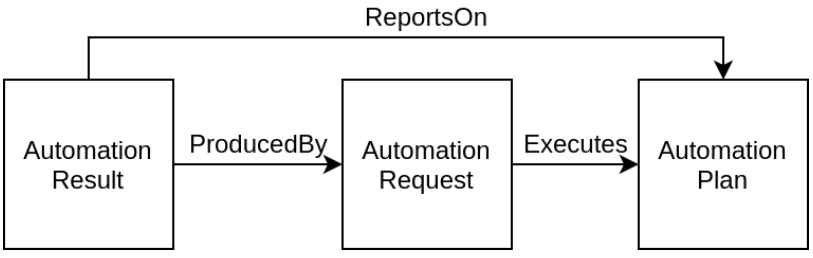

Figure 4.3: OSLC Automation resources and their relations [25]

Automation Plan

Automation Plans [25] represent the units of automation available for execution. Their main role is to define input parameters for Automation Requests using `parameterDefinition` reference properties. `Parameter Definitions` use the resource shape `oslc:properties` [24] which are resources with properties such as `name`, `allowed values`, `occurrence`, `value type`, or `default values`. The required capability for Automation Plans is only GET (reading the resource), and then a selection dialog capability is recommended. Parameter Definitions have no required or recommended capabilities. A basic scenario would be an OSLC provider with a single predefined Automation Plan that can be viewed by OSLC consumers, and used as a guide for creating Automation Requests.

Automation Request

Automation Requests [25] are created by OSLC consumers to request execution of an Automation Plan. The most important properties of a request are `inputParameter`, `state`, and `executesAutomationPlan` reference. Input parameter properties reference `Parameter Instance` resources (discussed in Section 4.3.2) which are also used for input parameters and output parameters of Automation Results. The required capabilities for Automation Requests are GET and POST (reading and creating the resource), and then a creation dialog capability is recommended. When creating a request the OSLC consumer needs to choose an Automation Plan, and provide input parameters for the Automation Request based on parameter definitions of the Automation Plan. After the Automation Request is created the OSLC provider will create an Automation Result. It is the consumers responsibility to poll for the result based on the Automation Request's state which indicates whether the execution has finished or not.

Automation Result

Automation Results [25] are produced by the OSLC provider based on a Automation Request. The most important part of a result are its `contribution` properties which reference various resources that were produced by the Automation Request execution, such as text output logs, binary files, or images. The contribution resource should have a `title`, a `description`, and a `type`. Other important properties of an Automation Result are `verdict`, `state`, `producedByAutomationRequest`, and `reportsOnAutomationPlan`. The state has the same values as the Automation Request state, and these two states should be somewhat consistent. The verdict property represents the result itself, ie. whether the execution passed, failed, or encountered an error. An Automation Result references the same input parameters as the originating Automation Request, and in addition contains output parameters. `Output Parameters` represent input parameters which had their values changed during execution, or they can represent parameters added by the OSLC provider during execution. The required capability for Automation Results is GET (reading the resource), and the recommended capabilities are PUT (updating the resource) and a selection dialog. The update capability is used to allow other agents, other than the OSLC provider, to add contributions to the result, in case there are any.

Parameter Instances

Parameter Instances [25] represent input and output parameters of Automation Requests and Automation Results. Their properties include `name`, `value`, and `description`. OSLC defines no required or recommended capabilities for Parameter Instances.

4.4 Eclipse Lyo

Eclipse Lyo [10] 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.

4.4.1 OSLC4J

OSLC4J [10] is a Java toolkit for developing OSLC providers and consumers. It contains Java object annotations for OSLC attributes and UI previews, support for service provider and resource shape documents, libraries for developing providers and consumers, sample applications, and a test suite. Two alternatives to OSLC4J are currently available or being developed. OSLC4Net is a toolkit for .NET environments, and OSLC4JS is a set of projects for JavaScript applications.

4.4.2 Lyo Designer

Lyo Designer [10] is an Eclipse plugin meant for development of OSLC providers and consumers based on the OSLC Core 2.0. specification. The designer consists of two parts: Lyo Toolchain Designer and Lyo Code Generator.

Lyo Toolchain Designer is a graphical modeling tool built with Eclipse Sirius. It is capable of modeling OSLC domains, OSLC vocabularies, OSLC toolchains, and adapter interfaces. Modeling a domain is done by creating a *Domains Specification Diagram*, creating a domain in it, creating the domain's resource shapes and properties, and configuring their properties, such as domain prefix (XML namespace), property occurrences, value types, or representations. A *Toolchain Model* consists of adapter interfaces and their relations. Each adapter can provide or consume certain resources. Consumed resources can be resources provided by one of the other adapter interfaces creating a connection between the two adapters and forming a toolchain. Each adaptor interface needs to have its internal structure modeled by an adaptor interface diagram. *Adaptor Interface Diagram* is a tree graph with a service provider catalogue as its root. Multiple service providers can be created, and each service provider can have multiple services. Services are what holds resource capabilities. Available capabilities are: basic capability (read, update, delete), creation factory, query capability, and selection and creation dialogs.

Lyo Code Generator can generate code using OSLC4J based on the models created with the designer. For the modeled domains the generator generates annotated classes for all resources and their resource shapes. For the modeled adaptor all the required logic to run a working Maven Web application is generated. And for all the modeled capabilities the generator creates placeholder functions to be implemented by the user, which are called when the adapter receives a HTTP request on the corresponding capability's URI. The generator places designated blocks in the generated code designed to protect user added code from being lost during the generation process. This allows the adapter to be developed iteratively. The generator also creates a basic Web UI, however, the generated methods

are annotated as deprecated in Lyo Designer 2.4 and need to be manually customized to provide useful functionality.

To create a single adapter the user needs to create a Maven Web application project and configure it for OSLC4J development. Then, the user needs to model all managed domains with all their resources and relations. Then, the actual adapter interface has to be modeled by adding managed resources and creating service providers, services, and all the required capabilities for the managed resources. Once everything is modeled, it is possible to generate code based on the models to fill in the previously created OSLC4J project. And then, the last things remaining are to populate the service provider catalogue (service provider infos function), and to implement placeholder functions for resource capabilities, tying them with the application being adapted.

4.4.3 Lyo Domains

Lyo Domains [15] is a Git repository containing models and generated classes of all the OSLC application domains. The reference domains can be used directly by importing the generated classes, or they can be imported as models into Lyo Designer to generate the classes yourself, modifying them beforehand if needed.

4.4.4 Lyo Store

Lyo Store [17] is a library for persistent storage of OSLC resources in a triplestore. Three different storage types are supported: `in-memory`, `on-disk`, and `SPARQL`. The original functionality was only to perform basic operations with resources (CRUD). However, eventually experimental implementation of the query capability based on the OSLC Query Syntax [50] was added, and it is still being developed. The query capability is currently only available for the SPARQL storage type. When used with Lyo Designer, Lyo Store makes it easy to add persistance and query capabilities to an OSLC provider.

4.4.5 Lyo RIO

Lyo RIO [16] is a set of sample implementations of the OSLC specifications meant as minimal implementations to be used as a reference for full implementations. It features sample providers and consumers to experiment with.

4.4.6 Lyo Test Suite

Lyo provides a test suite [19] used to verify whether a tool or adapter is compliant with the OSLC specifications. The test suite is based on JUnit, and provides tests for most domains. The tests are tailored for the sample implementations of OSLC providers available in Lyo RIO, and then there are some community made application specific tests. The test suite covers all resource shapes, all basic capabilities, and optionally even a simple version of the query capability.

Chapter 5

OSLC Adapter Design

This chapter covers the design of the Anaconda OSLC Adapter. The design was modeled using Lyo Designer 2.4 (Section 4.4.2) in a Sirius modeling project, and the created diagrams are presented here along with their detailed descriptions. This chapter covers the part of the adapter creation required before code generation which is discussed in the next chapter (Chapter 6). Most of the work in this chapter is essentially based on these two tutorials: [31], [32].

5.1 Adapter Architecture

There are three possible ways of adding OSLC support to an existing software tool. The first two, native support and creating a plug-in, require the tool to be a Web application. Since ANaConDA has no Web functionality, the only option for adding OSLC support is the third one, creating a new Web application. Thus, the adapter is a Java Web application managed by Maven and based on a Maven Web application archetype. Java was used because it currently has the most tools to aid OSLC provider creation available. Apache Maven [3] was used as the project manager because OSLC4J development projects use Maven dependencies, and because the adapter is launched using the jetty-maven-plugin [9]. Once launched the Adapter acts as a server with REST endpoints for OSLC capabilities and resources.

5.2 OSLC Domain Models

First step of modeling the adapter is to model the managed domains. To avoid modeling all the domains from scratch, we have used the reference domain models available in the Lyo Domains repository.

5.2.1 OSLC Core 2.0

The core domain needs to be managed by all OSLC providers. It contains resource shapes for resources such as *Property*, *ResourceShape*, and *ServiceProvider*, and it defines properties for the Property resource. The core domain model also contains other basic useful domains, namely *Dublin Core*, *FOAF*, *RDF*, and *RDFS*. Figure 5.1 shows the reference model exported from Lyo Domains using Lyo Designer.

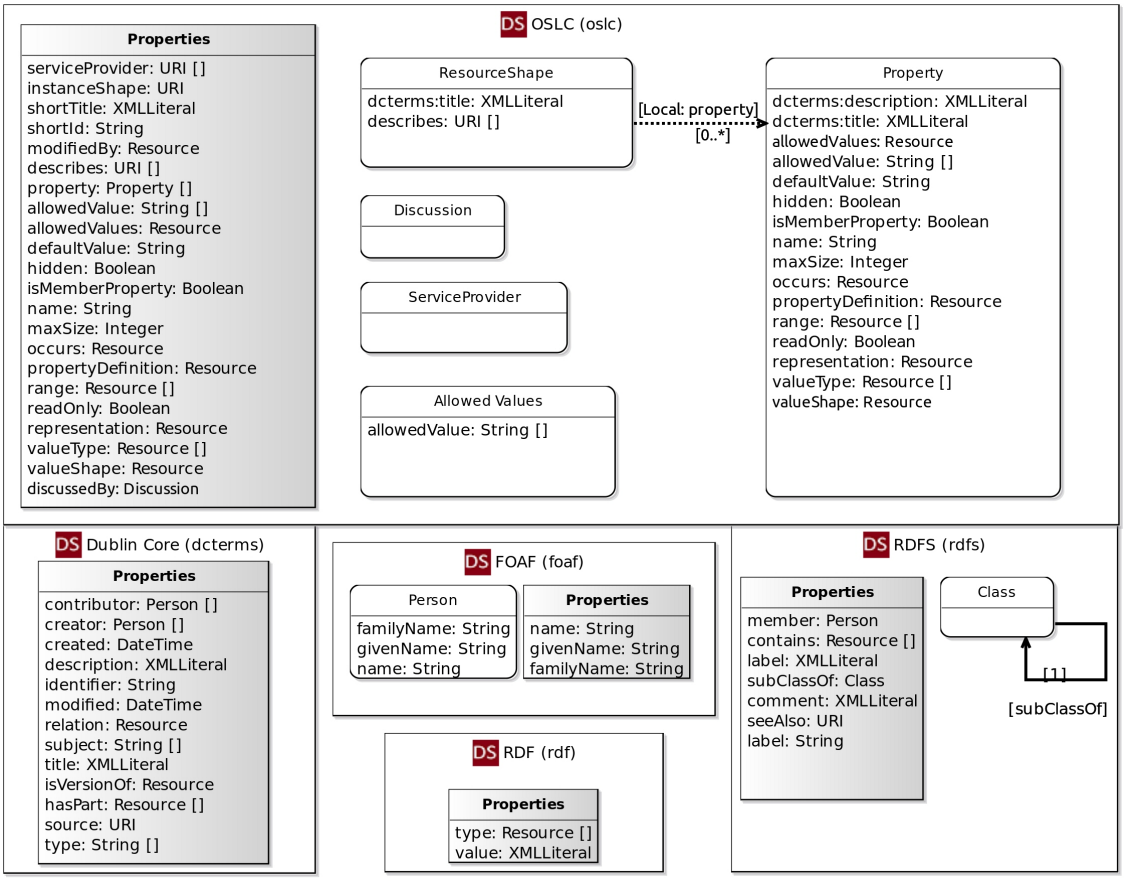

Figure 5.1: OSLC Core 2.0 domain model

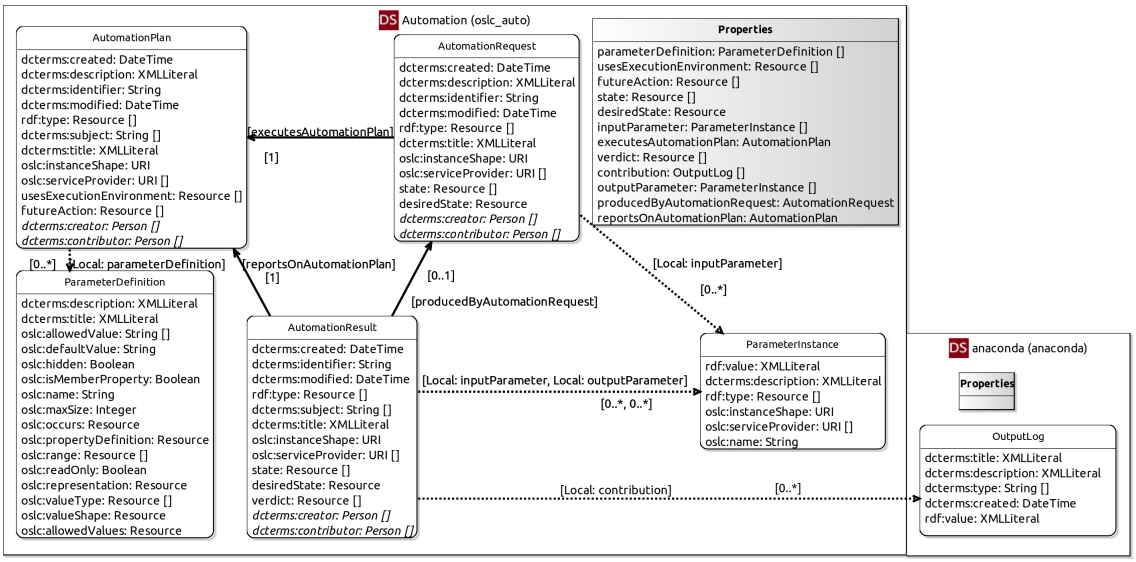

Figure 5.2: Specification Diagram - OSLC Automation domain model and ANaConDA domain model

5.2.2 OSLC Automation 2.1

The OSLC Automation domain and its resources are discussed in Section 4.3.2.

The reference model for the OSLC Automation domain was imported from Lyo Domain, however, some modification to the model have been made. Namely, properties `parameterDefinition`, `inputParameter`, and `outputParameter`, have their types modeled as `Resource` in the reference models, meaning that the value of the property would be a link reference to a resource. The specification says that these properties can be represented both as a *reference* and as *inlined* resources. For the ANaConDA Adapter we have decided to represent these properties as inlined resources so that the resources can be created with a single POST request along with the resource that contains these properties. To achieve this functionality the types of these resources had to be changed from `Reference` to `Local Resource`. Another change to the reference model was creating a new resource shape, `Parameter Definition`, which is the same as the OSLC Core `Property` resource shape only its `propertyDefinition` property is optional as defined by the OSLC Automation specification. This new type of resource is used as values for the `parameterDefinition` property of Automation Plans. A new ANaConDA specific domain was created to hold a resource shape for `Output Logs` which are used as values for the `contribution` property in Automation Results. Figure 5.2 shows the modified Automation domain model and its connection to the ANaConDA domain.

5.3 ANaConDA Adapter Model

The adapter's interface is displayed in Figure 5.3. The adapter manages all the OSLC Automation domain resources along with the added Parameter Definition (copy of the Core Property resource) plus Output Logs from the ANaConDA domain, and the adapter does not consume any resources.

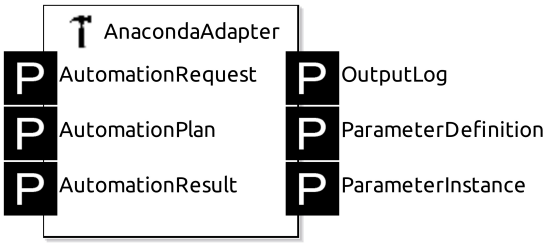

Figure 5.3: Toolchaing Diagram - ANaConDA Adapter interface model

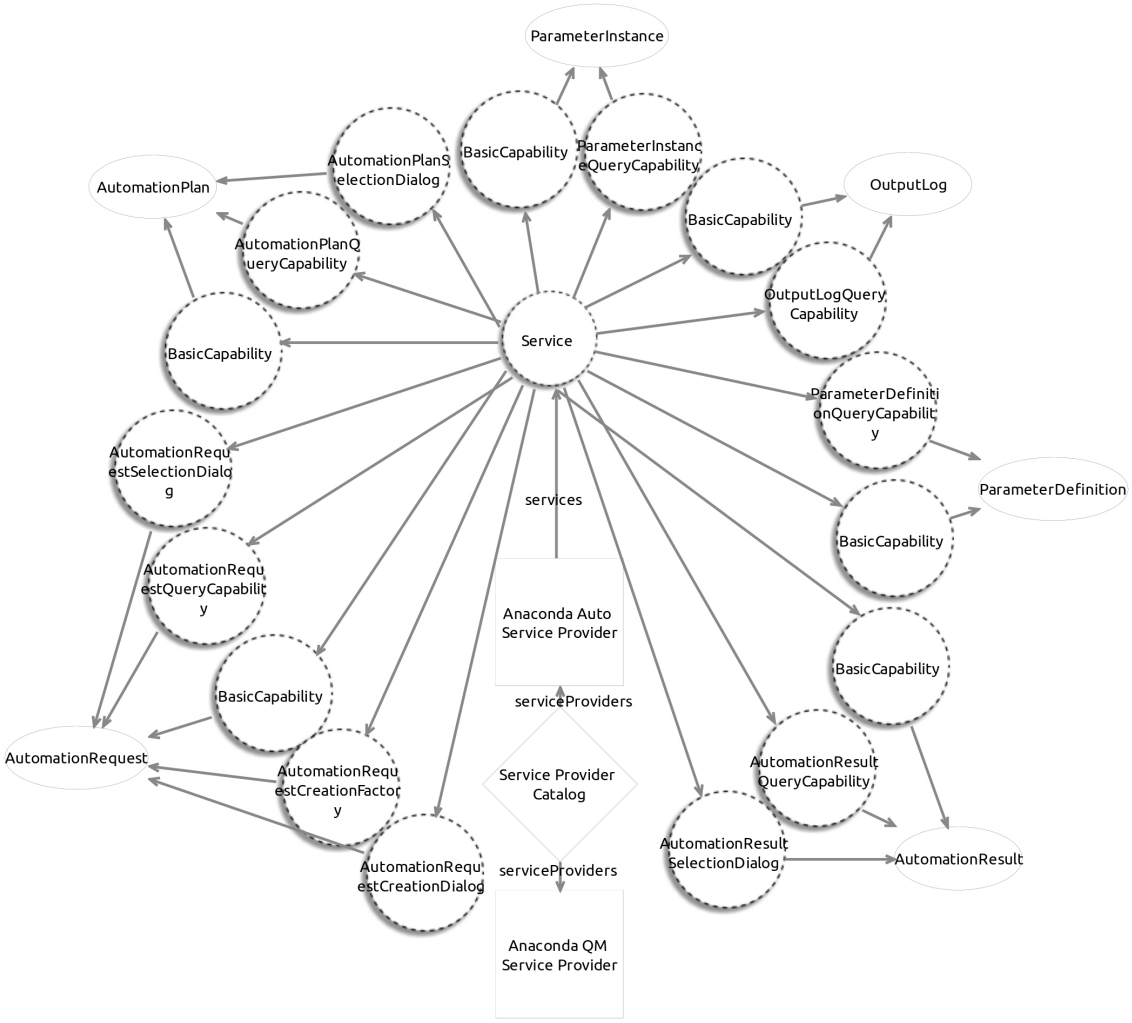

Figure 5.4: Adapter Interface Diagram - ANaConDA Adapter internals model

The internals of the adapter are displayed in Figure 5.4. The service provider catalogue contains two service providers. First is the Automation service provider that holds all the current functionality of the adapter, and second is the Quality Management service provider which stayed only as a placeholder for possible future extensions of the adapter[1]. The Automation service provider contains one service that holds all the capabilities for all resources. There was a version of the adapter with two services with the second service managing only `anaconda:OutputLogs`. It was decided to remove the second service as it added unnecessary complexity to the REST interface and its endpoint URIs. The adapter features all the `MUST` and `SHOULD` specified capabilities for each resource according to the specification, and all resources have a query capability. Basic capabilities can represent three different actions at once: read, update, and delete (only visible in detailed properties, not the diagram). The only updatable resource is the Automation Result, as advised by the specification, and the only resource which can be deleted is also the Automation Result. The specification does not give any recommendations on resource delete capabilities. We have decided to allow Automation Results to be deleted in case the persistent storage became too large, or in case a result was just a test resource or a broken resource. When an Automation Result is deleted by the user, the adapter also deletes the corresponding Automation Request and its parameters to keep the persistent storage consistent. An alternative approach would be to allow users to delete both Automation Results and Automation Requests independently, but this could cause inconsistencies such as a Result that references a deleted Automation Request. The only resource with a creation capability is the Automation Request. All other resources are created by the adapter as a result of a Automation Request being created by the user. Automation Plans are currently predefined and cannot be changed or created because the specification does not say they need to be creatable or updatable. Also since Automation Plans define units of automation they need to have a specific implementation in the adapter. An alternative would be to allow users to create Automation Plans, and specify the analyzed program this way. That would mean Automation Results would be clearly linked to a specific program instead to an analysis scenario. This would, however, alter the intended semantics of Automation Plans as they appear to be presented in the specification. The model includes creation and selection dialogs, creation for Automation Requests, and selection for Automation Results and Automation Plans. However, the dialog capabilities are not implemented because they are not required by the specification, and because so far there is no known usage for them in ANaConDA.

5.3.1 OSLC Automation Resource Mapping to ANaConDA

The OSLC Automation domain (Section 4.3.2) is perfectly suited for the core usage of ANaConDA (or any command line utility with the same usage).

To analyse a program using ANaConDA, the user runs a `run` script with some input parameters. The user can discover what input parameters to use using the `--help` parameter of the script. The basic parameters are: analyser to use, program to analyse, and any number of input parameters for the analysed program. Once launched, ANaConDA outputs analysis information to the standard output, essentially creating an output log. The OSLC Automation domain represents this process through resources. It contains three

[1]Originally the idea was for the adapter to model the Quality Management domain, modeling test cases, test plans, test results, etc. Later we found that the OSLC Automation domain is better suited for the current functionality of ANaConDA. However, it still is possible that an extension for the Quality Management domain may be added in future.

main resources: Automation Plans, Automation Requests, and Automation Results. The usage of ANaConDA translates like this:

1) `run.sh --help` → get an Automation Plan
2) `run.sh analyser program input` → create an Automation Request
3) read `output.log` → get an Automation Result

Automation Plans act as the `--help` parameter of ANaConDA's `run` script. The user can look at Automation Plans to find out what input parameters to use when requesting analysis of a program. Automation Requests represent the act of the user calling the `run` script with some input parameters. Creating a request causes the adapter to execute analysis of the requested program using ANaConDA and create an Automation Result for the request. Automation Results represent the output log of the analysis, and can be read or otherwise manipulated by the user.

By translating the steps of using ANaConDA to resources, we replace an action (executing a script) with a resource (Automation Request) which can then be persistently stored and have connections to other resources.

Chapter 6

OSLC Adapter Implementation

This chapter begins at the point of code generation using Lyo Designer 2.4 based on the models created in the previous chapter (Chapter 5). It covers the process of code generation and the major parts of the adapter's implementation, including the persistance solution not required by the specification. During development we have encountered some bugs and issues when working with Eclipse Lyo; all of those are described later in this chapter. Most of the work in this chapter is also essentially based on these two tutorials: [31], [32].

6.1 Base code generation with Eclipse Lyo

Once domains, adapter interface, and adapter internals are modeled, the user needs to configure the code generation and create the OSLC4J development project for the code to be generated into. The base project is a Maven Web application archetype with dependencies added for OSLC4J development. Before generating the code, there are two things that need to be configured, the second being an optional quality of life improvement. First, it is essential to configure the properties of the adapter interface in the Toolchain Diagram, namely the Java classes `base namespace` and `file paths` for Java, Jsp, and JavaScript files. This configuration should match the package and the paths of the earlier created OSLC4J Maven Web application project. Second, by creating a file named `generator.properties` in the modeling project and placing a `generationPath` property in it, you instruct the code generator which directory to use as the root for the code generation. Without this setting, the path would need to be selected manually every time the code is generated. Once all the above is done, the code can be generated.

6.1.1 Lyo Designer Bugs and Issues

We have encountered four issues with Lyo Designer during the adapter's development. Two of those issues are already tracked as bugs in the Lyo Git repository [14], or have been discussed on the OSLC forum [26]. We are discussing the remaining issues with the creators of Eclipse Lyo.

Inconsistent Getter and Setter Types

This is a confirmed bug in Lyo Generator [28]. When the domain model contains any properties with occurrence of `X-to-Many` the generated code will contain an inconsistency in the return types of getter methods and parameters of setter methods. When using Lyo Designer

2.4 this bug manifests when a consumer tries to create a resource by POSTing to a creation factory. The consumer will receive an error response with a message mentioning this exception: `OSLC011: Missing corresponding set method for method getStep of class %Generated DS class%`. When using an older version of Lyo Designer this bug prevents the adapter from being built and launched giving the same exception causing an error.

This bug can be fixed by changing the return type of all setters in the generated resource classes to a `Set` instead of a `HashSet`. We have made a bash script[1] which does this fix using the `sed` command line utility to temporarily hack around this bug until it gets fixed by the creators of Lyo Designer.

OSLC Core Classes Generated in a Wrong Directory

This is a confirmed bug [8]. The code generator generates some classes for the OSLC Core domain for all adapters. The classes and files, however, are generated in the generation root path instead of being generated in the adapter's directory along with its explicitly managed domains. When the Toolchain Model contains only one adapter, this bug is not an issue because it can be fixed by setting the generation root as the adapter's directory.

Managing OSLC Core Resources Causes Inconsistencies

This issue is currently being discussed with the creators of Lyo Designer. Every adapter uses resources from the OSLC Core domain. If these resources are not modeled in any of the adapter's models, then the generated code will not contain almost any classes from the Core domain, and they will be imported from a Maven dependency instead. However, if the adapter explicitly manages any of the Core resources, the code generator will generate classes for those resources, and the generated classes will be different than the ones imported from the Maven dependency. which will cause errors when trying to build the adapter. We are using the reference model of the Core domain imported from Lyo Domains, so there should be no errors in modeling the domain on our end. We encountered this issue during the modeling of the OSLC Automation domain because the property $parameterDefinition$ in Automation Plans references the $Property$ resource from the Core domain. We have solved this issue by creating a copy of the $Property$ resource shape inside of the OSLC Automation domain and naming it $ParameterDefinition$ because a similar approach can be seen in the reference implementation of an OSLC Automation provider in Lyo RIO.

Missing Domain Prefix Definitions

This issue is currently being discussed with the creators of Lyo Designer. The code generator generates a `package-info.java` file that defines domain prefixes. If these definitions are missing, the adapter will generate its own prefixes using a `j.X` pattern, where X is a number starting at 0 and incrementing with each generated prefix. The generator seems to generate only one such file for the whole adapter even though a copy of this file should be in every directory of every resource class package. This issue was fixed by copying the `package-info.java` file into each domain folder.

[1] dev_tools/fix_getters.sh

6.2 Implementing the OSLC Adapter's Functionality

This section presents the implementation details of the most important parts of the adapter. The implementation currently uses one predefined Automation Plan with multiple input parameters, and execution of an Automation Request is divided into multiple stages.

6.2.1 Predefined Automation Plan

The adapter does not support Automation Plan creation, instead the adapter creates one Automation Plan for OSLC consumers to query and reference. The Automation Plan supports analysis of programs that consist of a single source file, and it has six input parameters:

1. **Analyser** - Name of the analyser to use. Can be any analyser present in the local ANaConDA installation.
2. **Program** - Program to analyze. Value representation is selected in the ProgramDefinition parameter.
3. **ProgramDefinition** - How to interpret the Program parameter value. Options are:
   - `url download` - The program will be downloaded from a URL
   - `base64 string` - The program will be decoded from a base64 encoded string
   - `filesystem path` - The program will be copied from the adapter's local filesystem
   - `console command` - The program to analyze is a console command
4. **ExecutionParameters** - Optional parameter. Execution parameters for the analyzed program.
5. **CompilationParameters** - Optional parameter. Compilation parameters for gcc. The default value is `"-g"`.

Possible future extension would be to add another Automation Plan or to add more parameters to the current one, in order to add support for analyzing programs consisting of multiple source files, most likely compiled using a Makefile. Another possible extension would be to allow the program to analyze to be specified using a Git repository.

6.2.2 Automation Request Execution

Executing Automation Requests is the core of the adapter's functionality. The execution consists of three parts: fetching the program to test, compilation, and analysis execution. Each of those stages have a separate output log *contribution*: Fetching Log, Compilation Log, and Analysis Log. The adapter should respond to consumer requests with minimal delay. Since most of the execution stages can take a long time (especially the analysis), they have been moved into a separate thread. When an OSLC consumer creates a new Automation Request, the adapter immediately creates an Automation Result with its state set as `inProgress`, and returns it to the consumer. Then, a new thread is created to execute the Automation Request. When the execution finishes, the previously created Automation Result is updated by adding output logs as contributions, and changing its state and verdict to `completed` and `passed` respectively. Finally, the state of the Automation Request is

updated in the same way as the Result's state. The Automation Request execution consists of three stages described below.

Program Fetching

The first stage of Automation Request execution is transferring the source code of the program to analyze to the adapter. The predefined Automation Plan offers four different ways of submitting the program to analyze (see Section 6.2.1). The program's source is fetched into a `tmp` directory in the adapter's root directory. If the program to analyze is not found or the fetching fails (e.g. due to loss of connection), the Automation Result of the Request will have a its verdict set as `error`, and the `Fetching Log` contribution will contain the error message.

Program Compilation

The second stage of Automation Request execution is the compilation of the program to analyze. The adapter builds the program using the `gcc` compiler with the compilation parameters specified as the input parameter of an Automation Request. The compiler is called using `java.lang.Runtime.exec()`, and all its output is captured by the adapter and saved into the `Compilation Log` contribution. The compiled binary is saved in the `tmp` folder next to the corresponding source file. If the compilation fails, the Automation Results verdict value will be set to `error` and the `Compilation Log` contribution will contain the output of gcc.

Analysis Execution

The third and final stage of Automation Request execution is performing the analysis. ANaConDA is called using `java.lang.Runtime.exec()` by calling the `tools/run.sh` script with the analyser to use and execution parameters specified as the input parameter of an Automation Request. Path to ANaConDA has to be properly configured in the adapter's Java properties configuration file (see Appendix A). Output of the analysis is stored in the Automation Result as the `Analysis Log` contribution. If the analysis is executed with no errors, the verdict of the Automation Result is set to `passed`.

6.2.3 Persistence and OSLC Query

Eclipse Lyo includes Lyo Store (see Section 4.4.4), a library for persistent storage of OSLC resources. If used with a SPARQL triplestore, Lyo Store provides basic query capability using the OSLC Query syntax [50]. Although the query capability is not yet fully implemented, it is still being developed, meaning the adapter's query capability will improve over time as Lyo Store improves. We have created a wrapper class for Lyo Store to create a specific interface for the ANaConDA OSLC Adapter. The wrapper allows for resources to be read, stored, deleted, updated and queried through methods that are resource specific because each resource needs to be handled based on its specific properties. The reason for specializing these methods is that when resources are retrieved from the triplestore, they do not include their inlined resources and instead include only a reference to them. The resource specific wrapper method needs to retrieve all referenced resources one by one so that they would be properly inlined in the adapter's response to an OSLC consumer's request. The situation is very similar when updating a resource in the triplestore because each

inlined resource needs to be updated individually, otherwise resources can have multiple values for the same property, possibly corrupting the triplestore. It seems that Lyo Store performs a proper update for the main resources and then only a simple put for the inlined resources without removing their old values. The same also applies for deleting resources as well; each inlined resource has to be deleted individually.

SPARQL Triplestore

As mentioned before, a SPARQL server is required in order for Lyo Store to provide query functionality. We have used the Jena Apache Fuseki server [2] which is a SPARQL triplestore that offers multiple deployment options. We have chosen to use Fuseki as a WAR file deployed in a Jetty container. The Fuseki server and a Jetty distribution [9] is included with the adapter, however, using the Fuseki server is optional if the user already has a SPARQL triplestore or wishes to use a different one. All that is required to use a different SPARQL server is to change the SPARQL endpoint URIs in the adapter's configuration (see Appendix A). A SPARQL triplestore needs to be available when launching the adapter, otherwise the adapter reports an error and does not start. Starting the adapter without a triplestore available would cause all OSLC consumer requests to fail. Should the triplestore become unavailable while the adapter is running, the adapter will respond with an error to most consumer requests and report that the triplestore is unavailable in its console.

A Bug Discovered in Lyo Store

During the adapter's development we have discovered a previously unknown bug in the Lyo Store library. We have reported the bug, and it was fixed by the creators of Lyo Store [18]. When using the `oslc.where` query parameter to query for a string value representing an integer, ie. a numerical value enclosed in quotes, Lyo Store generates a SPARQL query that requests the oslc.where value without the enclosing quotes. This results in the query not returning the expected resources.

Chapter 7

Evaluation

This chapter evaluates the implemented ANaConDA OSLC Adapter, and discusses the process of creating an OSLC adapter along with the suitability of OSLC for academic tools.

7.1 Evaluation of the Implemented Adapter

This section describes how the created OSLC adapter was tested, and what results were achieved. The adapter was tested in three different ways. First, the adapter is backed up by a development test suite with a test for every expected functionality. Second, the adapter was verified by an OSLC compliance test suite. And third, the adapter was deployed on a server in the Honeywell company to get feedback from a real use case scenario. Finally, we discuss an interesting observation made about an adapter modeling the OSLC Automation domain.

7.1.1 Development Test Suite

During the adapter's implementation a test suite was being created to validate new implemented functionality of the adapter. Every expected functionality of the adapter should be covered by the test suite. The test suite is implemented as a Postman collection[1] than can be executed using Postman directly or from the command line using Newman [20]. The test suite contains 61 HTTP requests with a total of 187 tests.

7.1.2 Lyo Test Suite

To verify the adapter's conformance to the OSLC specifications, we have used the Lyo Test Suite [19]. The test suite covers both OSLC domains modeled by the adapter (OSLC Core and OSLC Automation). It tests the Service Provider Catalogue, Service Providers, Resource Shapes and Basic capabilities using all three data representations (RDF, RDF+XML, JSON). The results of the test suite are:

```
Runs:  167/168(25 skipped), Errors:  0, Failures:  6
```

The six failures are related to resource update. This is not the fault of the the adapter because the test suite tests resource update for the same resource as resource creation, but

[1] oslc_adapter/tests/Tests.postman_collection.json

the adapter is not supposed to provide such functionality. The only creation capability is for Automation Requests, and the only update capability is for Automation Results. This scenario does not seem to be covered by the test suite yet.

7.1.3 Real Use Case Trial

The most useful way of getting feedback on the adapter is actually trying to use it as a part of a real software development lifecycle. The adapter was deployed in Honeywell [11] in Brno to be experimented with by integrating with their tools. Honeywell in Brno has experience using OSLC clients and servers for Automation, Configuration Management, Performance monitoring, Quality Management, and Requirements Management. They currently use OSLC Automation (Section 4.3.2) to execute test cases that satisfy formal requirements.

The Git repository of the adapter has been made public so Honeywell's employees could clone it. No further assistance with setting up or using the adapter was needed from our end, other than the documentation included with the adapter and in the OSLC interface itself.

As reported by Honeywell, the adapter was deployed with no difficulties. It is possible to integrate the adapter with other tools. The adapter is more compliant to the OSLC specification than most other OSLC adapters that are currently being used in Honeywell. And its Query capability is more complex than what is currently used in the other adapters. This is mainly caused by using Eclipse Lyo (Section 4.4) to take care of the core OSLC functionality, resource shapes, and query capabilities. Based on the results achieved so far, Honeywell is interested in further cooperation on improving the adapter in the AUFOVER project [5].

An interesting use case for OSLC adapters has been pointed out by Honeywell. The connection of the adapter with a SPARQL triplestore allows an OSLC adapter created using Eclipse Lyo to be used to populate a SPARQL database which can then be queried directly using SPARQL or by other tools other than the adapter itself.

Recently, developers of *art2kitekt* [4] started experimenting with the adapter as well. Unfortunately, no feedback other than being able to deploy the adapter was received in time for the submission of this work.

7.1.4 OSLC Automation Domain Adapter

The OSLC Automation domain's resources (see Section 4.3.2), which make up the adapter's interface, are very well suited for tools with a command line interface used in a traditional unix fashion, ie. call a command line utility with some parameters and receive an output. What we have found is that the implemented adapter for ANaConDA in its current state essentially has two things making it specific to ANaConDA, input parameter definition and path to the ANaConDA's run script. Both of these can be easily changed to make the adapter work with a different command line tool. This opens two new paths for the adapter's future development. The adapter can either be extended to accommodate multiple tools at once by adding new Automation Plans with input parameters specific to each tool, or the adapter can be reused for other tools just by modifying the existing Automation Plan.

7.2 Evaluation of OSLC

This section discusses OSLC and the process of adopting OSLC. We focused on two areas. First, how difficult it is to add OSLC support to a tool by creating an OSLC adapter. And second, what capabilities and constraints come with using an OSLC compliant API.

7.2.1 Difficulty of Adding OSLC Support

OSLC can be hard to understand for someone who is not familiar with REST, RDF, Linked Data, or Web application development in general. It is necessary to understand that OSLC represents everything as resources, including actions, which can be hard to do. The documentation provided for OSLC is currently somewhat hard to find and piece together, and there is not many user guides or forum questions because OSLC is not yet too widely spread. However, the documentation is currently being updated and reorganized on a new website, and there is a new OSLC forum with developers actively answering questions.

OSLC specification defines only a small number of obligatory requirements and the remaining requirements are optional. This means a minimalistic OSLC compliant API can only offer a small number of capabilities, and can use only simple resources with few properties making it easier to create.

The process of creating an OSLC adapter from scratch would be complicated and time consuming. Fortunately, there are many tools, SDKs, and libraries meant to make the process easier. When Eclipse Lyo Designer is used with the predefined OSLC domain models and the Lyo Store library, the process of creating an OSLC adapter gets a lot easier. Lyo Designer will take care of most work including creating the whole Web application with a RESTful API compliant with the OSLC Core specification based on the user defined adapter interface model. And adding persistent storage of resources is mostly only a question of setting up a SPARQL triplestore when the Lyo Store library is used. The tools, however, are somewhat difficult to get familiar with, and their documentation does not explain everything so the user needs to discover how they work by experimenting. Also some of the tools are still under development with limited functionality.

In summary, OSLC is somewhat hard to understand in the beginning, but it becomes straightforward once the programmer understands the core ideas. And the process of creating an OSLC adapter is well accommodated by libraries and tools, making it fairly easy to create new adapters once the programmer is familiar with the available tools. However, the documentation needs to be reorganized and updated, which will hopefully happen once more people start using OSLC.

7.2.2 OSLC API Suitability

OSLC uses predefined resources which should be used by tools to communicate. That means a tool needs to either work with the OSLC resources natively or translate its own resources to the OSLC resources. This could be potentially hard to do but should always be possible for tools that belong to one of the defined OSLC domains because if the tool uses any special resources or resource properties not covered by the domain specification, it can always extend the OSLC specification while still being OSLC compliant.

The fact that OSLC is based around resources and even actions are represented as resources, means that its fairly easy to store those resources in a database for persistent storage and database resource queries, which can be very useful for some tools.

OSLC uses self-describing APIs meaning that there is a single URI that needs to be published to users to access the API and then discover all its capabilities through the API itself. This means less documentation is required, and that the discovery of capabilities can be automated.

This work focused on the OSLC Automation domain which we found to be perfectly suited for traditional command line utilities that are used by requesting an action and receiving its result. The usage of such tool will be almost the same through the command line or through OSLC, except that OSLC uses HTTP requests.

If all tools used OSLC compliant APIs, it would be very easy to connect them together without the need to tailor each interface to a specific tool. A test management tool that uses OSLC Automation to execute tests through specialized analysis tools can easily use any analysis tool with minimal extra work required, because the communication protocol and resource semantics will always be the same as defined by the standard.

In summary, OSLC APIs seem to be very well suited and do not introduce any significant constraints. They make it much easier to connect tools together or with a persistent database. The only time when OSLC can be unsuitable is when a tool is very specific and does not belong to any of the OSLC domains, and even then the tool can still use the OSLC Core domain only and model its own specific application domain.

Chapter 8

Conclusion

This work produced a working OSLC adapter for the ANaConDA framework, allowing it to be integrated with other OSLC compliant applications. The adapter allows ANaConDA to be used as an analysis tool with persistent storage of analysis results. The adapter's compliancy to the OSLC specification was verified using an official OSLC test suite, and the adapter's functionality was successfully tested by deploying the adapter in a real industry scenario in cooperation with Honeywell as a part of the AUFOVER project. Furthermore, the adapter is currently being experimented with by the developers of art2kitekt.

Creating the adapter has demonstrated that OSLC is a viable integration option for academic tools, such as ANaConDA. The hardest part of adding OSLC support to a tool is understanding OSLC, and learning how to use the SDKs and libraries provided for adapter development. But once the programmer knows OSLC and how to use the available tools, the process of creating an OSLC adapter becomes very straightforward. The information provided in this work will hopefully be enough to serve as an introduction guide for future OSLC adapter creators, making it easier for them to get familiar with OSLC.

This work also contributed to improving Elipse Lyo by discussing a few issues encountered in it with its creators, and by discovering a bug in the Lyo Store library which was then reported and fixed. In addition, this work presented two new data race analyser implementations for ANaConDA.

The majority of time spent on this work was studying the basics of OSLC and other related concepts, understanding the OSLC specifications, getting familiar with the tools used for OSLC development, and overcoming bugs in the tools used. The time frame given for this work allowed only for a basic version of the adapter to be developed. However, the development will continue as a part of my participation in the VeriFIT research group, slowly extending the adapter's functionality based on real usage demands as they arise.

Currently, there are three known possible extensions. First, adding more Automation Plans to allow analysis of more complex programs. Second, extending the adapter's functionality to allow users to create new analysers and fully configure Anaconda. And third, a possibility of reusing or extending the adapter for other tools similar to ANaConDA.

Bibliography

[1] Aggregated Quality Assurance for Systems (H2020, ECSEL JU). https://aquas-project.eu/. accessed: 2019-05-04.

[2] Apache Jena Fuseki. https://jena.apache.org/documentation/fuseki2/. accessed: 2019-05-04.

[3] Apache Maven. https://maven.apache.org/. accessed: 2019-05-04.

[4] art2kitekt UPV. https://demos.iti.upv.es/a2k. accessed: 2019-05-21.

[5] AUFOVER - AuFoVer - Automated Formal Verification. http://www.brno.ai/research/grants/index.php?id=1259. accessed: 2019-05-04.

[6] Automated Analysis and Verification Research Group - VeriFIT. http://www.fit.vutbr.cz/research/groups/verifit/. accessed: 2019-05-04.

[7] Brno University of Technology. https://www.vutbr.cz/. accessed: 2019-05-04.

[8] Domain classes are generated at the wrong time and the wrong path. https://github.com/eclipse/lyo.designer/issues/20. accessed: 2019-05-04.

[9] Eclipse Jetty. https://www.eclipse.org/jetty/. accessed: 2019-05-04.

[10] Eclipse Lyo. https://www.eclipse.org/lyo/. accessed: 2019-03-02.

[11] Honeywell. https://www.honeywell.com/. accessed: 2019-05-21.

[12] ISSTA 2018 Tool Demonstrations - Best Tool Demo. https://conf.researchr.org/event/issta-2018/issta-2018-demos-advances-in-the-anaconda-framework-for-dynamic-analysis-and-testing-of-concurrent-c-c-programs. accessed: 2019-05-04.

[13] Linked Data. http://linkeddata.org/. accessed: 2019-03-02.

[14] Lyo Designer. https://github.com/eclipse/lyo.designer/. accessed: 2019-05-04.

[15] Lyo Domains. https://github.com/eclipse/lyo.domains/. accessed: 2019-05-04.

[16] Lyo RIO. https://github.com/eclipse/lyo.rio. accessed: 2019-05-04.

[17] Lyo Store. https://github.com/eclipse/lyo.store. accessed: 2019-05-04.

[18] Lyo Store - OSLC Query with SPARQL, issues with the where clause. https://forum.open-services.net/t/lyo-store-oslc-query-with-sparql-issues-with-the-where-clause/224. accessed: 2019-05-04.

[19] Lyo Test Suite. https://wiki.eclipse.org/Lyo/LyoTestSuite. accessed: 2019-05-04.

[20] Newman. https://www.npmjs.com/package/newman. accessed: 2019-05-04.

[21] OASIS Open Projects. https://oasis-open-projects.org/. accessed: 2019-03-02.

[22] OOSLC 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: 2019-05-04.

[23] Open Services for Lifecycle Collaboration. https://open-services.net/. accessed: 2019-03-02.

[24] Open Services for Lifecycle Collaboration Core Specification Version 2.0. https://archive.open-services.net/bin/view/Main/OslcCoreSpecification.html. edited by John Arwe. 30 May 2013. Accessed: 2019-03-02.

[25] 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: 2019-03-02.

[26] OSLC Forum. https://forum.open-services.net/. accessed: 2019-05-04.

[27] OSLC Specifications. https://open-services.net/specifications/. accessed: 2019-03-02.

[28] OslcCoreMissingSetMethodException due to a regression in LyoD. https://github.com/eclipse/lyo.designer/issues/61. accessed: 2019-05-04.

[29] Postman. https://www.getpostman.com/. accessed: 2019-05-04.

[30] Resource Description Framework (RDF). https://www.w3.org/RDF/. accessed: 2019-03-02.

[31] Setup an OSLC Provider/Consumer Application. http://oslc.github.io/developing-oslc-applications/eclipse_lyo/setup-an-oslc-provider-consumer-application.html. accessed: 2019-05-04.

[32] Toolchain Modelling Workshop. http://oslc.github.io/developing-oslc-applications/eclipse_lyo/toolchain-modelling-workshop.html. accessed: 2019-05-04.

[33] Triplestore. https://en.wikipedia.org/wiki/Triplestore. accessed: 2019-05-04.

[34] Berners-Lee, T.; Fielding, R.; Masinter, L.: RFC 3986, Uniform Resource Identifier (URI): Generic Syntax. Request For Comments (RFC). 2005. doi:10.17487/RFC3986. Retrieved from: http://www.ietf.org/rfc/rfc3986.txt

[35] Carothers, G.; Prud'hommeaux, E.; Beckett, D.; et al.: RDF 1.1 Turtle. 2014. Retrieved from: http://www.w3.org/TR/2014/REC-turtle-20140225/

[36] Fiedor, J.; Hrubá, V.; Krena, B.; et al.: Advances in noise-based testing of concurrent software. *Software Testing, Verification and Reliability*. vol. 25. 09 2014. doi:10.1002/stvr.1546.

[37] Fiedor, J.; Mužikovská, M.; Smrčka, A.; et al.: Advances in the ANaConDA framework for dynamic analysis and testing of concurrent C/C++ programs. 07 2018. pp. 356–359. doi:10.1145/3213846.3229505.

[38] Fiedor, J.; Vojnar, T.: Noise-based testing and analysis of multi-threaded C/C++ programs on the binary level. *2012 10th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, PADTAD 2012 - Proceedings*. 07 2012. doi:10.1145/2338967.2336813.

[39] Fiedor, J.; Vojnar, T.: Noise-based testing and analysis of multi-threaded C/C++ programs on the binary level. 2012. doi:10.1145/2338967.2336813.

[40] Fiedor, J.; Vojnar, T.: ANaConDA: A Framework for Analysing Multi-threaded C/C++ Programs on the Binary Level. 01 2013. doi:10.1007/978-3-642-35632-2_5.

[41] Fielding, R. T.: *REST: Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation. University of California, Irvine. 2000. Retrieved from: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[42] Flanagan, C.; Freund, S.: The FastTrack2 Race Detector. 2017. Retrieved from: http://dept.cs.williams.edu/~freund/papers/ft2-techreport.pdf

[43] Flanagan, C.; N. Freund, S.: FastTrack. 2009. doi:10.1145/1543135.1542490.

[44] Harris, S.; Seaborne, A.: SPARQL 1.1 Query Language. 2013. Retrieved from: https://www.w3.org/TR/2013/REC-sparql11-query-20130321/

[45] Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*. vol. 21. 07 1978: pp. 558–565. doi:10.1145/359545.359563.

[46] Liao, C.; Lin, P.-H.; Asplund, J.; et al.: DataRaceBench: A Benchmark Suite for Systematic Evaluation of Data Race Detection Tools. SC'17. ACM. 2017.

[47] Luk, C.-K.; S. Cohn, R.; Muth, R.; et al.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. 01 2005. pp. 190–200.

[48] Mattern, F.: Virtual Time and Global States of Distributed Systems. 1988. Retrieved from: https://www.vs.inf.ethz.ch/publ/papers/VirtTimeGlobStates.pdf

[49] Pozniansky, E.; Schuster, A.: Efficient on-the-fly data race detection in multithreaded C++ programs. ACM. 10 2003. pp. 178–189. doi:10.1145/781527.781529.

[50] Ryman, A.: Open Services for Lifecycle Collaboration Core Specification Version 2.0 Query Syntax. https://archive.open-services.net/bin/view/Main/OSLCCoreSpecQuery. accessed: 2019-05-04.

[51] Savage, S.; Burrows, M.; Nelson, G.; et al.: Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. 1997. doi:10.1145/269005.266641.

[52] Vašíček, O.: Rozvoj prostředí ANaConDA pro testování a dynamickou analýzu paralelních programů. FIT VUT v Brně. 2018. projektová praxe, Brno.

[53] Vašíček, O.: Rozvoj prostředí ANaConDA pro testování a dynamickou analýzu paralelních programů. FIT VUT v Brně. 2018. projektová praxe, Brno.

Appendix A

OSLC Adapter Usage Guide

This chapter gives instructions on how to use the ANaConDA OSLC Adapter, and is meant for readers who have no prior experience with using a RESTful interface. During the development of the adapter we have used the Postman REST client [29] to create, send, and receive HTTP requests.

A.1 Adapter Configuration

The adapter needs to be configured to match the specific deployment environment. There are two configuration files for the adapter: the project `pom.xml` file and Java properties file `AnacondaAdapter.properties`. The only two values configured in the POM file are `host address` and `host port` for the adapter to run on. Then in the properties file the host's port and address needs to be configured again[1], along will all the remaining configuration properties: ANaConDA path, ANaConDA available analysers, and SPARQL endpoint URIs.

To use the Apache Jena Fuseki SPARQL triplestore, which is bundled with the adapter as a WAR file inside of a Jetty distribution[2], the Jetty container needs to be configured in order to change the server port and host. This can be done by changing the Jetty container's `start.ini` file. To create or otherwise manage datasets, use the Fuseki's Web UI. A dataset needs to be created in order to use the adapter. Fuseki offers three types of datasets - `in memory`, `persistent`, and `persistent (TDB2)`. The adapter works with the first two options (ie. `persistent (TDB2)` is not supported).

A.2 Using a REST Client

The intended way of using the Adapter is by sending HTTP requests to the REST interface endpoints. When sending GET requests (resource read or query) make sure to use the `Accept` header to choose what data representation the adapter should use in its response, options are RDF/XML, XML, or JSON. To explore the adapters capabilities send a GET request to the service provider catalogue (`/AnacondaOSLC/services/catalog/singleton`).

[1] Unfortunately, I did not manage to get the maven-properties plugin working, which means the adapter cant import Java properties into the POM file. That is why the host address and port needs to be configured twice.

[2] A complete distribution of Jetty is included with the adapter to allow users to just clone the adapter's repository, and run it immediately without needing to download additional software.

The adapter will return a list of all service providers and their URIs. Send a GET request to the URI of the Automation service provider, and the adapter will return a list of all the capabilities of the service provider and their URIs. Before creating an Automation Request you need to look at what properties does an Automation Request have. This can be achieved by GETing the Automation Request resource shape which has its own URI in the list of the service provider's capabilities.

To create an Automation Request send a POST request to the Automation Request creation factory. The body of the POST request needs to contain a valid representation of an Automation Request according to the resource shape. The representation can be in RDF/XML, XML, or JSON. The adapter will return meaningful error responses with a return code 400 (bad request) in case the POST request is not valid. If the creation is successful, the adapters response will contain the newly created resource with its new URI, and the response code will be 201 (created). The created resource can now be read by sending a GET request to its URI with the Accept header set. The response code will be 200 (OK), and the body will contain the resource if the resource exists. If the resource does not exist the response code will be 404 (not found).

In order to update a resource, given the resource has an update capability, you need to send a PUT request to the resources URI with the new resource representation in its body. The recommended way of doing this is by GETing the resource, modifying the body of the adapter's response, and then sending the modified body in the PUT request. The adapter's response will have the code 200 (OK) if the update was successful. The only resource that supports updating is the Automation Result.

To delete a resource send a DELETE request to its URI. If the deletion was successful the adapter's response will have the code 200 (OK). The only resource that supports deleting is the Automation Result.

To query for resources, find the URI of the query capability for a specific resource type, and send a GET request to it. The adapter will return a list of resources, and the response will contain an `oslc:totalCount` property which says how many items are in the list. The adapter will return resources in pages of 20 resources; the page size can be altered using the `limit` parameter, and different pages can be requested using the `page` parameter. The page parameter sets the amount of resources that should be skipped in the result. You can query for specific resources by adding parameters to the GET request according to the OSLC Query syntax. Currently, the adapter supports a fulltext search `searchTerms` parameter and a `where` parameter for property equality. The searchTerms parameter filters resources based on values of all their properties and its syntax is `oslc.searchTerms=textToFind`. The `where` parameter is used to find specific property-value pairs and it currently only supports testing equality of value types INT, URL, or String. It needs to be used with an `oslc.prefix` parameter which defines domain prefixes used in the `where` parameter. The prefix syntax is `oslc.prefix=namespace_name=<namespace_url>` and the `where` syntax is `oslc.where=namespace_name:property_name=property_value`.

A.3 Adapter's Web UI

The adapter does have a basic placeholder Web UI that was generated by Lyo Designer. This work did not focus on creating the Web UI, and thus it is far from finished. It is possible that the current UI will be used as a base for creating a proper UI in the future. The UI currently has the following functionality: discovering capabilities through

the service provider catalogue, reading resources that do not have inlined resources as properties, querying for resources without query parameters, and viewing resource shapes.

Appendix B

Further Advancements of ANaConDA

Apart from creating the OSLC Adapter, I have contributed to ANaConDA by implementing two data race detectors in C++ based on academic tool papers. Work on the anlaysers was not done during the official thesis time frame but rather throughout my whole study as a lead up to the thesis itself during my participation in the VeriFIT research group. This chapter summarizes the most important parts in a minimalistic manner. For full reports see [52] and [53] (written in Czech, not English).

B.1 Refactoring of Eraser

Eraser [51] was already implemented for ANaConDA but the implementation was not properly tested, so it was decided to refactor the analyser by creating a new implementation. Eraser is a simple data race detector that is based on the idea that each variable should be consistently protected by at least one lock. This assumption often holds, especially for simple programs, but more complex programs often use more complex synchronization. This makes Eraser an analyser that typically reports a high number of *false positives*[1].

B.1.1 Basic Algorithm

The very basic version of the algorithm maintains a set of locks, *lockset*, for each variable. The lockset is initialized to contain all the locks used by the analyzed program. Every time a thread accesses a variable the lockset of the variable gets intersected with the set of locks currently held by the accessing thread. When the lockset of a variable is empty, the analyser reports a warning. Figure B.1 summarizes the above described algorithm.

B.1.2 Algorithm Extensions

The basic algorithm is further extended to reduce the amount of false positives by adding states to variables. The extension tries to accommodate read-shared variables and the process of initializing shared variables. All variables start in the *Virgin* state. On the first write (ie. initialization) by a thread the variable goes into the *Exclusive* state which represents that the variable has so far been accessed only by a single thread (ie. is not

[1] Reporting a data race where there actually is none

Let *locks_held(t)* be the set of locks held by thread t.
For each v, initialize $C(v)$ to the set of all locks.
On each access to v by thread t,
set $C(v) := C(v) \cap locks\_held(t)$;
if $C(v) = \{\}$, then issue a warning.

Figure B.1: Eraser basic algorithm [51]

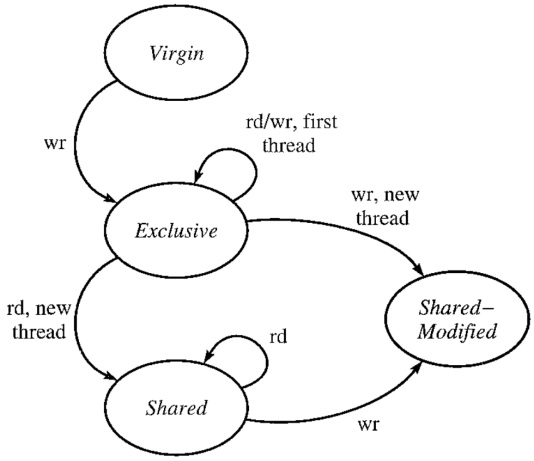

Figure B.2: State transfer graph of the extended Eraser algorithm. Source: [51]

shared yet). While in the *Exclusive* state, a variable's lockset does not get updated. The state changes only when a new thread accesses the variable. Once the variable is not in the *Exclusive* state its lockset is updated with every access. Depending on the type of access, the variable's state changes to either *Shared* or *Shared-Modified*. The *Shared* state represents read-shared variables. If the variable's lockset gets emptied while in the *Shared* state, then no warnings are issued by the analyser. The *Shared-Modified* state is the only state in which warning are issued for emptied locksets. It represents write-shared variables, and the variable enters this state either after any write in the *Shared* state or after a write by a thread other then the initializing one in the *Exclusive* state. Figure B.2 shows the states and state transfers described above.

B.2 Implementation of FastTrack2

FastTrack2 [42] is a precise and relatively fast dynamic data race detector that uses vector clocks [48] and the happens-before relation [45]. A *vector clock* can be seen as a vector of timestamps. Each timestamp represents the point of last contact with a thread (index of the timestamp in the vector corresponds to the *ID* of the thread). The *happens-before* relation is a relation between program events that represents their ordering based on other program events, as opposed to real time ordering. Two program events are ordered by the happens-before relation if at least one of these three conditions holds: both events were performed by the same thread, both events lock or unlock the same lock, or one of the

events is *fork\join* and the other is performed by the *forked\joined* thread. FastTrack 2 is an improved version of *FastTrack* [43] which is based on *DJIT+* [49]. The core idea of FastTrack is to take DJIT+ and make it faster. DJIT+ maintains a full vector clock for every thread, every lock, and every variable. In a program that has n threads every vector clock will take up $O(n)$ memory space, and operations with the vector clock will take $O(n)$ time. FastTrack leverages the fact that all writes or reads to a variable are ordered by the happens-before relation, as long as a data race has not occurred yet, and replaces vector clocks of most variables with epochs. *Epoch* is a tuple of a thread ID and a timestamp, and thus it takes up only $O(1)$ memory space and operations with it take only $O(1)$ time.

B.2.1 Algorithm

FastTrack maintains a full vector clock for every thread and every lock. And in most situations it maintains two epochs for each variable, one for the last read access and one for the last write access. The only exception is variables that are read-shared for which a full vector clock is needed. The time stored in vector clocks advances on lock unlocks and thread forks. The monitored program events are processed as follows (vector clock referred to as *vc*, *advancing a threads time* means incrementing the value of its vc at the same index as the thread's ID, *updating vc A with vc B* means setting all A[idx] to the maximum out of A[idx] and B[idx]):

- Unlock: Replace the lock's vc with the thread's vc, then advance the thread's time.
- Lock: Update the thread's vc with the lock's vc.
- Fork: Update the forkee's vc with the forker's vc, then advance the forker's time.
- Join: Update the joiner's vc with the joinee's vc.
- Write: Replace the write epoch with the time and ID of the accessing thread
- Read: Replace the read epoch with the time and ID of the accessing thread

On every memory access FastTrack checks whether the happens-before relation still holds by comparing the epochs of the variable with the vector clock of the thread. The happens-before relation holds if the timestamp in the epoch is *smaller than or equal* to the vector clock's value at the corresponding index. If the happens-before relation is violated, then a data race is reported if at least one of the unordered accesses is a write. If the happens-before relation is violated by two read accesses, then the variable becomes read-shared and FastTrack will maintain a full vector clock for it. Figure B.3 shows two examples of vector clocks and epochs changing during program analysis. The left example shows the effect of locking, unlocking, and writing to a variable. The right example shows the effect of fork, join, and read-shared variables.

B.3 Analyser Evaluation

Both new implementations were initially tested on a set of 14 student projects, implementing synchronization using monitors, which were used in PADTAD [39]. FastTrack2 was further tested on a data race benchmark suite DataRaceBench [46], and compared with three other data race analysers in the ANaConDA tool paper published at ISSTA2018 [37].

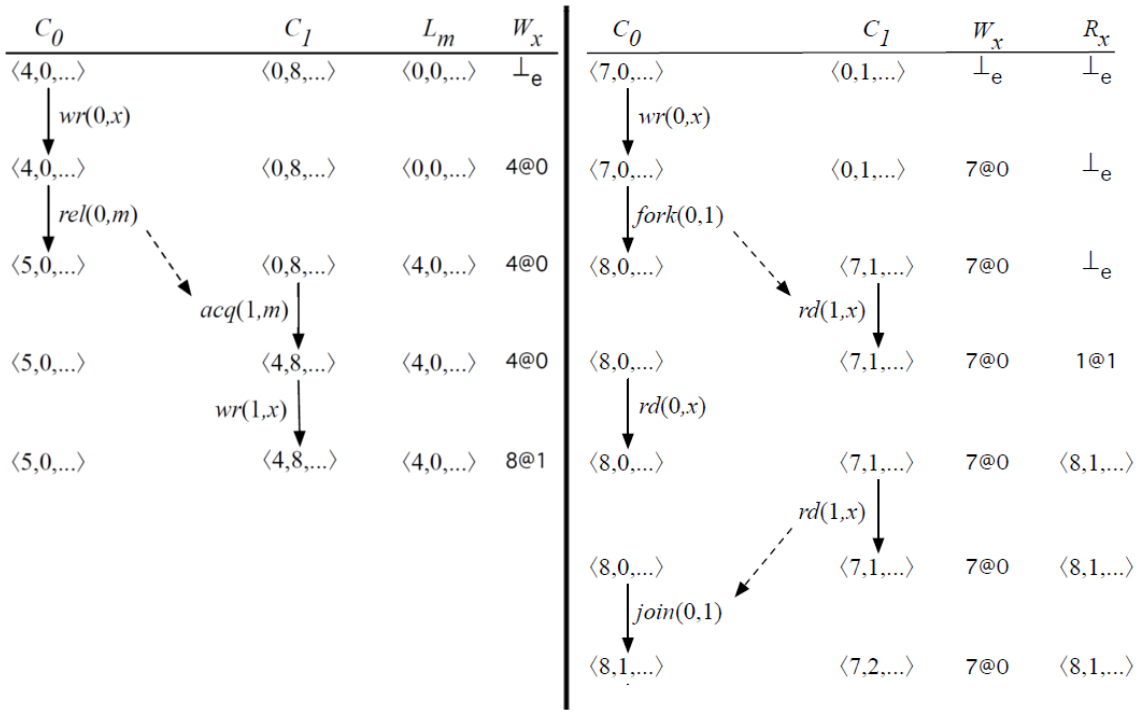

Figure B.3: Two examples of vector clocks and epochs during program analysis. Symbols: C thread, L lock, W write epoch of a variable, R read epoch of a variable. Source: [43]