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 SOFTWARE ANALYSIS

ADAPTÉR OSLC PRO ANALÝZU SOFTWARU

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. ONDŘEJ VAŠÍČEK**
AUTOR PRÁCE

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

BRNO 2021

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2020/2021

Master's Thesis Specification

Student: **Vašíček Ondřej, Bc.**
Programme: Information Technology and Artificial Intelligence
Specialization: Software Verification and Testing
Title: **OSLC Adapter for Software Analysis**
Category: Software analysis and testing

Assignment:

1. Study OSLC standard, in particular Automation domain defined by OSLC.
2. Analyse the requirements for tools for automatic analysis of software. Design an OSLC adapter for integration of static and dynamic analysers of software.
3. Implement the adapter design in the previous step.
4. Verify the basic functionality of the adapter using automated tests. Demonstrate the usefulness of the adapter on different analysers of software (e.g. ANaConDA, Perun, Valgrind, pylint).

Recommended literature:

- OASIS Standardisation Group. Open Services for Lifecycle Collaboration. https://open-services.net/
- 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. In: *Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York: Association for Computing Machinery, 2018, s. 356-359. ISBN 978-1-4503-5699-2.

Requirements for the semestral defence:

- The first two steps.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Smrčka Aleš, Ing., Ph.D.**
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2020
Submission deadline: May 19, 2021
Approval date: January 15, 2021

Abstract

The goal of this work is to provide an easy way of adding an OSLC compliant interface to an analysis tool. Such an interface allows tools to be easily integrated with other tools or systems, allows them to be used remotely due to its web based nature, and allows them to be easily connected with a database for persistency and queries. This is achieved by designing and creating an OSLC adapter using Eclipse Lyo that is universal enough to accommodate the functionality of most analysis tools. This is done by using the OSLC Automation domain interface and by leveraging the current command-line interfaces of analysis tools. This work provides an introduction to OSLC, Eclipse Lyo, and other related topics; defines requirements and differences of analysis tools; covers the design process of the adapter and the factors that impacted design decisions; and finally, presents the implemented adapter and evaluates it by using an automated test suite and then experiments with a set of different analysis tools. The most important evaluation indicator is that the current version of the adapter is already being used in practice to add an OSLC interface to four analysis tools: ANaConDA, Perun, Spectra (all three developed by VeriFIT); and HiLiTE (Honeywell).

Abstrakt

Cílem této práce je poskytnout snadný způsob, jak rozšířit analyzační nástroj o rozhraní splňující standard OSLC. Takové rozhraní umožňuje jednoduchou integraci nástrojů s jinými nástroji nebo systémy, umožňuje jejich vzdálené použití skrze webové služby a umožňuje je jednoduše propojit s databází pro databázové dotazy a pro perzistentní uložení dat. Toto je dosaženo návrhem a implementací OSLC adaptéru pomocí sady nástrojů Eclipse Lyo. Adaptér používá jako rozhraní doménu OSLC Automation a je dostatečně univerzální na to, aby skrze toto rozhraní pokryl funkcionalitu většiny analyzačních nástrojů za pomocí jejich stávajících rozhraní na příkazové řádce. Tato práce poskytuje úvod k OSLC, Eclipse Lyo a souvisejícím konceptům. Dále tato práce definuje požadavky a odlišnosti různých analyzačních nástrojů a diskutuje návrh adaptéru a faktory, které ovlivnily návrhová rozhodnutí. A nakonec prezentuje implementovaný adaptér a jeho vyhodnocení pomocí automatizované testovací sady a pomocí experimentů s řadou analyzačních nástrojů. Nejvýznamnější ukazatel hodnocení vytvořeného adaptéru je to, že už teď je používán v praxi pro přidání OSLC rozhraní k nástrojům ANaConDA, Perun, Spectra (všechny tři vyvíjené na VeriFIT) a HiLiTE (Honeywell).

Keywords

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

Klíčová slova

OSLC, OSLC adaptér, OSLC producent, OSLC server, OSLC konzument, OSLC klient, OSLC Automation, Eclipse Lyo, integrace nástrojů, verifikace a analýza software, ANaConDA, Facebook Infer, Perun

Reference

VAŠÍČEK, Ondřej. *OSLC Adapter for Software Analysis.* Brno, 2021. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Aleš Smrčka, Ing., Ph.D.

Rozšířený abstrakt

Analyzační nástroje pro verifikaci a validaci programů (tzv. analyzátory) jsou většinou vyvíjeny s vlastním jednoúčelovým rozhraním specifickým pro funkcionalitu daného nástroje. Méně pozornosti bývá často věnováno rychlé učící křivce použití analyzátoru a jeho integraci s jinými nástroji nebo systémy. Důsledkem je náročná integrace analyzátorů, která vyžaduje individuální přístup pro každý nástroj. Tento problém pomáhají řešit standardní rozhraní, které umožnují výrazně snazší integraci nástrojů podporující stejný standard. OSLC (Open Services for Lifecycle Collaboration) je jeden z těchto standardů, který se začíná rozšiřovat a být využíván jak ve výzkumných projektech, tak i v průmyslu [9].

Cílem této práce je poskytnout snadný způsob, jak rozšířit (nejen) analyzační nástroje o rozhraní splňující standard OSLC. Takové rozhraní umožňuje jednoduchou integraci nástrojů s jinými nástroji nebo systémy, umožňuje jejich vzdálené použití skrze webové služby a umožňuje je jednoduše propojit s databází pro databázové dotazy a pro perzistentní uložení dat. Podporuje tedy lepší automatizaci, integraci, udržitelnost a případně také přenostitelnost. Toto je dosaženo návrhem a implementací OSLC adaptéru, který je konfigurovatelný tak, aby byl schopen poskytnout funkcionalitu většiny analyzačních nástrojů za pomocí jejich stávajících rozhraní na příkazové řádce. Adaptér byl vytvořen za pomocí sady nástrojů Eclipse Lyo, jejímž cílem je usnadnit vytváření OSLC serverů a klientů. Model adaptéru byl vytvořen pomocí nástroje Lyo Designer a základ jeho kódu vygenerován pomocí nástroje Lyo Code Generator a knihovny OSLC4J s využitím referenčních modelů standardních domén v Lyo Domains a knihovny pro komunikaci s databází Lyo Store.

Rozhraní adaptéru používá standardní doménu OSLC Automation, jejíž hlavní případy užití jsou spouštění testů, nasazování softwaru a kompilace softwaru. Hlavní zdroje tvořící toto rozhraní jsou Automatizační Plány, Automatizační Požadavky a Automatizační Výsledky. Automatizační Plány reprezentují jednotky automatizace poskytované OSLC serverem. Automatizační Požadavky jsou to, co klienti vytváří, když chtějí požádat a provedení některé z nabízených jednotek automatizace. Automatizační Výsledky reprezentují výsledek provedení jednotky automatizace.

Aby mohl být adaptér řádně navržen, bylo nutné nejdříve definovat požadavky různých analyzačních nástrojů za pomocí experimentů se zástupci dynamických i statických analyzačních nástrojů. Požadavky obsahují kroky, které uživatel provádí s analyzačním nástrojem před spuštěním analýzy, při spuštění analýzy, v průběhu analýzy a po analýze. Návrh adaptéru a jeho funkcionality byl pak řízen tak, aby splnil všechny definované požadavky a byl tak použitelný pro co nejvíce analyzačních nástrojů.

Adaptér byl rozdělen na dva pod-adaptéry pro oddělení dvou odlišných případů užití potřebných pro použití analyzačního nástroje. Prvním pod-adaptérem je adaptér kompilační, který se stará o přenos souborů SUT[1] na analyzační server, o jejich správu a o jejich kompilaci. Druhým a komplexnějším pod-adaptérem je adaptér analyzační, který se stará o spouštění analýzy a o správu výsledků. I když oba pod-adaptéry musí sdílet část souborového systému pro přenos výsledků kompilace pro analýzu, mohou běžet v odděleném prostředí, a tedy mít snížené nároky na údržbu a aktualizace.

Analyzační pod-adaptér je konfigurovatelný tak, aby byl schopen poskytovat funkcionalitu většiny analyzačních nástrojů. Konfigurace adaptéru pro použití konkrétního analyzačního nástroje se provádí vytvořením Automatizačního Plánu pro daný nástroj, který defin-

[1] System Under Test

uje vstupní parametry analyzačního nástroje a zároveň říká adaptéru, jak analyzační nástroj spouštět. Adaptér po přijetí Automatizačního Požadavku pro spuštění analýzy od klienta, který obsahuje vstupní parametry analýzy, zpracuje vstupní parametry do podoby řetězce, který je pak exekuován pomocí nativního terminálu analyzačního serveru tak, aby byly všechny parametry specifikované klientem provedeny stejně jako na příkazové řádce.

Adaptér obsahuje parametry pro kontrolu běhu analýzy, jako například časový limit běhu, parametry pro vytváření konfiguračních souborů pro běh analyzačního nástroje, parametry pro nastavení proměnných prostředí nebo jinou přípravu před nebo po běhu analýzy a parametry pro kontrolu tvorby výstupů analýzy. Výsledky analýzy jsou složeny ze standardních výstupů, vytvořených nebo modifikovaných souborů a ze sémanticky vyšších výstupů analyzačního nástroje, jako je například detekce chyby. Standardní výstupy jsou vždy součásti Automatizačních Výsledků produkovaných analýzou. Soubory vytvořené nebo modifikované analýzou mohou být přidány jako výsledek analýzy pomocí regulárního výrazu nad jejich relativní cestou a jménem. Vyšší výstupy analyzačních nástrojů mohou být vytvořeny výstupními filtry, které mohou uživatelé vytvářet pomocí plug-in systému adaptéru. Takový filtr má na vstupu veškeré stávající výstupy analýzy a může je libovolně upravovat, prohledávat nebo upravovat jejich obsah a nebo vytvářet nové výstupy se specifickým významem. Adaptér je dále propojený s databází, která umožňuje perzistentní uložení všech zdrojů a poskytuje funkcionalitu databázových dotazů nad zdroji.

Vytvořený adaptér v našem testování pokrývá všechny požadavky analyzačních nástrojů definované v této práci a byl otestován pomocí automatizované testovací sady na platformách Linux i Windows. Byly provedeny manuální experimenty s použitím adaptéru s analyzačními nástroji ANaConDA, Facebook Infer, Perun, Valgrid, a Grep[2] pro ověření jejich použitelnosti. Adaptér je již používán pro nástroje ANaConDA, Perun a Spectra v rámci výzkumné skupiny VeriFIT. Dále jeden z výzkumníku z této skupiny vyvíjí plug-in pro vývojové prostředí Eclipse, který bude adaptér používat pro spouštění analýzy přímo z vývojového prostředí. A nakonec je adaptér aktuálně využíván ve firmě Honeywell v kombinaci s jejich nástrojem HiLiTE. Honeywell byl zdrojem několika užitečných podnětů k vylepšení adaptéru a oceňuje jednoduchou rozšiřitelnost a konfigurovatelnost adaptéru.

Možnosti dalších vylepšení adaptéru jsou například komplexnější systém pro řazení požadavků do front s možností prioritních požadavků, rozšířené možnosti autentizace a bezpečnostní aspekty použití adaptéru v nepřátelském prostředí. Nápady pro další práci v této oblasti jsou například vytvoření koordinačního adaptéru[3], který by agregoval několik analyzačních serverů s běžícími analyzačními adaptéry a distribuoval mezi ně požadavky na základě dostupných analyzačních nástrojů nebo vyvažování zátěže; a nebo vytvoření uživatelského rozhraní pro adaptér pomocí samostatné webové aplikace, která by poskytla přívětivé rozhraní pro lidské klienty (oproti strojovým klientům).

[2]Grep lze považovat za nejjednodušší možnou formu statické analýzy. Zároveň ukazuje použitelnost adaptéru s Unixovými utilitami.

[3]*master* adaptéru

OSLC Adapter for Software Analysis

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Aleš Smrčka, Ing., Ph.D. Supplementary information was provided by Jan Fiedor, Ing., Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Ondřej Vašíček
May 12, 2021

Acknowledgements

I would like to thank my supervisor Aleš Smrčka, Ing., Ph.D. and my main consultant Jan Fiedor, Ing., Ph.D. Additional help and consulting was provided by Prof. Tomáš Vojnar, Ing., Ph.D.; Bohuslav Křena, Ing., Ph.D.; and Mgr. Tomáš Kratochvíla. This work was supported by the TAČR AuFoVer project[4] and the H2020 ECSEL Arrowhead Tools project[5]. A special thanks to my girlfriend for proofreading this work, and for her unwavering support and plentiful encouragements.

[4]TAČR AuFoVer - https://www.vutbr.cz/en/rad/projects/detail/29833
[5]H2020 ECSEL Arrowhead Tools - https://www.fit.vut.cz/research/project/1299/.en

Contents

Chapter 1

Introduction

This chapter briefly discusses the motivation and the objectives of this work, the approach taken to achieve these objectives, and summarizes the results of this work.

1.1 Motivation and Objectives

In my bachelor's thesis [40] an OSLC adapter for the ANaConDA framework [13][11] was created. The previously created adapter was a useful addition that enabled ANaConDA to be integrated and then used in cooperation with Honeywell. Furthermore, OSLC support is a valued feature in international research projects like AuFoVer [5] and Arrowhead Tools [4]. Therefore, it was decided that similar OSLC adapters would be useful for other tools developed at our university. The original OSLC adapter was modified to work with a different analysis tool, Perun [14], in my Project Practice [41] to get more experience working with OSLC and to learn about the differences between adapters for different tools. The results showed that there was very little difference in the two adapters, because both tools are used through a command-line interface and their main differences are command-line parameters, which essentially boils down to executing a different string when running the tool. This led to the idea of creating one adapter that is universally usable for all analysis tools in that it adapts the OSLC interface to a command-line interface while providing features needed by analysis tools. Analysis tools can then have different command-line parameters and different analysis outputs which can be processed differently through the adapter's configuration.

A standardized OSLC interface is useful for an analysis tool because it allows the tool to be easily integrated with other systems which support the same standard. This makes the tool more likely to be adopted by new users and adds new potential use cases. OSLC, specifically, was chosen because it is a long-running open project with inputs from the industry itself, and lately it is becoming popular and adopted by a growing number of systems, such as IBM Rational DOORS or Atlassian JIRA (See [9] for a full list). Other advantages that come with an OSLC interface are an easy way of adding persistency and a web based interface which can then be used remotely. A web interface can be very useful for analysis tools since it allows them to be offered to users as services, and also because it allows tools to be setup and deployed at a server for users to utilize instead of forcing users to set up the tool on their own machines.

The objective of this work is to create an adapter that provides an OSLC interface for any command-line analysis tool without the need to modify the tool nor the adapter, with the exception of necessary configuration. The targeted group of users for the adapter are analysis tool developers who want to add an OSLC interface to their tool, or tool integration developers who want to integrate an analysis tool with their system using OSLC. The adapter is not meant to be used directly by human clients to run analysis since it only has a REST interface which is more suitable for machine clients (who can then provide a user friendly interface).

1.2 Approach and Contribution of the Thesis

The way to add an OSLC interface to a tool that is natively not a web application is by creating an adapter which implements one of the OSLC domains. This is the case for all the analysis tools considered in this work. Therefore, an OSLC adapter[1], was designed and implemented using the OSLC Automation domain, which is meant for use cases, such as test execution, deployment, or compilation. The adapter itself is actually a toolchain of two sub-adapters in order to separate two distinct functionalities required to use an analysis tool. The first one is the Compilation sub-adapter[2] for transferring the SUT to be analyzed to the analysis server and compiling it. And the second one is the Analysis sub-adapter[3] for executing analysis on previously prepared SUTs and managing its results. The Analysis sub-adapter leverages command-line interfaces of the adapted analysis tool and is configurable to be able to accommodate as wide a range of analysis tools as possible.

In order to design the adapter properly, the usage requirements, similarities, and differences of various analysis tools needed to be examined. This provided an understanding of what functionality needs to be provided by the universal adapter. Tools used as representatives for the requirements study and testing include ANaConDA [11], Facebook Infer [10], Valgrind [39], Perun [14], grep, HiLiTE, and Spectra [36].

The adapter was designed and implemented using Eclipse Lyo which is a set of tools and resources that supports OSLC adapter creation. Namely, Lyo Designer and Code Generator were used to model the adapter's domains and capabilities and to subsequently generate the code skeleton of the adapter. The code skeleton consists of OSLC compliant interface API endpoints and classes for domain resources. Reference domain models were used from Lyo Domains, and Lyo Store was used for database communication. The code skeleton was then filled with all the application logic needed for the adapter's required functionality.

1.3 Result

The adapter's functionality was verified by an automated system test suite and by experimental integration of the tools considered in the requirements study. The implemented adapter was found to sufficiently provide all main functionalities of all the tested tools in our experiments. The adapter also works on both Linux and Windows and is connected to a database to provide persistent storage of resources and query capabilities. More importantly, the adapter is already being used in practice in four use cases to provide an OSLC interface for ANaConDA, Perun, Spectra (all three developed by VeriFIT); and HiLiTE

[1] referred to as the OSLC Universal Analysis Adapter, *the OSLC adapter*, or *the adapter*
[2] referred to as the *Compilation adapter* or the *Compilation sub-adapter*
[3] referred to as the *Analysis adapter* or the *Analysis sub-adapter*

(Honeywell). Furthermore, a VeriFit researcher is developing an Eclipse plugin which plans to use the OSLC Universal Analysis Adapter to allow analyses to be executed directly from the Eclipse IDE.

1.4 Document Structure

This work consists of eight chapters. Chapter 1 introduces the motivation, objectives, proposed solutions and results of this work. Chapter 2 briefly covers general areas that this work relates to and presents the current state of things at the time of making this work. Chapter 3 provides a basic introduction to topics that needed to be studied for this work - OSLC and Eclipse Lyo (first point of the assignment). Chapter 4 defines analysis tool requirements which are then used to design capabilities of the OSLC Universal Analysis Adapter in Chapter 5 (second point of the assignment). Chapter 5 covers the adapter's architecture and design using the models created with Eclipse Lyo Designer, and provides a design perspective on accommodating analysis tool requirements defined in Chapter 4 (second point of the assignment). Chapter 6 describes important parts of the adapter's implementation and its features (third point of the assignment). Chapter 7 contains experiments including three case studies and the adapter's evaluation (fourth point of the assignment). Chapter 8 concludes this work.

Appendix A lists URLs for a public repository containing the adapter implemented in this work and parts of its Wiki pages. Appendix B shows a detailed example of running analysis using ANaConDA, including full sized XML files for completeness. Appendix C contains a configuration file for GitLab CI which can be used as inspiration for running a docker container to test the adapter.

Chapter 2

Background

This chapter introduces broad contexts for this work, including a summary of the current state of things at the time of working on this thesis.

2.1 Software Analysis and Verification

Various analysis algorithms were created and are being used to make the process of error tracking and identification easier. Different types of analysers use different approaches to detect errors, focus on different kinds of errors, and have different characteristics depending on the algorithm they use. An ideal analyser would have good performance and give sound, accurate results. To be considered accurate, an analyser needs to avoid reporting *false negatives* (not reporting an error that is present in the analysed program) and *false positives* (reporting an error that is not present in the analysed program). There are two main kinds of analysers: static and dynamic.

Static analysers use formal methods to analyse a program's source code without executing it. They aim to be sound and thus report no false negatives, i.e. if the analyser reports no errors then there really are no errors of that kind in the analysed program. However, current static analysers are plagued by reporting a large number of false positives and by having unusable performance while analysing real life systems. As a result, they often have to sacrifice some soundness to be usable.

Dynamic analysers, on the other hand, work by observing events in a program's execution. They are typically better at reporting no false positives since they can only report an error if they see it happen in the execution trace of the analysed program. However, dynamic analysis introduces significant overhead to the execution of the analysed program, and for an error to be detected it might be required to analyse a large number of program executions. To get better performance or to detect errors more reliably, analysers can use extrapolation which can sometimes lead to false positives.

2.2 Tool Integration

Integrating tools can make their use much more convenient and user friendly. For example, having email integration in a bug tracking system can automatically distribute notifications to programmers who need to fix them. Or, integrating a bug tracking system into an enterprise solution that, among others, contains a change management system that allows bugs to be easily linked with the changes that resolved them or caused them. Without

a standardized interface, the only way of integrating existing applications is by creating an adapter on a *tool to tool* basis. This makes it quite difficult and time-consuming to integrate tools because the adapter's creator needs to have good knowledge of the implementation of both tools being integrated, and the programmer needs to implement a specific adapter for every integration case, possibly even making changes to one of the tools in the process. Should the programmer decide to switch out the integrated tool with a different one, a whole new adapter would need to be created specifically for the new tool. To avoid creating adapters on a tool to tool basis, there needs to be a standardized interface. With a standardized interface, the only adapter a tool needs is the *standard interface adapter*. The tool can even use the standard interface natively without the need for an adapter. There should be no extra work required to integrate tools through the standardized interface in the ideal case. However, creating a perfect standard interface is not possible due to tools having different functionalities and input parameters. A feasible, useful approach is to define a standard communication protocol and information representation, leaving the actual contents of the interface extendable or customizable for each tool. This way, the only thing the programmer needs to do to integrate two tools is to translate the semantics of their interfaces.

2.3 Current State

As of the time of working on this thesis, there already were two OSLC adapters for two analysis tools implemented and working at our university, as was briefly mentioned in the motivation of this work (Section 1.1)

The first one is an OSLC adapter for the ANaConDA framework [13][11] which allows ANaConDA to be used as a regular dynamic analysis tool through an OSLC Automation interface. This adapter was created using Eclipse Lyo and is connected with a database for persistency, but has very limited additional features and a very simple build system which requires the SUT[1] to consist of a single file and to be transferred to the server again for every analysis. It was created in my bachelor thesis [40] and was successfully used to experimentally deploy ANaConDA in Honeywell and contributed to the AQUAS project [3] and the AuFoVer project [5].

The second OSLC adapter was created for Perun [14], a performance analysis tool, by modifying the existing ANaConDA adapter in [41] making design decisions with the universal adapter in mind. Perun is tightly coupled with the Git versioning system, and its analyses are meant to follow up on each other. For example, analysing a series of commits and comparing them to look for performance degradation or improvements. This means the adapter needs to maintain SUT resources separately from analysis requests so that multiple analysis requests can be executed on the same SUT with the same context. To manage SUT resources and to take care of compilation all together, the adapter was separated into an Analysis adapter and a Compilation adapter. The adapter was acceptance tested by the developers of Perun and contributed to the Arrowhead Tools project [4].

[1] system under test

Other Analysis Tool Candidates for Adding an OSLC Interface

There are still more tools that need an OSLC interface both at our university and outside of it. Examples of such tools are:

- Spectra [36] - A past-time LTL verification tool developed at VeriFIT.
- Facebook Infer [10] - A static analysis tool used at VeriFIT to create static analysers.
- Valgrind [39] - A dynamic analysis and instrumentation tool commonly used by developers.
- RoadRunner [16] - A dynamic analysis framework for Java.

Chapter 3

OSLC — Open Services for Lifecycle collaboration

This chapter introduces OSLC (Open Services for Lifecycle collaboration) [25] and its core concepts along with the underlying technologies[1]. The introduction focuses on explaining aspects of OSLC that are used in this work and therefore necessary to know to understand this work. For more information or a more detailed explanation refer to the OSLC Web site [25].

3.1 Foundation technologies of OSLC

Before OSLC itself is introduced, it is important to know the three main technologies that it is built on top of. This section names those technologies and provides a very quick overview of what they are.

3.1.1 REST

REST (REpresentation State Transfer) [15] is an architectural style for Web based applications which is mainly used to enable communication between computer systems. REST is resource-based, meaning it works with things rather than actions. Resources are represented and transferred using serializable formats like *JSON* or *XML*. A *REST API* consists of endpoints identified by *URI*s[2] to which it is possible to send different types of *HTTP* requests 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 the 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`.

3.1.2 RDF

RDF (Resource Description Framework) [33] is a data model for Web applications. Data is organized into triplets of *subject-predicate-object* which create relations between resources. Both resources and relations (properties) are identified by URIs which are used to represent them in triplets. RDF can have multiple representations, such as *turtle* [6], *RDF/XML*, or

[1] The bulk of this chapter has been taken from my bachelor's thesis [40] with modifications and updates.
[2] Unique-Resource-Identifier

RDF/JSON. Databases for persistent storage of RDF subject-predicate-object triplets are called triplestores. *Triplestores* [46] are optimized for storing triplets and querying triplets using a specialized query language, *SPARQL* [17].

3.1.3 Linked Data

Linked Data [45] is a set of rules defining how to publish and connect data on the Web 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.*“ [1]

3.2 Overview of OSLC

OSLC [25] is an *OASIS Open Project* focused on interoperability of software tools throughout the whole software development lifecycle. OSLC promotes a federated architecture where every tool manages its own resources and others can interact with them without knowing the internal implementation of the managing tool. 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* [15, 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. Each resource has an RDF representation, such as *RDF/XML*, *XML*, or *JSON*. Finally, resources can be linked together by URI identified *relations*.

OSLC specifications are divided based on integration scenarios into *domains*, such as *Quality Management*, *Requirements Management*, *Change Management*, or *Architecture Management*[3]. Domains can be imagined as definitions of standardized interfaces. They define resources and vocabularies that an interface should use as its elements. In order to define the basic communication protocol and rules there is the *OSLC Core* 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. Key elements of OSLC specifications are usage rules and patterns for HTTP and RDF, resource shapes and constraints, resource representation, resource operations, resource discovery, resource querying, and authentication. Figure 3.1 shows the layered architecture of OSLC.

Currently, the latest version of the OSLC specification is 3.0 including OSLC Core 3.0 released in September 2020, but some domains are still only at version 2.0 or 2.1. All specifications are designed to be backward-compatible. This work focuses on OSLC Automation 2.1 (latest version) and OSLC Core 3.0.

[3]For OSLC domain specifications refer to [30]

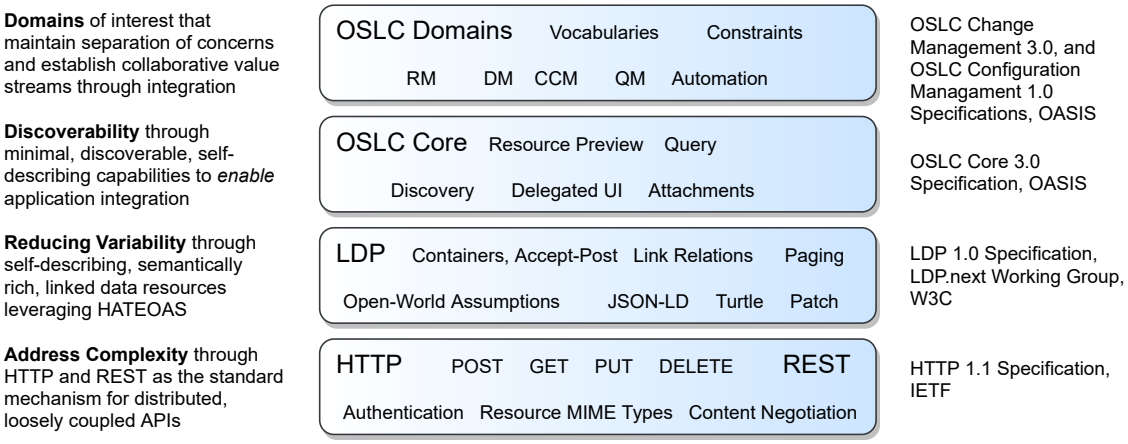

Figure 3.1: OSLC layered architecture (source: [28], remade)

3.2.1 OSLC Core Domain

The Core domain [26] specifies the basics needed by all other domains. It specifies how to use HTTP, RDF, resources in a standardized way and what features are required for all domains. It also defines error responses, resource paging, authentication, resource operations, common vocabulary, resource previews, delegated UI dialogs and others.

Resource Shapes

OSLC is based around *Resources* that are uniquely identified by URIs. These resources need to have a well defined type which is determined by a URI of a Resource Shape. *Resource Shapes* are RDF resources containing a list of *Properties* and information about those properties, such as `occurrence`, `value type`, or `allowed values`. Shapes, properties and resources are organized into XML namespaces which are defined as a tuple of namespace prefix and namespace URI. Most resource properties are optional which is determined by their occurrence. A resource might have just one or two properties with an occurrence of `exactly-one` or `one-or-many`, and the remaining properties will have occurrences of `zero-or-one` or `zero-or-many`, which makes them optional.

Basic Capabilities

Basic operations 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 typical way of updating a resource is reading the resource, modifying the

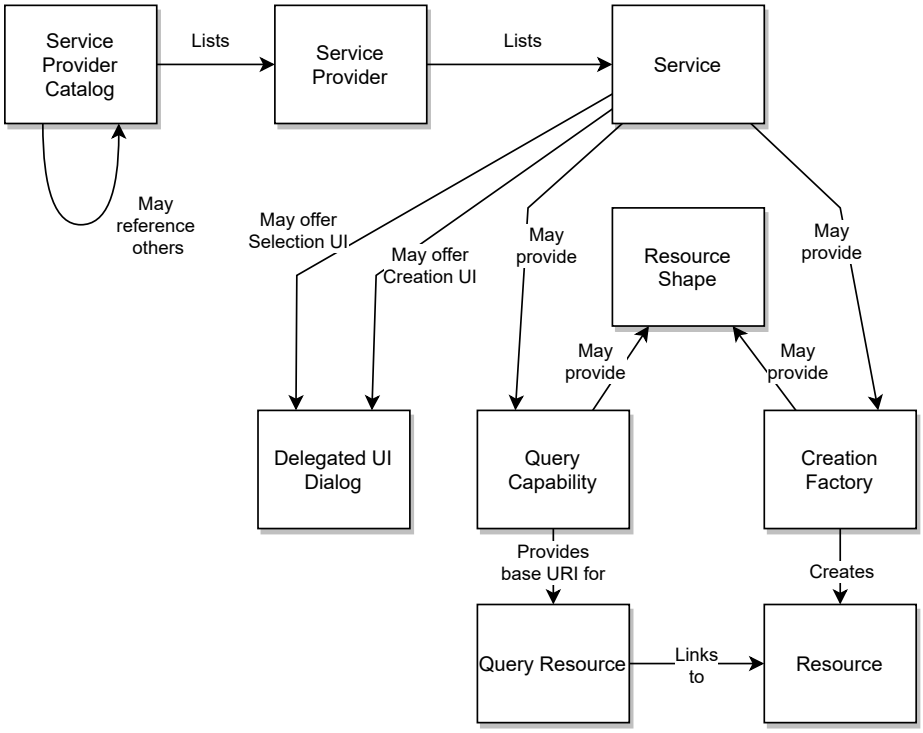

Figure 3.2: OSLC Server elements and their relations (source: [26], remade)

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`.

OSLC Server, OSLC Client, and Resource Discovery

There are two types of OSLC applications, clients and servers. An *OSLC Server*[4] 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 Client*[5] consumes the server's services by manipulating the server's resources through the server's capabilities. A client can discover what resources and capabilities are provided by a server 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 client. 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 3.2 for a graphical representation of the above described resources and their relations.

When an OSLC client wants to consume services provided by an OSLC server, all it needs to know is the URI of the service provider catalogue. From there it 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

[4]Formerly called *Provider* in OSLC Core 2.0 which was deprecated and replaced with *Server* in 3.0
[5]Formerly called *Consumer* in OSLC Core 2.0 which was deprecated and replaced with *Client* in 3.0

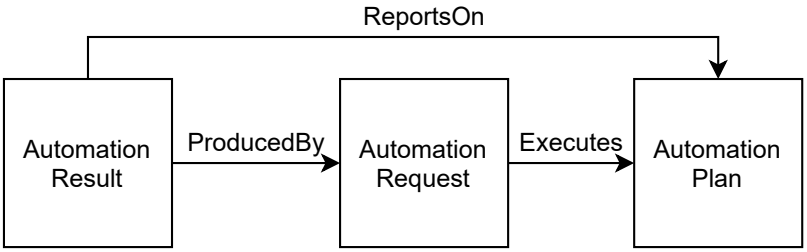

Figure 3.3: OSLC Automation resources and their relations [27]

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 server for all managed resources. Once a resource is created the client can read, update, or delete the resource by sending GET, PUT, or DELETE requests to the resource URI, provided that the OSLC server 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.

Query Capability

Query capability is used to list all resources or to find a subset of resources like in conventional database queries. 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. The request parameters `page` and `limit` are used to cycle through pages or change the page size. The OSLC Query Syntax specification [34] defines query parameters that can be added when using query capabilities to filter the results. 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 implied in their names. They should be implemented using a combination of HTML `<iframe>` and JavaScript to allow integrating tools to use the integrated tool's specific UI as part of their own UI.

3.3 OSLC Automation Domain

The OSLC Automation domain [27] is the one this work focuses on because it is the one the adapter will be using for its interface. This domain is meant for 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 its execution. The core resources of the OSLC Automation domain are *Automation Plans*, *Automation Requests* and *Automation Results*. Their relations are shown in Figure 3.3.

3.3.1 Automation Plan

Automation Plans 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` [26] which are resources with properties, such as `name`, `allowed values`, `occurrence`, `value type`, or `default values` making them similar to definitions of regular command-line parameters. 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 server with a single predefined Automation Plan that can be viewed by OSLC client, and used as a guide for creating Automation Requests similar to a `--help` command-line parameter.

3.3.2 Automation Request

Automation Requests are created by OSLC clients to request execution of an Automation Plan. The most important properties of a request are `inputParameter`, `state`, `desiredState`, and `executesAutomationPlan` reference. Input parameter properties represent parameters submitted as inputs of the Automation Request and they reference `Parameter Instance` resources whose properties include `name`, `value`, and `description`. making them similar to command-line arguments passed as parameters. Each input parameter has to match a Parameter Definition defined in the referenced executed Automation Plan. 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 client 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 server will create an Automation Result. It is the client's responsibility to poll for the result based on the Automation Request's state which can be `new`, `inProgress`, `queued`, `complete`, `canceling`, or `canceled`.

3.3.3 Automation Result

Automation Results are produced by the OSLC server based on an Automation Request. The most important part of a result is its `contribution` properties which reference various artifacts that were produced by the Automation Request execution, such as text output logs, binary files, graphs, 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 and the possible values are `unavailable`, `passed`, `failed`, `warning` or `error`. An Automation Result references the same input parameters as the originating Automation Request, and also contains new output parameters. `Output Parameters` represent input parameters which had their values changed during execution or represent parameters added by the OSLC server 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 server, to add contributions to the result in the case that there are any; or, its used to cancel execution of an Automation Request by updating its `desiredState` property.

3.4 Eclipse Lyo

Eclipse Lyo [19] 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.

3.4.1 OSLC4J

OSLC4J [19] is a Java toolkit for developing OSLC servers and clients. It contains Java object annotations for OSLC attributes and UI previews, support for service provider and resource shape documents, libraries for developing servers and clients, sample applications, and a test suite. 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.

3.4.2 Lyo Designer

Lyo Designer [19] is an Eclipse plugin meant for development of OSLC servers and clients. 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 then 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 adapter interface needs to have its internal structure modeled by an adapter interface diagram. *Adapter 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 capabilities (read, update, delete) in form of a separate service , creation factory, query capability, and selection and creation dialogs. This work uses the latest version of the designer which is 4.0.[6]

Lyo Code Generator [8] 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 adapter, all the required logic to run a working Maven Web application is generated including the Maven project itself. 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 since Lyo Designer 2.4 and only provide basic browsing functionality without modifications. Since Lyo Designer 4.0, a Swagger UI [35] is generated for the modeled adapter which can be used as a fully capable REST client replacement for using the adapter.

[6] Previously created adapter's for ANaConDA and Perun were based on Lyo Designer 2.4

To create a single adapter 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. Then, the last steps remaining are populating the service provider catalogue (service provider infos function), and implementing application logic inside placeholder functions for resource capabilities.

3.4.3 Lyo Domains

Lyo Domains [21] 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 manually, modifying them beforehand if needed.

3.4.4 Lyo Store

Lyo Store [23] 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 [34] 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 persistence and query capabilities to an OSLC server.

3.4.5 Lyo RIO

Lyo RIO [22] 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 servers and clients to experiment with.

3.4.6 Lyo Test Suite

Lyo provides a test suite [24] 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 servers 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. However, there is no need to use the test suite when using Lyo Designer since it already generates OSLC compliant code.

Chapter 4

Defining Analysis Tool Requirements

In order to design an universal adapter, usage requirements of different analysis tools need to be examined. This chapter discusses what requirements need to be fulfilled by the adapter during various stages of analysis in general; what special requirements different types of tools have, and what limitations the adapter has. A summary of how these requirements are fulfilled by the adapter is listed in Section 6.2. Tools used as representatives for the requirements study include:

- ANaConDA [11] - dynamic analysis,
- Valgrind [39] - dynamic analysis,
- Perun [14] - dynamic analysis and Git integration,
- Facebook Infer [10] - static analysis,
- Grep - very simple static analysis and general UNIX utility representative,
- HiLiTE - test case generation,
- Spectra [36] - dynamic analysis.

4.1 General Requirements for Analysis Execution

This section examines different stages of analysis execution — setup, before, initiation, during, and after. We tried to cover all requirements needed for an analysis tool to be used properly as if the user was working directly with a command-line.

4.1.1 Initial Setup

To prepare the analysis executor, the analysis tool first needs to be installed. This step is only performed once, typically by the system administrator (not an adapter client). The process of installing the tool itself does not need to be covered by the adapter's OSLC interface. However, since the adapter is universal it needs to have customizable configuration which allows the system administrator to register the newly installed tool with the adapter so that the adapter knows what the tools interface looks like and can then provide its analysis capabilities to clients.

4.1.2 Pre-Analysis Execution

Once a specific analysis of a SUT is planned, the SUT needs to be transferred to the analysis server and prepared for analysis. This functionality is its own automation scenario and has its own adapter in our solution. The two requirements mentioned in this section both require their own output logs, input parameters, and can have different outcomes. That is why they have been moved into their own adapter which is separated from analysis execution. More details on the actual solution will be provided in Chapter 5.

Transfer the SUT to the server

Files of the SUT need to be transferred to the server which is running the OSLC adapter. A variety of ways to transfer a SUT need to be supported to accommodate different client use cases, such as cloning a Git repository or downloading the SUT from a general URL, direct upload of the SUT, and path to a SUT which the client transferred to the server through its own means. The adapter needs to provide as many of these transfer options as possible. In addition, the process of fetching the SUT might fail, e.g. due to a connection issue. This means the adapter needs to provide logs of the SUT transfer process so that users can look at them to see errors.

Building the SUT

The SUT might need to be compiled depending on the kind of SUT and analysis tool. Typically a SUT would be a folder containing a number of source files and an included build solution, such as a Makefile, a build script, Gradle, or Maven. The build process is performed by executing a build command that uses one of the mentioned build solutions. Thus, the adapter needs to allow clients to submit any build command and then be able to execute it. In addition, the process of building the SUT might produce outputs that need to be available to users so they can check whether the build process went as expected, and the build process could fail which means the adapter needs to represent the compilation result somehow.

4.1.3 On Analysis Initiation

Once a SUT is ready on the analysis server and there is a particular analysis to be executed, the user might need to modify the configuration of the analysis tool for the specific analysis and then start the analysis execution.

Configure the analysis tool

The analysis tool might have parameters that can be configured that change the tools behavior during the analysis. For example, ANaConDA has configurable noise-injection[1] or location information verbosity through a configuration file. Such configuration files can be located in the SUT directory or might only be located inside of the tool's directories. The adapter needs to provide a way to create and modify these configuration files in the SUT directory for each analysis execution. However, changing configuration files located in the tool's directories can prove difficult, since it might collide with other concurrent analyses that share the same configuration file and need different configuration.

[1] Inserting sleep into the program's execution to enforce less common thread interleavings.

Analysis tools can also be configurable using environment variables, so the adapter needs to provide a way to set environment variables for executing analysis.

Discover the analysis tool's parameters

A new user who has never used an analysis tool might need to look up its usage typically using a `--help` parameter. The adapter needs to provide a way to discover input parameters either by executing the analysis tool with a `--help` parameter or by having the tools interface entirely described in the tool's Automation Plan.

Specify Tool Input Parameters and Execute Analysis

Executing the analysis itself is usually performed by calling the analysis tool with certain command-line arguments. The adapter needs to allow clients to submit any input parameters as they would on a command-line and properly interpret commands which can contain quotes, wildcards, etc. The adapter should also check whether the submitted input parameters are correct or not to avoid calling the analysis tool only to get an error result due to incorrect input parameters. Often, users would also want to execute analysis with a timeout in case it deadlocks or just runs longer than acceptable, or the user might want to execute the analysis multiple times and aggregate the final result, or the user might want to queue up multiple different analysis runs in a sequence. The adapter should provide such features too, either through its own input parameters or by allowing users to run their own custom scripts as the analysis launch command.

4.1.4 During Analysis Execution

Once the analysis has been started, a command-line user could see the analysis progress in real time and would be able to abort the analysis if needed.

Monitor analysis status

Monitoring the analysis progress can be useful for the user to get information on how much time is left or if the tool already reported any errors, etc. The adapter should allow users to poll for the status of the analysis execution, ideally providing the latest state of the tool's outputs as well.

Cancel execution

A user might decide that an analysis was started by accident, or with the wrong input parameters, or is taking too long, and can decide to abort the analysis. The adapter should allow users to abort analysis execution as well.

4.1.5 Post-Analysis Execution

After the analysis finishes, the user wants to browse and process the outputs of the analysis, get information about the analysis run, such as the total time, or prepare for more analysis runs.

Get analysis outputs

The main reason for performing an analysis is getting the analysis tool outputs. Typically the standard output, standard error output, and return code. But analysis tools can also produce new files with outputs, graphs, or other artifacts. The adapter needs to allow users to retrieve all possible analysis tool outputs including all newly produced files. A user might even want to retrieve the whole SUT directory to use it elsewhere with the modifications produced by the analysis tool (e.g. modified source codes).

Get information about execution run

The adapter needs to provide information about the analysis run, such as the total running time, the resources used, etc.

Persist analysis outputs

The user might want to save analysis results persistently into a database. The adapter could provide this functionality for users using a persistent database.

Examine analysis outputs

Typically an analysis tool would produce an output which contains individual reports of issues found in the SUT during the analysis. Users might then, for example, only be interested in analysis results which contain logs that did find issues in the SUT. Therefore, the adapter should allow users to query contents of analysis outputs or otherwise post-process them to produce other aggregated results.

Run follow up analysis

The user might want to run a new analysis on top of the current analysis results, for example, when using multiple tools which use each other's outputs. This means the adapter needs to provide a way of running multiple analyses with the same analysis context.

Clean up

If SUTs retain all modifications made to them by analysis, there might be scenarios where a new analysis might require the SUT to be clean, with new fresh context which should also be achievable in the adapter. This could be achieved by a cleaning operation on a SUT or by cloning clean SUT.

4.2 Tool Type Specific Requirements

The previous section focused on general requirements needed by most tools. This section focuses on specific requirements of different types of tools.

4.2.1 Dynamic Analysis

Dynamic analysis (Section 2.1) tools perform analysis by monitoring the execution of a SUT. Dynamic analysis tools that were considered in this work include ANaConDA, Valgrind, and Spectra. Their special requirements come from the need to execute the analysed SUT.

Build the SUT

In order for a SUT to be executable it needs to be compiled or otherwise prepared after being transferred to the analysis server. A need to build the SUT was already mentioned in the previous section as a general requirement but is pointed out here again because it is specific to dynamic analysis tools. The adapter needs to provide a way to build even the most complex SUT's by using their intended build systems. In addition, a dynamic analysis tool might need to compile the SUT in its own specific way (which is the case with Spectra). Which means the adapter needs to allow clients to skip the initial build process when creating a SUT at the server and push it back to the analysis tool level which comes later.

Use the SUT launch command

The analysis tool needs to be able to launch a SUT in order to be able to analyse its execution. This means the tool typically expects the launch command of the SUT to be passed as one of its input arguments. Thus, the adapter needs to have the SUT launch command saved as one of the SUT resource's properties so that analysis can then look it up later.

4.2.2 Static Analysis

Static analysis (Section 2.1) tools perform analysis by scanning the SUT's source codes. Unlike dynamic analysis tools, they do not need the SUT to be compiled and can even require it not to be compiled so that they can build it themselves to see which source files are needed. Static analysis tools that are considered in this work include Facebook Infer and Grep.

Knowledge of the SUT build command

The analysis tool needs to know which source files is the SUT made out of. In the case of Facebook Infer, this is achieved by supplying the SUT build command to it as one of its input arguments. The adapter needs to represent the SUT build command as one of the SUT's properties so that analysis tools can look it up later.

Building the SUT might not be required

Static analysis tools do not need the SUT to be compiled and ready to launch. They might even be usable on a fragment of a SUT which is not compilable on its own. The adapter needs to allow clients to supply a build command with the SUT but skip the initial build process while creating the SUT on the server in case the build is not possible at all or not desirable.

4.2.3 Stateful Analysis or Combination of Tools

Some analysis tools might need multiple analyses to be performed in succession on a single instance of a SUT. Tools like Perun, for example, can analyse whether performance has degraded across multiple commits, which means performance analysis needs to be performed on each commit and then an aggregating analysis needs to be executed on the previous

results. Another option is using a combination of tools which depend on each other's outputs.

Keeping SUT context for multiple analysis executions

Analysis tools might need to perform multiple analyses on the same SUT. RESTful interfaces are stateless, however, the analysis server manages stateful resources like a SUT resource in our case. This allows the adapter to use one SUT resource in multiple analysis executions either by physically running all analyses in the same directory, or by managing analysis context as an artifact that can be transferred between directories.

4.2.4 Other Tools

Some tools, which might even be considered as a whole different type of tool other than analysis tools, are tools that do not necessarily analyse a SUT as an application but, for example, generate test cases instead. Their analysis input might be a general artifact like a set of requirements to transform.

Generic artifact as the analysis input

In case of a tool that does not analyse applications it might not be possible to describe its input artifacts using SUT properties like build command or launch command. The adapter needs to allow artifacts of any type or shape to be transferred to the server for analysis. The process of transferring artifacts to the analysis server, however, still might be complex and needs its own result and log of outputs.

4.3 Limitations

Using an analysis tool through a stateless RESTful interface has its limitations. These are discussed in this section.

Interactive Tools

An analysis tool could require continuous interaction from the user to, for example, control analysis decisions. The adapter does not support these tools since creating support for passing inputs to an execution thread would require excessive work, and we did not find any tools that would require such functionality. Typically analysis is started using one command and then fully executes without further interaction.

Tool configuration outside of its input parameters

Analysis tools can have configuration files that alter their behavior. In the case of ANaConDA there is an input parameter that allows users to specify a path to the configuration file which is to be used instead of the default one that is located in the tool's directories. However, if an analysis tool does not provide a similar way to specify a new configuration file for each analysis execution, then the tool itself needs to be modified before each analysis execution. The adapter can provide a way for clients to configure the tool itself prior to executing analysis, but there might be issues when requesting more then one analysis execution with different configurations at the same time. This functionality might be provided

by the adapter if there is a way to, for example, only ever execute one analysis using a given tool at a time.

Running multiple instances of a SUT or an analysis tool

Analysis, or even running applications, can be limited by the hardware capabilities of the analysis server. For example, an analysis tool might use a Matlab distribution as part of its analysis process, which is a fairly expensive application to run, which might result in a requirement to only ever execute one instance of the analysis tool to keep its performance at a reasonable level. Some applications might not even be able to run in multiple instances at all due to various reasons. This includes SUT's to be analysed and not just analysis tools.

Tools without a command-line interface

In case an analysis tool only has a graphical user interface or can otherwise not be used on its own through a command-line, then it can not be used by the adapter. Such tools first need an adapter that allows them to be used as standalone command-line tools.

Chapter 5

OSLC Adapter Design

As mentioned in the previous chapters, the OSLC adapter created in this work was designed to be universal for all command-line analysis tools. This chapter focuses on the design of the adapter. It presents design diagrams and explains design decisions in regard to the analysis tool requirements defined in the previous chapter.

Eclipse Lyo was used to model the adapters domain, toolchain, and capabilities. The created models can be found throughout this chapter. For a description on how to model using Eclipse Lyo and what diagrams are used, see my bachelor's thesis [40], Lyo Wiki [20], or the OSLC Developer Guide [29].

5.1 Architecture — Separating Analysis and Compilation

Analysis tool usage requirements include two main areas - compilation and analysis. These are both mentioned as example integration scenarios for the OSLC Automation domain. The Automation domain (explained in Section 3.3) was picked as the domain used to model our adapter's interface. Automation Plans represent functionality provided by the adapter which is the compilation of SUTs and different kinds of analysis in our case. Automation Requests are what clients create to request execution of any of the provided Automation Plans. Automation Results represent outputs produced by request executions.

Due to compilation and analysis being two different integration scenarios, it was decided to split the designed adapter into two sub-adapters to form a toolchain (two cooperating OSLC servers with separate interfaces). Having two separate interfaces is useful because it avoids mixing two unrelated usage scenarios in one place, which could cause unwanted confusion in the interface. Having a standalone compilation server allows for it to potentially be replaced in the future by a more complete compilation and deployment system created by someone else, and it also allows us to use the Compilation adapter on its own in case there is a suitable use case for it in the future. Figure 5.1 shows the resources making up the interfaces of the two adapters.

The two adapter's are designed to run on the same server so that the Compilation adapter can be used to create SUT's on the server and the Analysis adapter can then access them to execute analysis. Figure 5.2 shows an overview of the adapter's deployment. The left part represents a client that is using the adapter's functionality. The typical way to interact with the adapter is directly using its interface through a REST client (ideally by integrating it with another application) or the Swagger UI built in with the adapter. In the future, however, there could be other ways of using the adapter, such as through a user

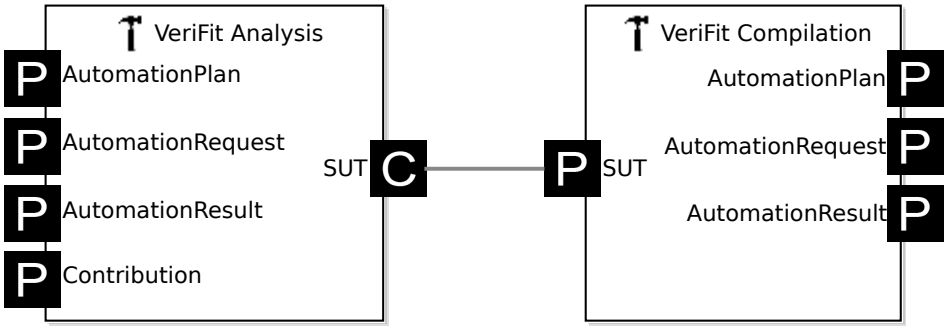

Figure 5.1: Toolchain diagram showing the two adapters. `P` stands for *produces* and `C` stands for *consumes*. Made with Eclipse Lyo

friendly web application or through an IDE plugin (displayed in yellow). The right part shows the analysis server running the OSLC adapter consisting of the Analysis adapter and the Compilation adapter. SUT's are created and managed by the Compilation adapter and used by the Analysis adapter. Analysis tools need to be setup on the same server so that the Analysis adapter can use them to execute analysis. And both the Analysis and Compilation adapter use a database to store their resources. The arrows in the middle (displayed in blue) represent communication between the client and the adapter. The first step is always creating a SUT on the server using the Compilation adapter. The second step is to execute analysis on the previously created SUT.

Figure 5.3 shows a high-level system sequence diagram of the complete OSLC adapter. There are three main parts in the sequence. First, a new client which has never used the adapter before will need to discover its interface through the Service Provider Catalogue. Second part is creating a SUT using the Compilation adapter. And the last part is executing analysis using the Analysis adapter. Interaction with both the Compilation adapter and the Analysis adapter looks very similar because they both use the OSLC Automation interface. Both interactions consist of creating a request, polling for its state, and then getting the final result. Before executing analysis, the Analysis adapter needs to retrieve the SUT resource to be analysed from the Compilation adapter.

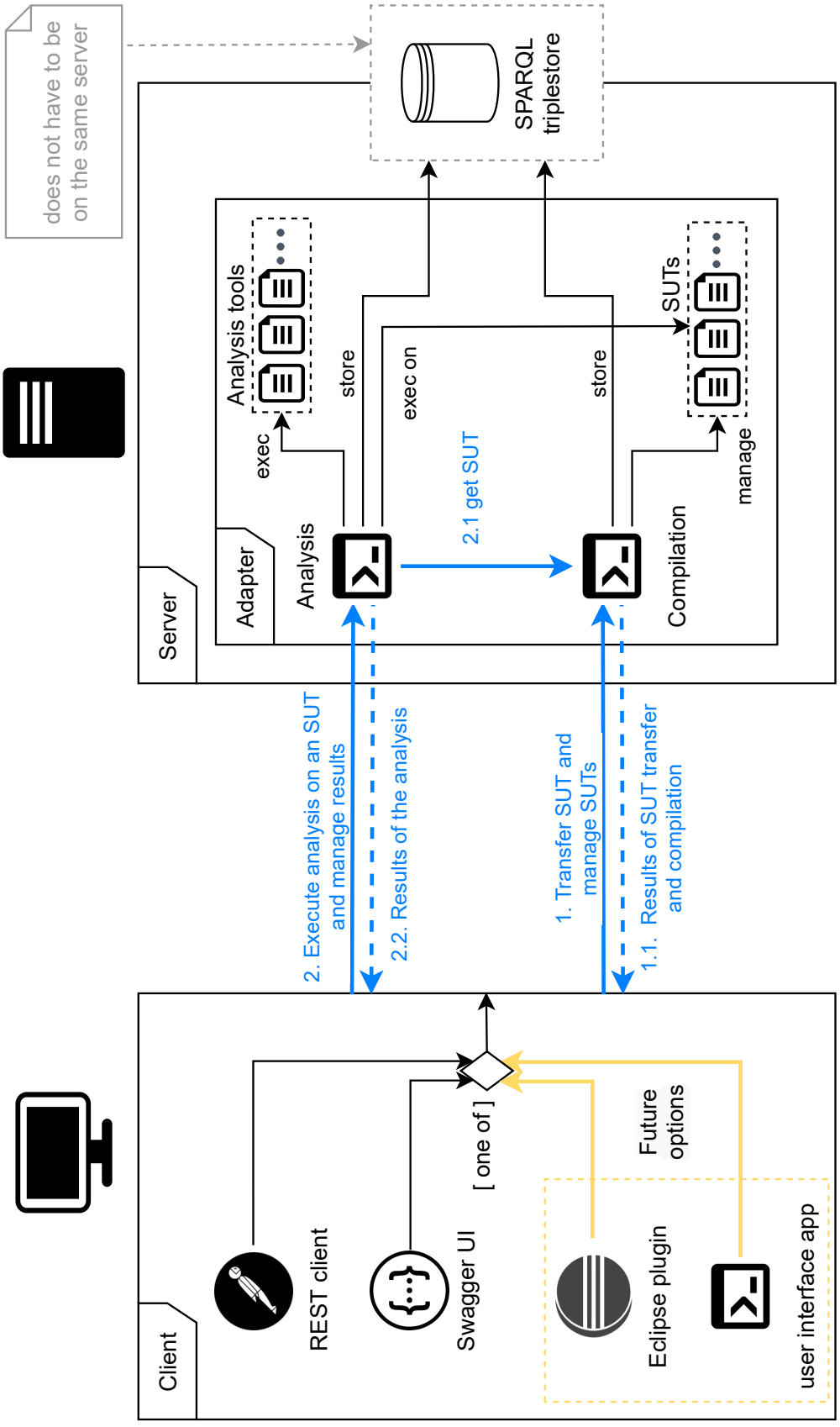

Figure 5.2: Adapter deployment and communication overview.

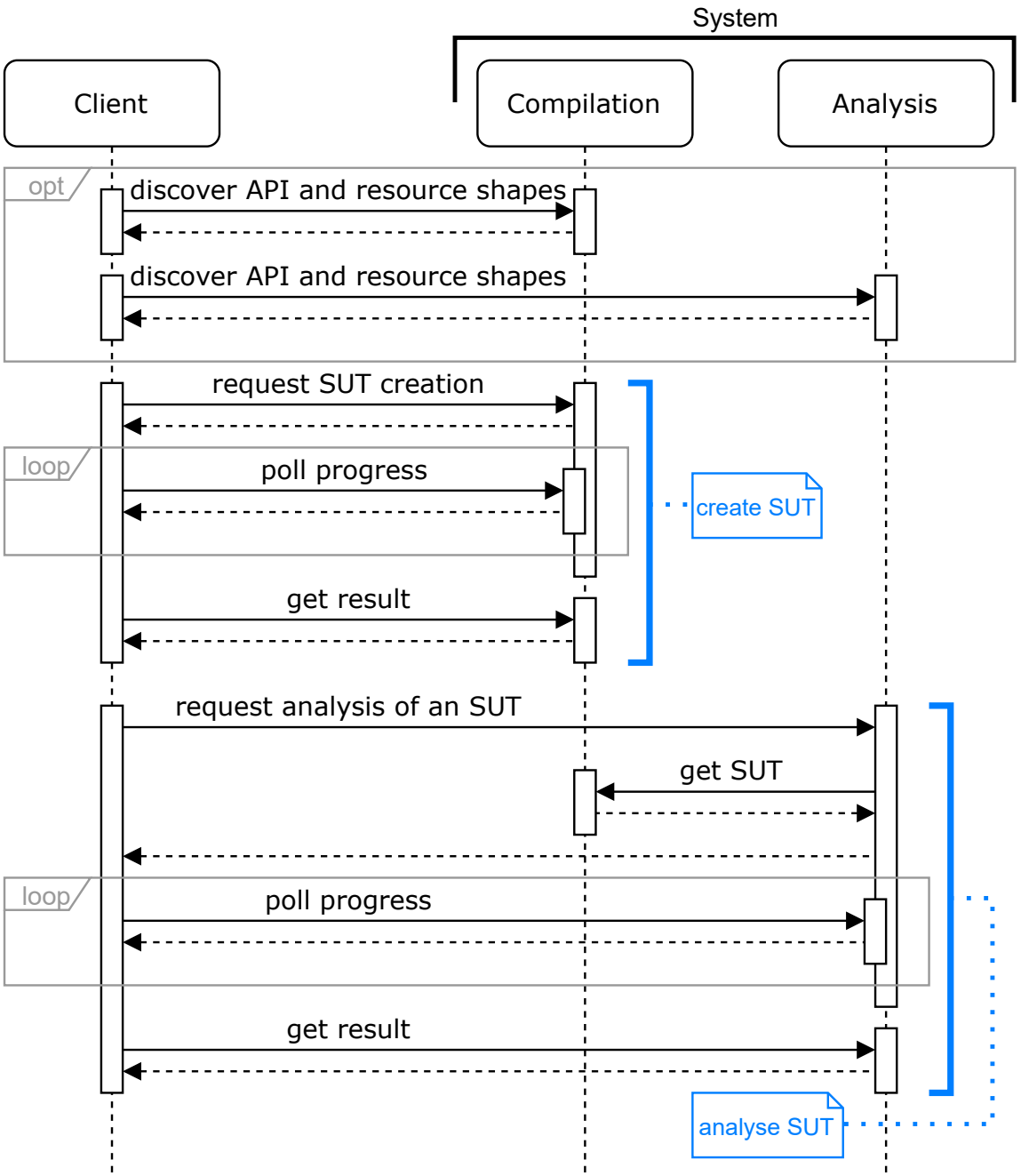

Figure 5.3: High-level system sequence diagram of the Universal Analysis Adapter.

5.2 Domain Model of Both Adapters

As mentioned before, the interface of both adapters is modeled after the OSLC Automation domain. The domain model created in Eclipse Lyo is shown in Figure 5.4. All the Automation domain resources have been imported from Lyo Domains (see Section 3.4.3) with small modifications to make the generated code fit our use case. Modifications to the imported resources all are within the standard's specifications.

One resource relation from the Automation Request to the Automation Result called `producedByAutomationRequest` was added to make it easier for clients to locate Automation Results without using query capabilities. Our adapter can afford using a relation in this direction because it only uses Automation Requests and Results on a one to one basis where there is always exactly one Automation Result for each Request. Another addition is a resource shape for *ParameterDefinition*s, which is modeled after the *oslc:Property* resource as defined by the standard with two modifications — adding a property `fit:commandlinePosition` and adding a property `fit:valuePrefix`. These are then used in the Analysis adapter to define tool interfaces in Automation Plans (more information in Section 6.2.3). Next, a resource shape for *Contribution*s was created with properties recommended by the specification, except for our own custom property called `fit:filePath` that is used in Contributions which represent files created during execution. An entirely new resource `fit:SUT` was created in our own domain namespace. This resource is used to represent SUT resources as described in the previous section and will be described in more detail in Section 5.6. A new relation from the Automation Result to the SUT resource was added. It is called `createdSUT` and is similar to the `contribution` relation used for Contributions. This allows the SUT resource to be more easily retrieved from the Automation Result by looking up the createdSUT property instead of checking all the Contribution resources. The created SUT is, however, also referenced through a Contribution resource for standard compliance.

The `parameterInstance` resource had a `foaf:name` property instead of the `oslc:name` property (this is likely a typo in Lyo Domains). The `allowedValue` and `defaultValue` properties of *ParameterDefinition* resources had value types `Resource` (i.e. a general link to any resource) which were changed to `String`. The standard says the value type can be any resource based on the value of the `valueType` property defined in the same *ParameterDefinition* resource. Properties `inputParameter`, `outputParameter`, `contribution`, and `parameterDefinition` had their representation changed from `Resource` to `LocalResource` which allows the generated code to actually display these properties as local resources. And finally, the `rdf:value` property inside *Contribution* resources was changed to `String` from `XMLLiteral` as the standard allows any values and `XMLLiteral` has semantic restrictions.

5.3 Mapping OSLC Automation Resources to Actions

An important concept to know in order to properly understand the adapter created in this work is how the OSLC Automation resources translate to actions, which a user would otherwise perform with an analysis tool directly. To demonstrate this, we provide an example of using ANaConDA in Figure 5.5. ANaConDA [13][11] is a dynamic analysis tool with a command-line interface. It is executed using a `run.sh` script with various parameters. The only important ones for this demonstration are a `--help` parameter which is used to show a list of all possible parameters, `analyser` parameter which is used to specify which analyser to use, `binary` parameter which stands for the SUT launch command, and `input`

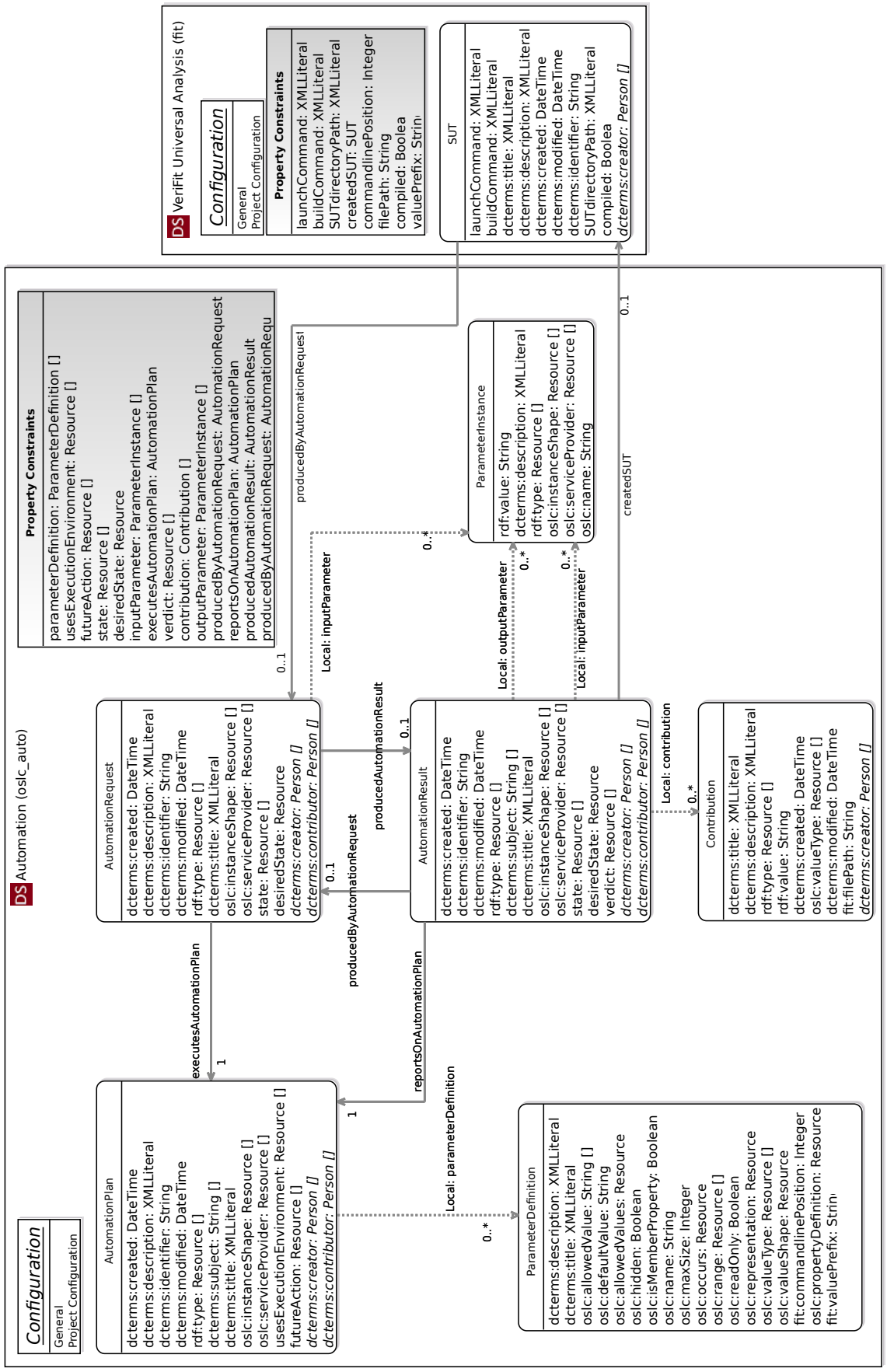

Figure 5.4: Domain model used by both adapters. Made with Eclipse Lyo

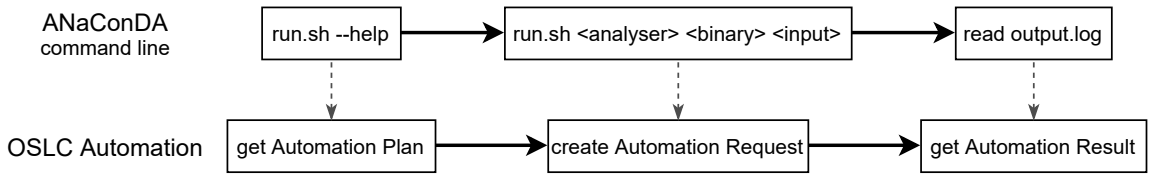

Figure 5.5: Mapping actions performed when using an analysis tool to OSLC Automation domain resources.

which stands for SUT input parameters. Running ANaConDA with the `--help` parameter translates to getting the Automation Plan defined by the Analysis adapter since it contains information about all possible input parameters that can be used to run analysis. Then, running ANaConDA with the other three parameters translates to creating an Automation Request with the appropriate input parameters in the Analysis adapter since it causes the adapter to execute analysis and produce a result. Finally, reading the output produced by ANaConDA (from the standard output or from a file) translates to getting the Automation Result produced by the Analysis adapter since it contains Contribution resources that represent all analysis outputs produced by the execution.

Creating a SUT resource translates to the OSLC Automation domain in a similar way. To learn what input parameters are available when creating a SUT, clients need to get the Automation Plan of the Compilation adapter. Then, to request a SUT to be created, clients need to create an Automation Request in the Compilation adapter with the appropriate input parameters. Finally, to retrieve the created SUT resource (or just a link to it), clients need to get the Automation Result produced by the Compilation adapter.

5.4 Compilation Adapter

The first of the two sub-adapters which form the toolchain created in this work is the Compilation adapter. As outlined in Section 5.1, its main purpose is to create and manage SUT's. Having a separate Compilation adapter is useful because it makes the interface cleaner, allows us to potentially replace the Compilation adapter with a more complete solution in the future, and allows us to reuse it to provide compilation capabilities in other use cases or systems.

Capabilities provided by the Compilation adapter are shown in Figure 5.6 which was modeled in Eclipse Lyo. The adapter contains a single service provider with a single service that holds all of its capabilities. Managed resources are resources from the Automation domain — Automation Plans, Automation Requests, and Automation Results; and our custom SUT resource. Capabilities provided for all the Automation resources were defined based on the standard's specification [27] so that all the capabilities marked as `MUST` and `SHOULD` (i.e. required and recommended) are available to clients. Query capabilities and selection dialogs are defined for all resources even though the standard only requires them for some resources. The adapter provides these capabilities for all resources because of optional persistency and because their implementation is covered by Lyo Store (see Section 3.4.4).

Automation Plans are a read-only resource with a read capability (hidden under *AutomationPlans* in the diagram), a query capability, and a selection dialog.

Automation Request is the only resource that clients can create with a creation factory capability or a creation dialog. Automation Requests have all basic capabilities (read,

update, delete), a query capability, and a selection dialog to allow clients to browse past requests. The update capability is defined by the standard as a way to cancel execution of an Automation Request, and the delete capability was included in the adapter as a way to manage persistent resources.

Automation Result is again a mainly read-only resource for clients with a read capability, a query capability, and a selection dialog. Automation Results, however, also have an update capability defined by the standard to allow agents (other than the adapter that might take part in the automation execution) to contribute to the result, and a delete capability was again included in the adapter for managing persistent resources

SUT resources described in Section 5.6 are the main products of this adapter's automation and were defined for this adapter specifically, therefore there are no standard requirements for them. Capabilities defined for SUT's in the adapter are a read capability, an update capability, a delete capability, a query capability, and a selection dialog. The delete capability was, again, included for persistent resource management, and the update capability is meant to allow users to modify SUT properties like its build command or its launch command.

The Compilation adapter has a single Automation Plan defined. The Automation Plan represents the only unit of automation provided by the Compilation adapter which is creating a SUT resource. Input parameters of the Automation Plan allow clients to pick a way of transferring the SUT's files to the Compilation adapter and to specify parameters of the SUT, such as the build command and the launch command. More details are provided later in Section 6.2.

5.5 Analysis Adapter

The Analysis adapter is the second and more complex one of the two sub-adapters which form the toolchain created in this work. Its purpose is to execute analysis in a universal way so that any analysis tool can be used through the adapter. Capabilities provided by the Analysis adapter are shown in Figure 5.7 which was created in Eclipse Lyo. The adapter has a single service provider with a single service which contains all the adapter's capabilities. Resources from the Automation domain have the same capabilities as was also the case with the Compilation adapter, so to avoid repeating the same information, please refer to the previous Section 5.4.

The differences are that SUT resources are not managed by the Analysis adapter and thus have no capabilities defined by the Analysis adapter. SUT resources are instead consumed by the Analysis adapter which can be seen in the top left part of the diagram. Another difference is that the Analysis adapter provides capabilities for Contribution resources. A read capability allows clients to download files produced by analysis, an update capability allows clients to modify them, and a delete capability allows clients to delete them.

Automation Plans in the Analysis adapter correspond to analysis tools which are adapted by the adapter. Primarily, there is one Automation Plan for each analysis tool. The input parameters of the Automation Plan are divided into two groups. First are the common input parameters for all Automation Plans that control general features of the Analysis adapter itself, which are not directly related to a specific analysis tool. These include parameters, such as execution timeout, reference to the SUT resource to be analysed, and adding files produced by the analysis as Contributions. In the second group are analysis tool parameters which are defined specifically for each analysis tool by the server admin

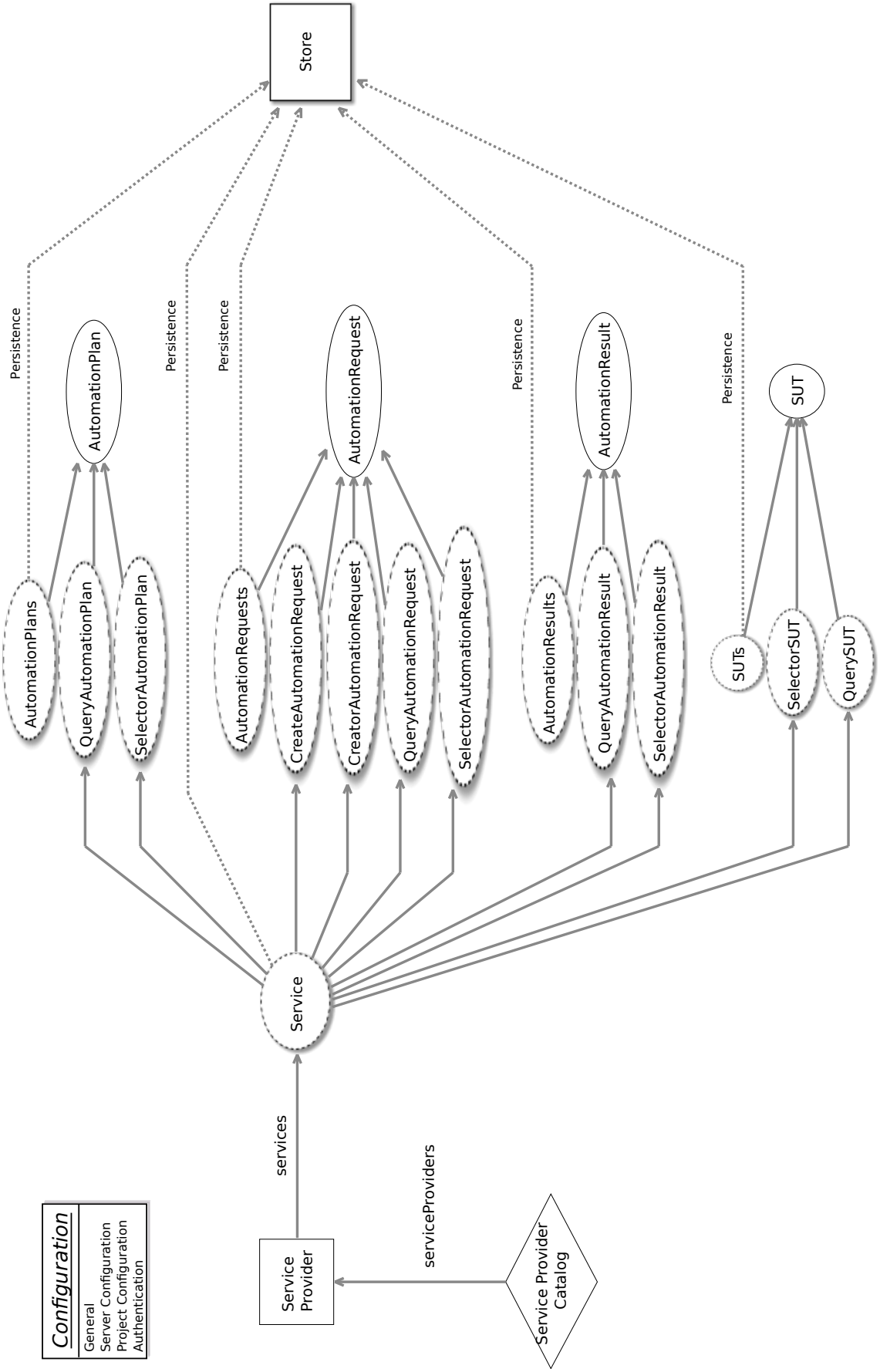

Figure 5.6: Compilation adapter capabilities. Made with Eclipse Lyo

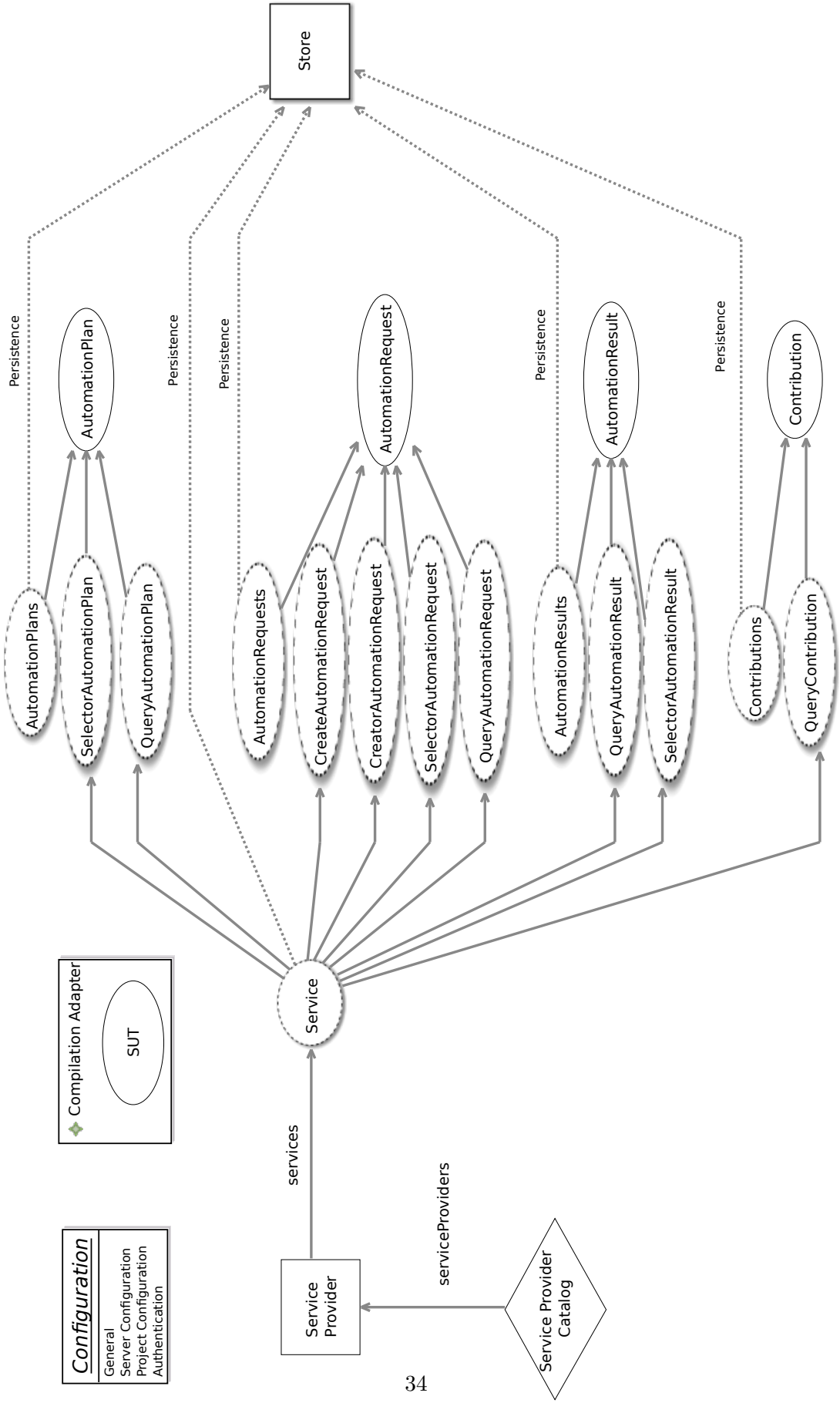

Figure 5.7: Analysis adapter capabilities. Made with Eclipse Lyo

in the adapter's configuration. These parameters define the interface of the specific analysis tool by directly corresponding to possible values to be inserted to specific positions on the command-line as arguments while launching analysis. More details are provided in Section 6.2.

5.6 SUT Representation and Life Cycle

SUTs have been mentioned multiple times previously as a very important part of the adapter toolchain. They are created and managed by the Compilation adapter and then used by the Analysis adapter. The interaction between the two adapters was shown in the high level version of the system sequence diagram in Figure 5.3. When an analysis of a SUT is requested, the Analysis adapter gets the SUT resource to be analysed from the Compilation adapter and uses its properties to execute analysis.

The SUT resource has non-functional properties like title, description, identifier, creation and last modification timestamps, and a creator reference. The most important property of a SUT is its `SUTdirectoryPath` which is a file system path to the SUT directory. This is where the SUT will be transferred to once created and also where the SUT will be compiled. The same directory is also used to run analyses on the specific SUT. Other properties include `buildCommand` and `launchCommand` which hold commands that should be used to compile and launch the SUT respectively. The last property `compiled` is a boolean flag which says whether the SUT has been compiled (i.e. is ready to run).

The lifecycle of a SUT resource can be seen in Figure 5.8. At first, the SUT creation request is created in the Compilation adapter which is represented by the *to be processed* state. Once the Compilation adapter creates a worker thread for the created request, its state changes to *creation in progress*. Depending on the result of the SUT fetching and compilation process, the SUT resource is either created and published to the client or the SUT resource is deleted as failed. SUT creation can also be canceled by the client prior to finishing. After creation, a SUT resource is clean until a first analysis is executed on it by the Analysis adapter. SUT resources work as workspaces for analysis executions, and analysis can make modifications to them. A potential future addition to the lifecycle is a way to clean a SUT to restore it to its initial state (shown in yellow) which would allow clients to get a clean SUT without creating a whole new one.

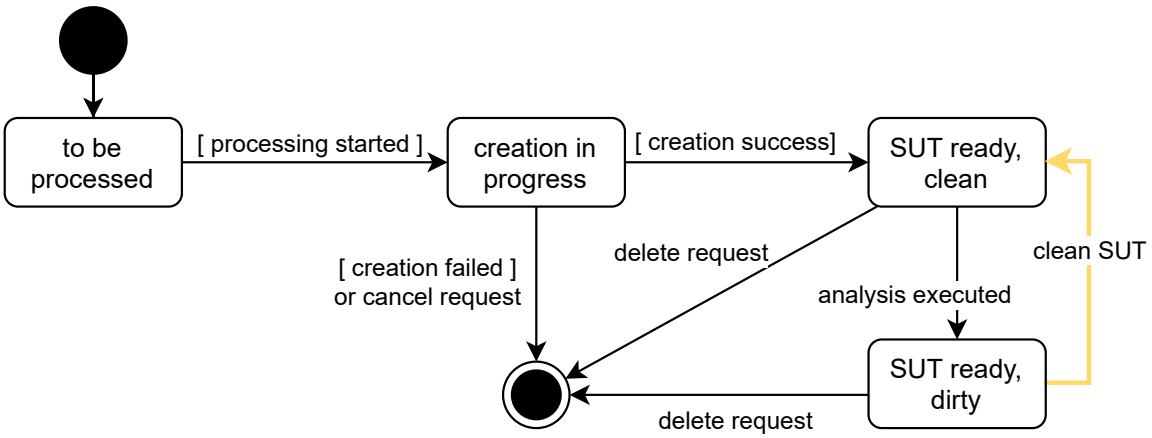

Figure 5.8: SUT lifecycle diagram

Chapter 6

Implementation

This chapter covers important parts of the adapter's implementation, especially in regard to accommodating all analysis tool requirements defined in Chapter 4.

6.1 Adapter Core

The base code of the adapter was generated using Lyo Code Generator based on models created with Lyo Designer which were described in Chapter 5. The generated code is a Java web application managed using Maven [38] with generated classes for all domain resources, a service provider catalogue for API discovery, and with REST API endpoints for modeled capabilities using JAX-RS. Basic visualization of the interface is also generated as a web page using javascript, which allows clients to browse the adapter's interface and the provided resources. However, it does not provide a complete way to create resources without a REST client. Since Lyo 4.0, the generated code includes a Swagger UI [35] which visualizes all available endpoints and their resource types and descriptions. The Swagger UI also allows resources to be created and queried the same way a REST client like Postman [32] would. All generated endpoints support multiple resource representation formats, namely RDF/XML, JSON, turtle, and for retrieving resources also HTML.

In our case, there are two separate Maven projects one for each sub-adapter. Then, there is a third Maven project which contains all resources shared by both adapters including the generated domain classes. The adapter runs using Maven Jetty [7] plugin and is packaged with a set of scripts to build and run the whole adapter as a toolchain including a database. A SPARQL triplestore, Apache Jena Fuseki [37], deployed in a Jetty server distribution is packaged with the adapter to provide persistent storage and query capability without the need to setup a whole fresh database server. The adapter also uses *jSEM* [12] (Simple Extension Manager for Java) a library that allows Java classes to be easily looked up and used dynamically. jSEM is used to allow users to define their own output filters. Scripts packaged with the adapter also take care of configuration by copying files from a single configuration directory into their appropriate places on the build and startup of the adapter. These include configuration of host addresses and ports for both sub-adapters and the database, SPARQL endpoints for the sub-adapter to connect to the database, persistency settings, basic authentication settings, and other internal settings of each sub-adapter. The adapter comes with scripts for both Linux and Windows and was developed and tested on both operating systems.

To provide a clearer idea of what communication with the adapter looks like, refer to Figure 6.1. It is a low level version of Figure 5.3. Automation Requests are created by sending a POST request containing an Automation Request resource to one of the sub-adapters which results in starting either a SUT creation process or an analysis process. The adapter's reply to such a POST request will be an Automation Request resource with all the properties set as the adapter created the resource. Some properties are ignored when specified by the user because they should semantically be determined by the adapter, like a creation and last modification timestamp, URI of the resource, identifier of the request, or state of the request. The identifier of the created Automation Request will match the last part of the resource's URI and it is a serial number set by a counter that increments with each new Automation Request. The state of the created Automation Request should be `new`, but in practice it will mostly be either `inProgress` or `queued`, because the adapter starts processing all requests immediately which means that even the first client response will contain a state other than `new` for optimization reasons. After an Automation Request has been created, the adapter will start executing the required unit of automation (i.e. analysis or compilation depending on the sub-adapter). Clients then have to poll the Automation Request resource by sending GET requests to its URI to check its `state` property, which will be updated by the adapter throughout its execution. Clients need to keep polling the Automation Request until a resource is returned with its state set as `complete` (or `canceled` in case a client has requested a cancellation). Note that clients can choose to poll the Automation Request or its associated Automation Result as both of these resources have the same state property. Automation Results can be identified in three ways. The first one, and the only one defined by the standard, is by querying all Automation Results to find the one which contains a link property to the desired Automation Request (`producedByAutomationRequest` property). This adapter gives clients two more options to identify Automation Results associated with their request. First, Automation Results will have the same identifier as their associated Automation Request which means the last part of their URI will be the same ID number. This allows clients to directly *„guess“* the Automation Result URI by copying the ID. A second more proper way is a custom property `producedAutomationResult`, which is part of all Automation Requests and links to their associated Automation Result. Automation Results need to be retrieved by clients by sending a GET request to their URI in order to see outputs of the execution. The `verdict` property signifies the result of execution from the point of view of the adapter (determined by exceptions being thrown during execution or the return code of the execution). And the Automation result will contain inlined Contribution resources with outputs of the execution (e.g. standard output, return code, created files, etc.).

6.1.1 Executing Commands From Java

An important functionality of both sub-adapters is executing a command submitted by a client. The aim of the adapter is to allow commands to be executed through it exactly the same way as they would be directly through a command-line. This means interpreting wildcards, quotes, escape sequences, pipes, multiple commands on one line, etc. To achieve this the adapter needs to execute submitted commands using the native shell of the server it is running on, which is currently assumed to be *bash* for Linux and *powershell* for Windows. The local shell can be easily executed from Java using *ProcessBuilder*. A minor issue with executing any command submitted by a client is that a command can contain multiple positional arguments which would need to be passed as individual parameters to Process-

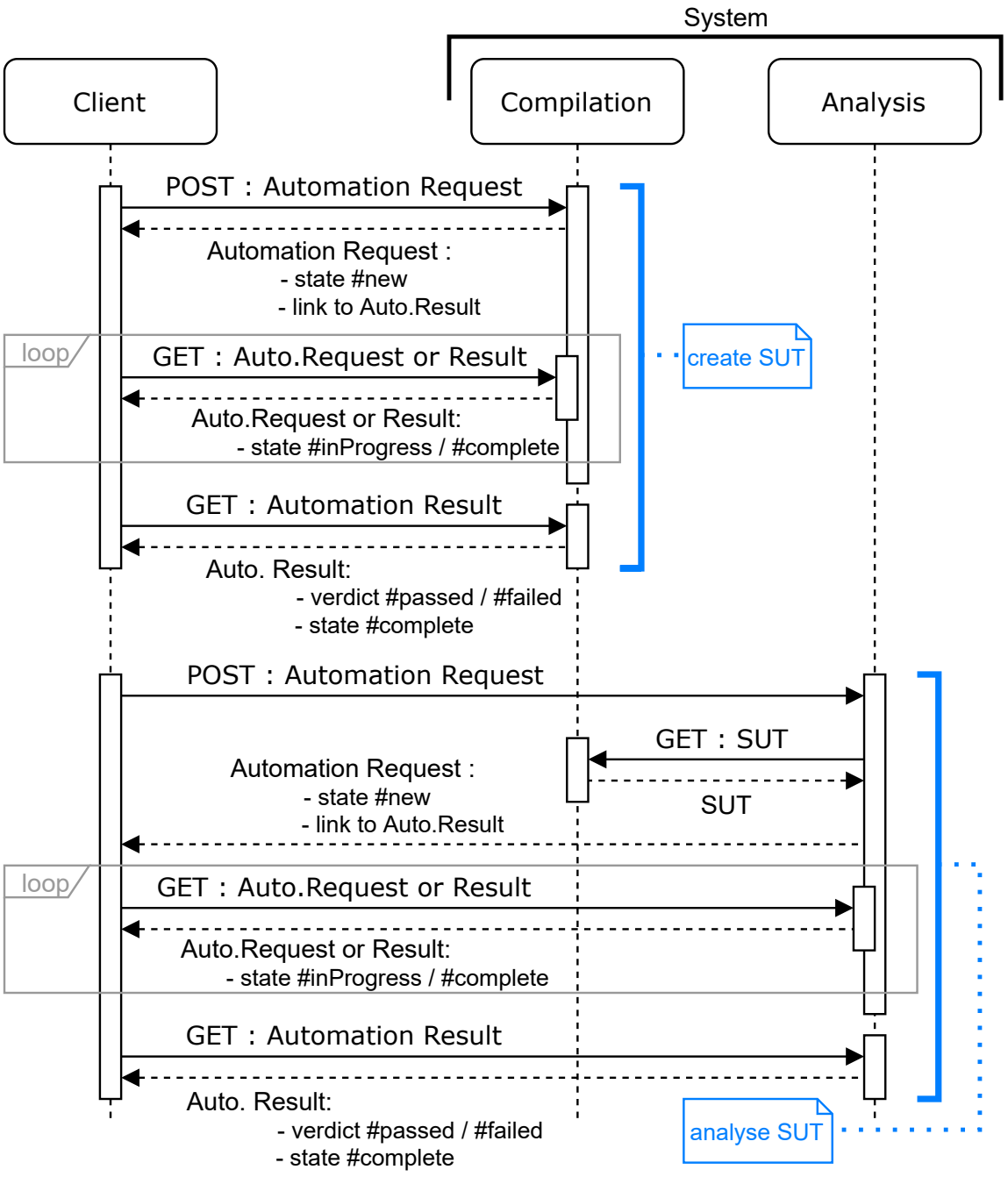

Figure 6.1: Low-level system sequence diagram of the Universal Analysis Adapter.

Builder. A solution for bash was to use `bash -c` as the first parameter, and the whole command to execute as the second parameter. However, a similar solution had issues with preserving quotes in powershell. The current solution does not execute commands directly, but rather pastes them into a script file followed by an `exit $?` or `exit $LastExitCode` first and then executes the created script file. This solution with an added `catch` clause works well for powershell. Although for bash, to preserve functionality the script to execute will contain a `bash -c $1` and the command string to execute needs to be passed as a parameter to the script. This way, execution of multiple commands on one line or with quotes or wildcards is possible using both powershell and bash. This allows clients to *copy-paste* commands they would normally use to compile SUTs or to run their analysis tool directly into the adapter. The executed script will be created in the SUT directory so it can be viewed by clients when debugging their execution in case of any issues. ProcessBuilder also allows environment variables to be set before starting execution, which is then used by the adapter to set environment variables supplied by clients when requesting analysis.

6.1.2 Processing Requests

The way client requests are processed is given by Eclipse Lyo and its generated code. All client requests received by the adapter through its RESTful API are processed using JAX-RS to deserialize OSLC resource objects and passed to stub methods generated by Lyo Code Generator. Each such method corresponds to one of the API endpoints which provide CRUD functionalities for OSLC resources. Most of these methods return an OSLC resource object which is then serialized and returned to the client as a response to their request.

Each request is handled in a separate thread which means multiple request handlers can run concurrently at the same time. `GET`, `UPDATE`, and `DELETE` requests all perform their entire functionality before they return a response to the client. This means verifying that the request is well formed; interacting with the database to either get, update, or delete a resource; and then returning a result to the client. Automation Request `POST` requests are handled differently because their function is to start execution of a unit of automation (analysis, or compilation). The fist step is to verify that the request is valid which means checking that all input parameters and properties have the right values. If the request is not valid, then an error response is returned to the client in the form of an `oslc:Error` resource shape and an appropriate response code. Then, a new Automation Request resource and a matching Automation Result resource are created and saved in the database, and a new thread is spawned that takes care of the actual automation execution. Finally, an Automation Request with state `inProgress` or `queued` is returned to the client as a response. The created thread then performs the requested unit of automation independently of the adapters API and other client requests and updates the Automation Request and Automation Result resources in the database to state `complete` once finished.

6.2 Universally Fulfilling Tool Requirements

This section discusses analysis tool requirements defined in Chapter 4 and presents how the OSLC Universal Analysis Adapter fulfills those requirements. As a reminder, Figure 6.2 summarizes the requirements defined in Chapter 4. The numbering used in the figure will be used as reference in the following sections.

1. General Requirements for Analysis Execution
   (a) Initial Setup (Section 6.2.3)
   (b) Pre-Analysis Execution (Section 6.2.1)
      i. Transfer the SUT to the server
      ii. Compile the SUT
   (c) On Analysis Initiation
      i. Configure the analysis tool (Section 6.2.5)
      ii. Discover the analysis tool's parameters (Section 6.2.3)
      iii. Specify analysis input parameters (Section 6.2.4)
      iv. Execute analysis tool (Section 6.2.4)
   (d) During Analysis Execution (Section 6.2.6)
      i. Monitor analysis status
      ii. Cancel execution
   (e) Post-Analysis Execution
      i. Get analysis outputs (Section 6.2.7)
      ii. Get information about execution run (Section 6.2.7)
      iii. Persist analysis outputs (Section 6.3)
      iv. Examine analysis outputs (Section 6.3)
      v. Run follow up analysis (Section 6.2.2)
      vi. Clean up (Section 6.2.2)
2. Tool Type Specific Requirements (Section 6.2.9)
   (a) Dynamic Analysis (ANaConDA, Valgrind)
      i. Build the SUT (Section 6.2.1)
      ii. Use the SUT launch command (Section 6.2.3)
   (b) Static Analysis (Facebook Infer, grep)
      i. Knowledge of the SUT build command (Section 6.2.3)
      ii. Building the SUT might not be required (Section 6.2.1)
   (c) Stateful Analysis or Combination of Tools (Perun)
      i. Keeping SUT context for multiple analysis executions (Section 6.2.2)
   (d) Other Tools / Test Case Generation (HiLiTE) (Section 6.2.1)
      i. Generic artifact as the analysis input

Figure 6.2: Tool requirements summary. The links to sections on the right side identify sections which describe how the adapter fulfills these requirements.

6.2.1 SUT Creation

Requirements `1.b` are related to creating a SUT resource and are handled by the Compilation adapter. The Compilation adapter has a single Automation Plan which has input parameters that control both SUT transfer to the server and compilation of the SUT.

First group of input parameters control the way the SUT is transferred to the server. There are multiple parameters with different values which the client can choose from, only ever using one of them at a time. These parameters are named `source*type*` and include a Git repository URL to clone, a generic URL to directly download from, a base64 encoded string containing the SUT itself, or a file system path to a file already present on the server. A full list of these parameters is shown below or can be found at the adapter's Wiki page [43].

1. `sourceGit` - SUT files will be cloned from a Git repository using the repository URL[1]
2. `sourceUrl` - SUT files will be directly downloaded from a URL. The downloaded artifact will often be a ZIP file which needs to be unpacked after downloading.
3. `sourceFilePath` - SUT files will be retrieved directly from the server's file system using a file path. This source option is meant for SUT source files which were transferred to the adapter's server through other means (e.g. using an enterprise deployment system).
4. `sourceBase64` - SUT file will be transferred as part of the Automation Request as a base64 encoded string. The parameter's value expects a strict value format with the relative file path and name on the first line and the base64 string starting at the second line.

The second group of input parameters specify properties of the SUT resource to be created. These are parameters which correspond to the SUT's properties, namely its launch command and its build command. The SUT `buildCommand` should contain a string representing a command to be used to compile the SUT and is directly executed by the Compilation adapter. The SUT `launchCommand` is used by the Analysis adapter when running analysis on the SUT, for example, when using a dynamic analysis tool which needs to execute the SUT. Note that both of these properties and their corresponding input parameters are optional to accommodate SUT's that might not have a launch command or a build command (requirement `2.d`). However, these parameters can still be useful even for such SUT's to hold information other than a launch command or a build command, even though this would somewhat violate their intended semantics.

The third and final group are parameters controlling the transferring and compilation process. These currently include two parameters. A boolean parameter `unpackZip` which tells the adapter whether to *unzip* the transferred SUT before compilation. And another boolean parameter `compile` which is used to toggle compilation to allow clients to skip compilation in case the created SUT does not need to be compiled, e.g. when running static analysis (requirement `2.b.ii`).

The Compilation adapter uses Java libraries to transfer SUT files. However, performing compilation has to be done through executing a string using shell to cover all possible compilation methods a SUT can use. The SUT `buildCommand` supplied through automation

[1] Currently, only public repositories are supported.

input parameters is used as the string to be executed to compile the SUT (more detail on executing commands in Section 6.1.1). This way clients can use any compilation method accessible through a regular command-line, such as Make, Maven, gcc, or a custom script.

Output Contributions of SUT creation are divided into two parts. The first part is output of the SUT transfer process which has its own text output log as one Contribution resource. For example, such Contribution resources can contain outputs produced by Git while cloning a repository. The second part is the outputs of the SUT compilation process. These are divided into two Contribution resources — one for `stdout` and one for `stderr`. Contributions contain meaningful outputs even if there was an error during the SUT creation process. For example, a Contribution could contain the message of an exception that was thrown or the error output of executing a Makefile.

The result of the SUT creation process is represented in the `oslc_auto:verdict` property in the corresponding Automation Result. Possible values that the Compilation adapter uses are `unavailable`, `passed`, `failed`, and `error`. The verdict is `unavailable` if SUT creation did not finish yet. Verdicts `failed` and `error` both represent unsuccessful outcomes of the SUT creation process. Verdict `error` is used when there are any exceptions thrown during the SUT creation process. It means that there was an internal error in the Compilation adapter, the build command of the SUT was invalid, or the SUT transfer process failed. Verdict `failed` is used when the Compilation process returns non-zero, signifying that the SUT build command was correct but the build process did not finish successfully, e.g. due to syntax errors in the source code. Verdict `passed` is used when the Compilation process is successfully completed with no errors. An SUT resource is created only if the verdict value is `passed`. The Automation Result of a successful SUT creation process contains a `createdSUT` property which holds a link to the newly created SUT resource.

6.2.2 SUT as Workspaces

Currently, the adapter created in this work uses SUT resources as workspaces, as was mentioned in Section 5.6. This means that each SUT resource corresponds to a directory which is created on the server during SUT creation. The path to the directory is stored in the `SUTdirectoryPath` property of each SUT resource. SUT files get transferred into this directory and the build command is executed in this directory.

When an analysis is requested, the Analysis adapter reads the SUT directory property and executes analysis directly inside of it. Thus, all modifications made to the SUT directory including SUT file modifications or new files have a permanent effect on the SUT resource. This allows clients to run multiple analyses on the same SUT within the same context, e.g. when one analysis produces files which are required by the other analysis (accommodating requirement `1.e.v`). A downside of this solution is that SUT resources will get *dirty* and need to be cleaned for use cases where a clean SUT is required for each analysis. Currently, the only way to achieve this is by creating a whole new SUT resource (the only way to fulfill requirement `1.e.vi`).

Some analysis tools might need files produced by an analysis to be modified before running a follow up analysis. This is the case with Perun [14] which needs to run an *init* command first, and only then can it execute analysis. Furthermore, files produced by the initialization command might need to be modified before executing analysis. To provide this functionality, the Analysis adapter allows users to send UPDATE requests with an `octet stream` data type to Contribution resources which represent files to directly

overwrite contents of those files. Similarly, clients can also download Contribution files directly as `octet streams`.

6.2.3 Analysis Automation Plans and their Configuration

Requirement `1.a` „Initial Setup“ needs to be performed directly by an administrator of the server which runs the adapters. The admin installs an analysis tool which is to be used through the OSLC Universal Analysis Adapter. Running analysis is handled by the Analysis adapter which needs to be configured to be able to provide functionality of the new analysis tool. As mentioned in Section 5.5, the Analysis adapter primarily contains one Automation Plan for each analysis tool. These Automation Plans need to be created by the admin of the analysis server through the adapter's configuration.

Having one Automation Plan for each tool seems to be the best solution for a logically well-defined interface. Automation Plans correspond to units of automation provided by an automation server, and in the case of the Analysis adapter, units of automation correspond to different analysis tools. Clients can then browse available Automation Plans to see what analysis tools are available and look at their properties to learn what input parameters they accept. This is how the Analysis adapter fulfills requirement `1.c.ii`. Note that it can be useful to define multiple Automation Plans for a single analysis tool as this allows Automation Plans to work as pre-prepared templates for running a specific type of analysis using that tool by using default values. This would allow the adapter to, for example, have the most common uses of an analysis to be defined as separate Automation Plans with no input parameters needed from clients.

Automation Plans are loaded from a configuration directory which contains definitions of all Automation Plans for all configured analysis tools. The admin of the server needs to create Automation Plans for every tool that is to be adapted by the Analysis adapter. Basic properties of the Automation Plan, such as identifier, name, description, or creator need to be defined using an XML representation of the actual Automation Plan in an `.rdf` file. An example of such rdf file for ANaConDA is shown in Figure 6.3. The Automation Plan identifier is very important and has to be unique because it is used to make the URI of the Automation Plan, which is used by clients to link Automation Requests to Automation Plans.

The main functional part of the actual Automation Plan resource is its parameter definitions, which define Automation Request input parameters, and the interface of the analysis tool. Parameter definitions are local resources which means they are embedded directly inside of their Automation Plan and do not have their own URIs or capabilities defined for them. Their most important properties are their `name` which identifies them and is used to match them with input parameters submitted with Automation Requests, and their `occurrence` which determines whether an input parameter is required or optional. Other useful properties include `defaultValue` which is a value that is used for an input parameter if a client does not use that parameter explicitly, and `allowedValue` which allows a set of values to be enumerated inside of an Automation Plan which are then used to restrict what values a client can use for those input parameters when creating an Automation Request. There are three types of parameter definitions which can be defined in the Analysis adapter when adding an analysis tool.

The first group are parameter definitions which define the command-line interface of the analysis tool represented by the Automation Plan. These parameter definitions are identified by having a `fit:commandlinePosition` property. Values submitted as input pa-

```
1  <oslc_auto:AutomationPlan>
2    <dcterms:identifier>anaconda</dcterms:identifier>
3
4    <oslc_auto:parameterDefinition><oslc_auto:ParameterDefinition>
5          <oslc:name>config</oslc:name>
6          <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
7          <fit:commandlinePosition rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1<
               /fit:commandlinePosition>
8          <fit:valuePrefix rdf:datatype="http://www.w3.org/2001/XMLSchema#string">--config
               </fit:valuePrefix>
9    </oslc_auto:ParameterDefinition></oslc_auto:parameterDefinition>
10
11   <oslc_auto:parameterDefinition><oslc_auto:ParameterDefinition>
12         <oslc:name>analyser</oslc:name>
13         <fit:commandlinePosition rdf:datatype="http://www.w3.org/2001/XMLSchema#int">2<
               /fit:commandlinePosition>
14         <oslc:occurs rdf:resource="http://open-services.net/ns/core#Exactly-one"/>
15         <oslc:allowedValue>atomrace</oslc:allowedValue>
16         <oslc:allowedValue>fasttrack</oslc:allowedValue>
17   </oslc_auto:ParameterDefinition></oslc_auto:parameterDefinition>
18
19   <oslc_auto:parameterDefinition><oslc_auto:ParameterDefinition>
20         <oslc:name>launchSUT</oslc:name>
21         <fit:commandlinePosition rdf:datatype="http://www.w3.org/2001/XMLSchema#int">3<
               /fit:commandlinePosition>
22         <oslc:defaultValue>True</oslc:defaultValue>
23         <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
24   </oslc_auto:ParameterDefinition></oslc_auto:parameterDefinition>
25
26   <oslc_auto:parameterDefinition><oslc_auto:ParameterDefinition>
27         <oslc:name>executionParameters</oslc:name>
28         <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
29         <fit:commandlinePosition rdf:datatype="http://www.w3.org/2001/XMLSchema#int">4<
               /fit:commandlinePosition>
30   </oslc_auto:ParameterDefinition></oslc_auto:parameterDefinition>
31 </oslc_auto:AutomationPlan>
```

Figure 6.3: Example of an Automation Plan definition inside of an `anaconda.rdf` file used to configure the Analysis adapter to run ANaConDA. The Automation Plan contains four parameter definitions. The first one at lines 4-9 is an optional parameter specifying a path to a configuration directory. The second parameter at lines 11-17 is a compulsory parameter used to pick one of the available analysers. The third parameter at lines 19-24 is a special parameter which instructs the adapter to place the SUT launch command at the specified command-line position. The fourth parameter at line 26-30 is an optional parameter used as inputs for the executed SUT. Note that for space reasons non-functional properties like titles and descriptions were omitted.

rameters when requesting analysis are then combined into a string based on their command-line positions by the Analysis adapter. The whole string containing all parameter values is then prefixed by the analysis tool launch command to form a string to be executed when launching analysis. These parameter definitions can also have an optional property `fit:valuePrefix` which holds a string prefix which should be combined with the parameter value before combining it with all other parameters. This allows non-positional parameters to be used, for example, a parameter which should look like this on the command-line - `--arg argValue`. Such a parameter would have a prefix value of „`--arg` “ and clients would then submit only the „`argValue`“. The simplest and most universal way to define input parameters of an Automation Plan is to have a single input parameter with a string data type. The value of such an input parameter would then hold a free string of any command-line arguments for the analysis tool. However, a better way to define input parameters is having multiple input parameters that correspond to actual arguments in the analysis tool's interface. By defining data types, multiplicity, allowed values, or default values, the adapter can then check whether submitted input parameters are valid and return an error without even launching the analysis tool, instead of going through the whole execution process only to get an error output from the analysis tool.

The second group of parameters are special input parameters which can be defined in an Automation Plan and are recognized by the Analysis adapter and thus processed differently. These are parameters, such as SUT build command or SUT launch command, which also have a `fit:commandlinePosition` property, that instruct the Analysis adapter to lookup the launch command or the build command of the analysed SUT. It then automatically inserts them on a specified command-line position so that the client does not have to look these up themselves. Values of these parameters are boolean toggles for enabling or disabling placement of their respective command on the command-line. These parameters are a quality of life feature to fulfill requirements `2.a.ii` and `2.b.i`.

The third group of parameters are common parameters which are used for all analysis tools and control behavior of the Analysis adapter. These are added to all Automation Plans after they are loaded from the adapter's configuration. These input parameters control features of the Analysis adapter itself which are unrelated to analysis tools, such as an execution timeout, or general processing of files produced by the analysis (for example, a regular expression to match files to be added as Contributions or creating a downloadable zip archive of all output files). These parameters are described in Section 6.2.5. They can be manually defined in the tools Automation Plan configuration as well by using the same parameter definition name. This allows default values of these common parameters to be customized. When the adapter loads Automation Plan configuration and finds a redefinition of one of the common input parameters, it will only take the custom default value and use it to overwrite the built in default value.

The last part of defining an Automation Plan is configuring properties that will not be part of the actual Automation Plan resource. These are located in a `.properties` file with the same name as the `.rdf` file containing the Automation Plan. An example of such a file is shown in Figure 6.4. The most important configuration property is the `toolLaunchCommand`, launch command of the analysis tool which needs to be specified so that the Automation Plan knows how to launch the analysis tool. Other properties to configure include queuing policy (`oneInstanceOnly`) or default arguments for the analysis tool (`toolSpecificArgs`). That need to be included on the command-line for every analysis execution in order to make the tool work with the Analysis adapter by, for example, re-

```
1  # Full path to the tool executable
2  # IMPORTANT: Use double backslash on windows! (\\ instead of just \)
3  toolLaunchCommand=/path/to/anaconda/run.sh
4
5  # Arguments to always use on the command-line when launching the tool
6  #toolSpecificArgs=--no-color
7
8  # If set to true, then only one AutomationRequest executing this AutomationPlan will be
       running at a time and the
9  # remaining ones will be placed in a queue.
10 oneInstanceOnly=False
```

Figure 6.4: Example properties definition inside of a `anaconda.properties` file used to configure the Analysis adapter to run ANaConDA. There are three properties. The first one at line 3 tells the adapter how to execute the analysis tool. The second one at line 6 holds parameters that always need to be included on the command-line, which is not applicable for ANaConDA. The third one at line 10 tells the adapter that any number of ANaConDA instances can run at the same time.

moving colors from outputs which can cause problems with XML encoding of tool outputs or by making the tool non-interactive.

Demonstration of representing tool parameters

To demonstrate what can be done using Parameter Definitions defined above, let's have a look at a few examples of typical types of analysis tool input parameters and how to define them using Parameter Definitions.

The easiest type of parameter is a fixed positional parameter without any value restrictions. Imagine a script with this interface „`./script <arg1> [arg2]`“, where both parameters have no value restrictions and the second one is optional. Parameters for this script would be defined using two Parameter Definitions. The first one would be named `arg1`, would have command-line position `1`, and occurrence `exactly-one`. Optionally, a default value could be defined and the occurrence changed to `zero-or-one` which would result with the default value being used when a client does not supply this parameter, which makes sure that there will always be a value while the parameter is optional for clients. The second parameter would be named `arg2`, and would have command-line position `2` and occurrence `zero-or-one`. Values submitted by clients for these parameters would then be placed at their corresponding command-line positions in the executed string.

Parameters with value restrictions can be defined using allowed value properties. An example of a script „`./script <mode>"`, where mode is a parameter which only allows single digit values of 1 to 3. The Parameter Definition for this parameter would be named `mode`, and would have command-line position `1`, occurrence `exactly-one`, and three allowedValue properties, each with one of the allowed values `1`, `2`, and `3`. Values submitted by clients for such parameters are checked by the adapter and rejected with an error if they do not match any of the allowed values.

The second type of parameters are option flags which do not have a fixed command-line position. Imagine a script with this interface „`./script [options] <arg1>`“, where `arg1` is a simple positional argument just like in the first example with command-line position

2, and `options` represents a number of optional flags. These optional flags are `--flag1`, `--flag2=value`. Both of these option flags will have their own Parameter Definitions named after one of the options and with occurrences `zero-or-one`. The trick to define flags like these is to then define the same command-line position for all of them, in which is `1` in our case. This causes values of these parameters to be placed on to the command-line in random order relative to each other, but at the right position relative to other parameters at other positions. The second trick needed is usage of a value prefix, in our case `--flag2=`, which makes it so that when a client submits a `value`, what actually gets placed on the command-line is `--flag2=value`. The first parameter `--flag1` is more specific in that it does not take any value, so its semantics are only about submitting or not submitting this parameter. To achieve this behavior we can use a value prefix `--flag1` and a single allowed value for this parameter, with the value being empty (eg. an empty xml value property). This will only allow clients to submit an empty value for this parameter or to submit this parameter without a value property all together. What gets placed on the command-line in this case is „`--flag1 `“ where the ending space represents the empty value of this parameter.

The final and third type of parameters are parameters which for any reason can not be defined using the approaches demonstrated above. Any type of parameter can be represented like this `./script <arguments>`, where `arguments` will be represented by a parameter definition which is expected to contain any string. The value of such an argument can then, for example, contain „`--flag1 --flag2=value`“ which is effectively two parameters submitted using a single parameter definition.

A very important thing to note is that the string constructed from parameters submitted by clients is then executed using native shell. This means the string is processed exactly as if it was pasted directly into the command-line, meaning actual positional arguments are separated by spaces (which are inserted by the adapter automatically between parameters) and quotes need to be used to pass values with spaces as a single parameter.

A limitation of this approach is defining parameters for tools with branching interfaces or nested interfaces. Imagine a script that has these two launch options „`./script command1 [options1] <arg1>`“ and „`./script command2 [options2] <arg2>`“. There are two main commands `command1` and `command2`, which unfortunately each have different options and different positional sub-parameters. There is currently no way to logically define such an interface using a single Automation Plan while keeping the definition *clean* (ie. not *hacking* the definition). The way to do this would be to define Parameter Definitions for all possible parameters normally based on their command-line positions. However, clients could then use `command1` with sub-parameters or options which belong to `command2`, which would result in the analysis tool reporting an error. The only way to define such an interface cleanly would be to define a separate Automation Plan for each of the main commands. The first one would only allow users to use `command1` sub-parameters and options, and the same for the second one with `command2`. However, this approach separates the analysis tool into two Automation Plans which can make it harder to identify Automation Results that were created by the same tool.

6.2.4 Analysis execution

Specifying input parameters and launching analysis (requirements 1.c.iii, 1.c.iv) are closely tied with previous sections 6.2.4 and 6.2.3 which describe how commands are executed by the adapter and how input parameters (which form the command to execute) are defined.

Specifying input parameters and executing analysis is performed by creating Automation Requests that execute a selected Automation Plan. To specify input parameters, clients need to add input parameter properties to the new Automation Requests. Every input parameter has to match a Parameter Definition from the executed Automation Plan by having the same name. As described before in Section 6.2.3, Parameter Definitions can have a command-line position property which tells the adapter that their values should be used on the command-line when executing the analysis tool. The Analysis adapter simply takes all input parameter values which correspond to such Parameter Definitions and concatenates them into a single string to be executed. This string is then prefixed by the tool launch command and tool default arguments defined in tool properties which were described in Section 6.2.3 and executed as was described in Section 6.1.1. Note that the adapter can be instructed to omit the analysis tool launch command to allow more flexibility when launching commands. This is needed for Spectra [36] which is a dynamic analysis tool which compiles the SUT itself to inject analysis elements into it and the SUT is then executed on its own without Spectra.

Analysis is executed by the Analysis adapter in separate threads which means multiple analyses can run at the same time. Analysis tools can be configured to only ever allow one instance of them to run at a time. The adapter keeps a FIFO queue for each such analysis tool in which requests are kept and subsequently executed one at a time. A more complex queuing system could be added in the future if needed to, for example, allow priority requests or set of tools which are restricted to only ever run one at a time.

6.2.5 Common Analysis Input Parameters

As mentioned in Section 6.2.3, Automation Plans defined for analysis tools in the Analysis adapter get extended by a number of common input parameter definitions which control the adapter's behavior rather than the analysis tool interface. This section names all the common input parameters since they represent features of the adapter that are important to cover some of the analysis tool requirements from Chapter 4. Most of these parameters are optional and have default values which can be modified by redefining the parameter when configuring an Automation Plan for an analysis tool. In the future there will likely be more features added as new common input parameters to, for example, allow analysis to be executed multiple times, scheduling execution, setting priority of requests, or any other feature that might be needed.

Link to SUT — SUT

The most important parameter added to all Automation Plans is a link to a SUT resource. This is how clients tell the adapter what SUT to analyse which also determines what directory the analysis is going to get executed in.

Execution timeout — timeout

The timeout input parameter is self-explanatory and allows clients to specify a time limit for the analysis execution. If the time limit runs out the analysis is terminated and the `oslc_auto:verdict` of the result will be `failed`.

Matching output files — outputFileRegex

Analysis execution can produce or modify files. These files might need to be of interest to clients and thus the adapter needs a way of knowing what files to include in Automation Results as Contributions. More on analysis outputs in section 6.2.7. The input parameter holds a regular expression which the adapter uses to match files inside of the SUT directory. Any file that was modified during analysis and matches the regular expression will be included as a Contribution for clients to see.

Creating a ZIP file of all output files — zipOutputs

A use case for the Analysis adapter can be to execute analysis and then retrieve all of the files produced by it as a single ZIP file to process manually by a client. For this purpose there is a boolean flag parameter which when set to true results in a special Contribution being included in the Automation Result which represents a ZIP file containing all the output files matched by the regular expression for outputs described in the previous paragraph. The ZIP file can then be directly downloaded using its URI.

Processing outputs — outputFilter

This parameter controls a feature described in section 6.2.7. In short, to give clients more control over analysis outputs the Analysis adapter features plug-in filters which can be created by users to process Contribution resources. There can be multiple filters available for an Automation Plan and clients need to pick one when creating an Automation Request.

Analysis run configuration files — confFile and confDir

These parameters give clients a way to configure analysis tools using configuration files. For example, to allow ANaConDA clients to configure noise injection. An analysis tool can have an input parameter in its native interface that specifies a path to a configuration directory to use when launching analysis. Clients of the adapter can achieve this in two ways.

The first option is the `confFile` parameter which allows clients to specify a file name and its contents when requesting analysis, and then the Analysis adapter will create the file in the SUT directory before executing analysis. The filename can include a directory which will be created automatically, and this parameter can be used multiple times to create multiple configuration files.

The second option is the `confDir` parameter which allows clients to submit a base64 encoded zip file and a path to a directory. The Analysis adapter then creates the requested directory inside of the SUT directory and unzips the base64 encoded file into it.

However, these two parameters only allow configuration of tools which have a way to specify a path to a configuration file or directory when launching analysis. If the configuration file needs to be located outside of the SUT directory, clients need to use a feature described in the next paragraph.

Actions before and after analysis — beforeCommand, afterCommand

To ensure clients can perform any needed configuration for an analysis run, the adapter allows a command to be specified to run just before and just after executing analysis. This allows clients to, for example, change a configuration file which is located in the root folder

of an analysis tool before analysis or to move an analysis tool log file into the SUT directory after analysis so as to include it as an analysis output. Each of these commands will have its own standard output and standard error output Contribution resource in the Automation Result.

If the before command fails (i.e. return non-zero), then analysis will not be executed which allows the before command to be used as a way to create conditional analysis runs. For example, a before command can check whether the compiled SUT can even be launched at all to avoid starting up an expensive analysis tool only for the analysis to fail immediately. Similarly, if the analysis fails, the after command will not be executed.

Note that currently, the beforeCommand can not be used to set environmental variables before analysis because it is executed in a separate shell. To set environmental variables, clients need to use the feature described in the next paragraph.

Setting environment variables — envVariable

Some analysis tools can be configurable using environment variables. The `envVariable` parameter allows clients to set environment variables when requesting analysis. This parameter can be used multiple times to set multiple variables.

6.2.6 Polling for Execution State and Cancelling Execution

Requirements `1.d` are covered directly by the nature of the OSLC Automation domain through the Analysis adapter. Once an Automation Request was created, clients can poll its `oslc_auto:state` property by getting the Automation Request to monitor the state of analysis execution. The same property with a consistently matching value is in the corresponding Automation Result which also contains a `oslc_auto:verdict` property that holds information about the execution's outcome once it finishes. These properties allow clients to monitor the execution state. Currently, the adapter only adds analysis outputs to the Automation Result after execution finishes which means there is no way to see partial outputs during execution. This feature could potentially be added in the feature if needed, however, this would likely require optimizations in database communication to avoid performance issues.

Canceling execution is defined by the standard using a PUT request to update the `desiredState` property of an Automation Request during its execution. By updating the desiredState property to `canceled` clients can request the adapter to cancel execution of an Automation Request. The Analysis adapter also allows clients to delete Automation Requests which causes them to be canceled first and then deleted.

6.2.7 Analysis Output Contributions

This section discusses requirements `1.e` in the Analysis adapter. Outputs of executing analysis by the Analysis adapter are more complicated than in the case of the Compilation adapter because they can be very different depending on the specific analysis tool that is being executed. Contribution resources that are common for all analysis tools are `stdout`, `stderr`, `returnCode`, `executionTime`, and `statusMessage` which are always produced when executing a command. The `stdout`, `stderr` and `returnCode` contributions contain direct outputs of the analysis execution. When a `beforeCommand` or an `afterCommand` is used, then there will also be standard output Contributions for each of the commands. The `executionTime` Contribution contains the total execution time of analysis in millisec-

onds, and the `statusMessage` contains information from the adapter about the execution process.

However, analysis tools can produce files such as images, tables, text logs, or other artifacts. In order to include these as Contributions of an Automation Result, the Analysis adapter needs to know which files to include. This is achieved through an input parameter of the analysis process which holds a regular expression that is used to match file names of files modified during analysis execution (see Section 6.2.5). The adapter then takes a snapshot of the modification times of all files in the analysed SUT directory before executing analysis, and then again after executing analysis. All files modified during analysis that match the specified regular expression are then added as Contributions. All Contributions that represent a file produced by analysis contain a `fit:filePath` property which contains a path to the represented file. Depending on the file type detected by Java, Contributions will have their `valueType` set to `string` for text files or `base64binary` for binary files. Originally, the adapter included contents of all files directly as values of Contribution resources. However, this turned out to not be always desired especially when using persistent storage of resources and running analysis which produces very large files (e.g. ANaConDA can produce gigabytes worth of outputs). For this reason, the Analysis adapter was extended with a plug-in system for user defined output filters described in Section 6.2.8. The adapter comes with a number of predefined output filters, such as a filter that loads contents of stdout and stderr files, or a filter that loads contents of all non-binary files.

Contribution resources have a GET capability defined for an `octet stream` accept header which allows clients to directly download Contribution files.

6.2.8 Analysis Plug-in Output Filters

Clients might need full control over Contribution resources produced during analysis, especially when using persistent storage of resources. The idea is to run all Contributions through a filter which transforms them in any way to fit the clients use case. The filters need to allow clients to process Contribution resources in any way, so a plug-in system was added to the Analysis adapter using the jSEM [12] library that allows users to program their own output filters using Java.

Output filters can be defined by the server admin, and then picked by clients when executing analysis. The output filter is placed at the end of the Automation Result creation process after all Contribution resources have been created. All Contribution resources produced by analysis execution get passed as an in/out parameter to the output filter represented as a set of key-value maps containing entries `name` and `value`. The inputs can then be processed in any way by the output filter, including deleting some Contributions, changing their names or values, extracting information from the inputs to create a new outputs, etc.

To define an output filter users need to create two files in the PluginFilter configuration directory of the adapter. The first one of the files is a `.properties` file which is used by jSEM to identify plug-in classes. An example of such a file is shown in Figure 6.5. There are three properties to define. First is the interface which is being implemented by the plug-in; this property will be the same for all output filters. Second is the class which implements the output filter; users need to change the last part of the qualified path to match their custom class. Third is a property which holds the name of the Automation Plan this output filter is meant for. The second file which needs to be defined is a Java class file containing implementation of the output filter. An example of a class definition is shown in Figure 6.6

```
1 # do not change this value
2 implements=cz.vutbr.fit.group.verifit.oslc.analysis.outputFilters.IFilter
3
4 # class with implementation of the filter -- change the last part only to match your .java file
5 class=pluginFilters.customPluginFilters.ExamplePluginFilter
6
7 # says which tool is this filter meant for -- needs to match an AutomationPlan identifier
8 tool=anaconda
```

Figure 6.5: Example properties defined for a plug-in output filter designed for ANaConDA

(imports have been removed to save space). The class needs to implement two interfaces - IFilter and IExtension[2]. The IExtension interface is required by jSEM, and the IFilter interface is required for an output filter. The IFilter interface defines a `filter` function which has a single parameter which is a list of maps. Each member of the list represents one Contribution as a map of key-value pairs. The keys in each map correspond to Contribution resource properties - `title`, `value`, `valueType`, `filePath`, and `description`. The filePath map element holds a path to a file represented by the Contribution and can be used to load the contents of the file or modify the file. Once an output filter is defined, it will be loaded by the Analysis adapter during its build process[3] and its name will be added as one of the allowed values for the `outputFilter` parameter (defined in Section 6.2.5) of the Automation Plan which the filter was defined for.

The Analysis adapter comes with a number of predefined output filters which are available for all Automation Plans. These currently include filters for loading content of stdout and stderr, loading contents of all non-binary files, and removing all file values. The default output filter loads contents off all non-binary files. Note that filters can be chained to form more complex filters, however, currently the only way to do this is by defining a new filter which chains other filters internally. In the future, there could be an input parameter when creating analysis which would allow a number of filters to be chained together.

6.2.9 Special Requirements

Requirements `2.` listed for different analysis tool types have mostly been covered in the previous sections already.

Building the SUT is covered in Section 6.2.1 just as skipping compilation. Knowing and using the SUT launch command or the SUT build command is provided by having a SUT resource which can be read by clients and contains properties holding the two commands. On top of that, the Analysis adapter provides extra functionality to retrieve the SUT launch command or build command automatically from the SUT resource and insert it to the command-line when launching analysis, as was explained in Section 6.2.3. Keeping context for multiple analysis executions comes from leaving outputs produced by analysis in the analysed SUT directory as mentioned in Section 6.2.2. Finally, using generic artifacts as a SUT can be done just by instructing the Analysis adapter to not perform compilation and not supplying a launch command and a build command if desired.

[2] cz.vutbr.fit.group.verifit.jsem.IExtension; cz.vutbr.fit.group.verifit.oslc.analysis.outputFilters.IFilter;

[3] Unfortunately, in its current state the Analysis adapter has to be re-compiled in order for plug-in filters to update.

```
1  public class ExamplePluginFilter implements IFilter, IExtension {
2
3    final String name = "raceDetectedFilter";
4
5    public void filter(List<Map<String, String>> inoutContributions) {
6
7      // run Contributions through a builtin parser to load stdout and stderr file contents
8      new AddStdoutAndStderrValues().filter(inoutContributions);
9
10     // look for data race detection reports in stdout
11     Boolean dataRaceFound = false;
12     for (Map<String, String> contrib : inoutContributions) {
13       String title = contrib.get("title");
14       if (title.equals("stdout")) {
15         String contentsOfTheStdout = contrib.get("value");
16         if (contentsOfTheStdout.contains("Data race detected at")) {
17           dataRaceFound = true;
18         }
19       }
20     }
21
22     // create a contribution representing the result (based on the stdout contents)
23     Map<String, String> contrib = new HashMap<String, String>();
24     contrib.put("id", "example_id");
25     contrib.put("title", "DataRaceDetected");
26     contrib.put("description", "Holds the result of data race analysis.");
27     contrib.put("value", dataRaceFound.toString());
28     contrib.put("valueType", "http://www.w3.org/2001/XMLSchema#boolean");
29     inoutContributions.add(contrib);
30   }
31 }
```

Figure 6.6: Example of class file definition for an output filter for ANaConDA that loads stdout and stderr file contents, looks for a data race report in stdout, and creates a new Contribution which says whether a data race was reported or not.

6.3 Persistency and Resource Management

The adapter toolchain created in this work uses a SPARQL triplestore database to store all OSLC resources. The triplestore is required for the adapter to function. Lyo Store [23] is used to communicate with the database and the database is a Apache Jena Fuseki [37] distribution deployed in a Jetty [7] server. Lyo Store also provides query capabilities using OSLC Query Syntax [34] for resources when connected to a triplestore. Users should be able to use their own triplestore just by configuring the adapter to use its SPARQL endpoints.

The triplestore can be configured to use persistent storage of resources which can be useful for archiving analysis results. If persistency is not enabled, the adapter will delete all resources on each restart. To enable persistency, users need to create a persistent dataset in the triplestore and configure the Compilation adapter to persist SUT directories in its configuration file. The adapter uses a bookmark Automation Request resource to keep track of the latest issued Automation Request ID to be able to resume creating new IDs after restarts when using persistency. When using persistent storage of resources, clients need a way to manage the persistent database. One way to do this, which can be useful for non-persistent storage as well, is a configuration parameter of the adapter called `keep last N` which defines a window size that determines how many Automation Requests should be kept in the database. All Automation Requests older then the window size will be deleted automatically along with all their associated resources and files. A second way to manage the database is through resource delete and update capabilities.

Clients can pick one of two approaches when deleting resources. The first option is deleting resources one by one, cherry-picking exactly which resources to delete and which to keep. This allows clients to have full control over what gets deleted, but can also lead to broken links. For example, when a client chooses to only delete an Automation Request and leave its associated Automation Result in the database, the Automation Result will contain a link to the deleted Automation Request which will return a `not found` error response when followed. The second option is using a `cascade` parameter set to true when deleting a resource. This will cause the adapter to delete all associated resources at the same time to make sure that the database stays consistent with no broken links. Cascading resources are Automation Request, Automation Results, SUTs for the Compilation adapter; and Automation Request, Automation Results, and Contributions for the Analysis adapter. Cascade deleting any of these will result in all the other ones getting deleted as well, with the exception of deleting Contributions which do not support cascading to other resources because they do not contain a link to their associated Automation Result. Deleting a SUT resource will result in its directory getting physically deleted on the server as well. Note that SUT directories are currently used as analysis workspaces which means deleting a SUT directory will also delete all files produced during analyses of that SUT. Similarly, deleting a Contribution resource which represents a file will result in that file being physically deleted as well.

Updating resources can only be done once their execution has finished. This means Automation Result, Contributions, and SUTs can never be updated while their associated Automation Request has not yet finished. However, Automation Requests themselves can be updated even during execution but only if the update request contains a `state` property set to `canceled`. Such update requests are used to cancel execution of Automation Requests as described in Section 6.2.6. Once execution finishes, all resources can be updated. But not all properties are allowed to be updated because changing them would break adapter logic, such as changing the `identifier` or `state` of an Automation Request, or changing the

`createdByAutomationRequest` property of an Automation Result. Properties which can be updated are `title`, `description`, `creator`, `contributor`, and `extendedProperties`. For Automation Requests it is also the `desiredState` property. For SUTs it is also their `buildCommand` and `launchCommand`. For Contributions it is also their `value`, `valueType` and `filePath`. Note that Contributions which represent a file also accept `octet stream` update requests to update file contents as mentioned in Section 6.2.2.

Every resource created by either of the sub-adapters is immediately stored in the database, and every resource requested from the adapter by clients is retrieved from the adapter. This approach was chosen because performance of the adapter itself was not a priority during implementation, because an analysis run can easily take hours which means slower response times from the adapter due to database communication should not be an issue. Should this become a problem in the future, the adapter could be refactored to cache resources in memory to reduce database communication.

6.4 Security and Authentication

The adapter comes with an inherent security issue since it allows any SUT to be executed on the analysis server. Furthermore, clients can basically execute any command they want on the analysis server as well. This means the adapter can not be used safely to provide functionality to un-trusted clients. However, this security issue comes from the very nature of the adapter use case and there is little that can be done to improve it. The best we can do when dealing with un-trusted clients is to run the adapter in a container which can be easily wiped to its original state in case of a rogue client.

There is another security issue rooted in update and delete capabilities of Contribution resources in the Analysis adapter which are meant to give clients as much control as possible over Contributions. Some Contribution resources can represent files through their `filePath` property and this property can be updated by clients. Should a client update a file path of a Contribution to e.g. „/“ on Linux and then request deletion of the Contribution resource, the adapter would then try to delete the entire root folder of the server. To prevent this from happening, the adapter does not allow Contribution resources to be updated to paths which are outside of the analysed SUT directory and does not allow files outside of the SUT directory to be deleted either. However, this does not prevent a rogue client from setting a SUT launch command to something like `rm -rf /` or uploading a SUT which performs a similar action.

The adapter currently supports Basic Authentication using a single username and password which needs to be defined in the adapter's configuration. The OSLC standard also defines authentication using OAuth, however, this feature was not yet implemented in the adapter due to time constraints and because it is currently not required. The adapter runs on HTTP by default, but can be configured to run on HTTPS. This is just a question of enabling HTTPS in the Jetty plugin which runs the adapter, see the adapters Wiki page [43] for details. Users can also run the adapter in their own container with its own HTTPS settings.

Chapter 7

Experiments and Evaluation

The adapter's functionality was verified by an automated test suite using Postman [32] collections that contain over 700 HTTP requests. The test suite contains system tests that cover the core functionality and error handling of the adapter, features required to fulfill requirements defined in Chapter 4, and basic usage scenarios of all the tested analysis tools. It can be executed using Postman [32] by importing the collections or directly from the command-line using Newman [31]. The adapter includes a test script which executes all tests, and uses GitLab CI to execute tests automatically using a Linux docker image (The GitLab CI configuration file can be found in Appendix C).

Further manual experiments were performed by executing analysis using ANaConDA, Perun, Valgrind, Grep, and Facebook Infer to make sure these tools can be used through the adapter. Section 7.1 goes into detail of using ANaConDA, Valgrind, and Facebook Infer through the adapter.

The adapter covers all the requirements described in Chapter 4. Due to allowing clients to execute almost any command on the server based on their custom Automation Plans with custom input parameters, the adapter should be able to provide functionality of any unix style command-line utility. Persistence of resources is available through configuration of the triplestore and the adapter offers capabilities for maintaining the persistent database.

7.1 Case Studies

To demonstrate that the Universal Analysis Adapter does allow both dynamic analysis tools and static analysis tools to be used, this section presents a case study of using the adapter to run analysis using ANaConDA and Valgrind (dynamic analysis), and using Facebook Infer (static analysis). All examples of XML files included in these case studies have been significantly stripped down due to their size. Examples of full XML files for ANaConDA can be found in Appendix B.

7.1.1 ANaConDA

Command-line interface of ANaConDA [13][11] looks like this:

```
tools/run.sh [optional parameters] <analyser> <sut> <sut inputs>
```

Optional parameters include `--time`, `--config <dir>`, `--run-type <type>`, `--help`, and a few others.

First, let us look at the configuration that needs to be created by the server admin. The Automation Plan is shown in Figure 7.1 and its accompanying properties file is shown in Figure 7.2. These figures are similar to figures used as examples in Section 6.2.3.

Parameters Defined in the Automation Plan correspond to parameters which make up ANaConDA's command-line interface. Lines 4 and 11 represent non-positional optional parameters which all have command-line position `1` to be placed in a random order before the other parameters. The `config` parameter uses a value prefix and also accepts values so that the string placed on the command-line will look like „`valuePrefix inputValue`“. The `time` parameter is a flag that does not accept values, which is represented by a value prefix and a single empty allowed value. This results in the string placed on the command-line containing only the prefix which is the flag itself. Lines 19, 28, and 35 represent positional parameters which have no value prefixes, and some have a restricting set of allowed values. The `launchSUT` parameter (line 28) is a special parameter which will cause the adapter to fetch the SUT launch command and place it at the specified command-line position.

The properties file defined for ANaConDA in Figure 7.2 specifies a path to its launch script and disables queuing of requests.

With configuration complete, let us look at how an Automation Request is created to request analysis execution. First a SUT resource needs to be created using the Compilation adapter which is shown in Figure 7.3.

Input parameters used are: `sourceUrl` (line 6) to specify a SUT source, `buildCommand` (line 14) to set the SUT build command, `launchCommand` (line 18) to set the SUT launch-Command, and `unpackZip` (line 10) to instruct the adapter to unzip the downloaded SUT. The request also needs a link to the `executedAutomationPlan` (line 4) to identify which unit of automation should be executed. The adapters response will contain an Automation Request with a `producedAutomationResult` property which can be used to GET the Automation Result associated with this request. The Automation Result will contain output parameters which hold default values used by the adapter for optional input parameters which were not specified when creating the Automation Request. In this case there will be one output parameter, `compile=true`. Any other `source*` parameter could be used instead of the `sourceUrl` parameter according to the client's needs. Available SUT `source*` parameters are listed in Section 6.2.1.

```
1  <oslc_auto:AutomationPlan>
2    <dcterms:identifier>anaconda</dcterms:identifier>
3    <oslc_auto:parameterDefinition> <oslc_auto:ParameterDefinition>
4      <oslc:name>config</oslc:name>
5      <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
6      <fit:commandlinePosition rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</fit
           :commandlinePosition>
7      <fit:valuePrefix rdf:datatype="http://www.w3.org/2001/XMLSchema#string">--config </
           fit:valuePrefix>
8      <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
9    </oslc_auto:ParameterDefinition> </oslc_auto:parameterDefinition>
10   <oslc_auto:parameterDefinition> <oslc_auto:ParameterDefinition>
11     <oslc:name>time</oslc:name>
12     <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
13     <fit:commandlinePosition rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</fit
           :commandlinePosition>
14     <fit:valuePrefix rdf:datatype="http://www.w3.org/2001/XMLSchema#string">--time </fit
           :valuePrefix>
15     <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#boolean"/>
16     <oslc:allowedValue></oslc:allowedValue>
17   </oslc_auto:ParameterDefinition> </oslc_auto:parameterDefinition>
18   <oslc_auto:parameterDefinition> <oslc_auto:ParameterDefinition>
19     <oslc:name>analyser</oslc:name>
20     <oslc:occurs rdf:resource="http://open-services.net/ns/core#Exactly-one"/>
21     <fit:commandlinePosition rdf:datatype="http://www.w3.org/2001/XMLSchema#int">2</fit
           :commandlinePosition>
22     <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
23     <oslc:allowedValue>atomrace</oslc:allowedValue>
24     <oslc:allowedValue>fasttrack2</oslc:allowedValue>
25     <oslc:allowedValue>tx-monitor</oslc:allowedValue>
26   </oslc_auto:ParameterDefinition> </oslc_auto:parameterDefinition>
27   <oslc_auto:parameterDefinition> <oslc_auto:ParameterDefinition>
28     <oslc:name>launchSUT</oslc:name>
29     <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
30     <fit:commandlinePosition rdf:datatype="http://www.w3.org/2001/XMLSchema#int">3</fit
           :commandlinePosition>
31     <oslc:readOnly rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean">true</oslc:
           readOnly>
32     <oslc:defaultValue>True</oslc:defaultValue>
33   </oslc_auto:ParameterDefinition> </oslc_auto:parameterDefinition>
34   <oslc_auto:parameterDefinition> <oslc_auto:ParameterDefinition>
35     <oslc:name>executionParameters</oslc:name>
36     <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
37     <fit:commandlinePosition rdf:datatype="http://www.w3.org/2001/XMLSchema#int">4</fit
           :commandlinePosition>
38     <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
39     <oslc:defaultValue></oslc:defaultValue>
40   </oslc_auto:ParameterDefinition> </oslc_auto:parameterDefinition>
41 </oslc_auto:AutomationPlan>
```

Figure 7.1: `anaconda.rdf` file used in our experiments with ANaConDA. The file had to be significantly stripped down to fit on one page. Parts that were omitted include enclosing <rdf> tags xmlns definitions, most optional input parameters, and non-functional properties like titles and descriptions.

```
toolLaunchCommand=/path/to/anaconda/run.sh
oneInstanceOnly=False
```

Figure 7.2: `anaconda.properties` file used in our experiments with ANaConDA.

```
POST TO: http://host:port/compilation/services/resources/createAutomationRequest
<oslc_auto:AutomationRequest>
  <dcterms:title>anaconda test</dcterms:title>
  <oslc_auto:executesAutomationPlan rdf:resource="http://host:port/compilation/services/
      resources/automationPlans/0" />
  <oslc_auto:inputParameter> <oslc_auto:ParameterInstance>
    <oslc:name>sourceUrl</oslc:name>
    <rdf:value>http://www.stud.fit.vutbr.cz/~xvasic25/bank.zip</rdf:value>
  </oslc_auto:ParameterInstance> </oslc_auto:inputParameter>
  <oslc_auto:inputParameter> <oslc_auto:ParameterInstance>
    <oslc:name>unpackZip</oslc:name>
    <rdf:value>true</rdf:value>
  </oslc_auto:ParameterInstance> </oslc_auto:inputParameter>
  <oslc_auto:inputParameter> <oslc_auto:ParameterInstance>
      <oslc:name>buildCommand</oslc:name>
      <rdf:value>make</rdf:value>
  </oslc_auto:ParameterInstance> </oslc_auto:inputParameter>
  <oslc_auto:inputParameter> <oslc_auto:ParameterInstance>
      <oslc:name>launchCommand</oslc:name>
      <rdf:value>./bank</rdf:value>
    </oslc_auto:ParameterInstance>
  </oslc_auto:inputParameter> </oslc_auto:AutomationRequest>
```

Figure 7.3: SUT creation Automation Request example.

With a SUT resource created, let us look at how to request analysis execution using ANaConDA. The Automation Request sent to the adapter by a client is shown in Figure 7.4.

Input parameters used from ANaConDA's interface are `time` (line 10), `config` (line 13), `analyser` (line 27), and `executionParameters` (line 31); and common adapter input parameters used are `confDir` (line 17) to create a configuration directory, `outputFilter` (line 23) to select an output filter, and `SUT` (line 6) to link to the SUT resource to be analysed. The string executed by the Analysis adapter will look like this:

```
tools/run.sh --time --config ./anacondaConf atomrace ./bank
```

The `time` and `config` parameters will be placed in a nondeterministic order with their prefix values right after ANaConDA launch command; followed by the SUT launch command fetched from the SUT resource automatically; ending with SUT execution parameters. The `confDir` parameter will cause an `anacondaConf` directory to be created in the SUT directory and filled with configuration files which will then be used by ANaConDA. And the `outputFilter` parameter will cause Contribution resources to be processed using a custom plug-in filter specially for ANaConDA (described later). The created Automation Request will again contain output parameters with default values of parameters not specified by the client. These will include `outputFileRegex=.^` (match nothing), `timeout=0` (no timeout), `zipOutputs=false`, `toolCommand=true` (use tool launch command), and `launchSUT=True` (place the SUT launch command at a specified command-line position).

For this example, the only interesting Contributions inside the Automation Result produced by analysis execution are its standard outputs, its return code, and a custom contribution produced by the output filter. Figure 7.5 shows an Automation Result resource with all properties omitted except for the Contribution resources of interest.

The Automation Result has a `state` property (line 2) and a `verdict` property (line 3) which indicate that the execution was completed successfully. The Automation Result would also include the entire standard output (line 4) produced by ANaConDA (only a part is shown here for space reasons) during analysis that will contain analysis reports which clients are looking for when using ANaConDA. Other Contributions include `stderr` (line 16) and `returnCode` (line 12) There is a custom Contribution produced by the custom output filter (line 22) which holds a boolean value representing whether a data race was detected.

```
POST TO: http://host:port/analysis/services/resources/createAutomationRequest
<oslc_auto:AutomationRequest>
  <dcterms:title>Analysis with ANaConDA</dcterms:title>
  <oslc_auto:executesAutomationPlan rdf:resource="http://host:port/analysis/services/resources
      /automationPlans/anaconda" />
  <oslc_auto:inputParameter><oslc_auto:ParameterInstance>
      <oslc:name>SUT</oslc:name>
      <rdf:value>http://host:port/compilation/services/resources/sUTs/*ID*</rdf:value>
  </oslc_auto:ParameterInstance></oslc_auto:inputParameter>
  <oslc_auto:inputParameter><oslc_auto:ParameterInstance>
      <oslc:name>time</oslc:name>
  </oslc_auto:ParameterInstance></oslc_auto:inputParameter>
  <oslc_auto:inputParameter><oslc_auto:ParameterInstance>
      <oslc:name>config</oslc:name>
      <rdf:value>./anacondaConf</rdf:value>
  </oslc_auto:ParameterInstance></oslc_auto:inputParameter>
  <oslc_auto:inputParameter><oslc_auto:ParameterInstance>
      <oslc:name>confDir</oslc:name>
      <rdf:value>anacondaConf
*base64encoded configuration directory*
</rdf:value>
  </oslc_auto:ParameterInstance></oslc_auto:inputParameter>
  <oslc_auto:inputParameter><oslc_auto:ParameterInstance>
    <oslc:name>outputFilter</oslc:name>
    <rdf:value>AnacondaRaceDetection</rdf:value>
  </oslc_auto:ParameterInstance></oslc_auto:inputParameter>
  <oslc_auto:inputParameter><oslc_auto:ParameterInstance>
      <oslc:name>analyser</oslc:name>
      <rdf:value>atomrace</rdf:value>
  </oslc_auto:ParameterInstance></oslc_auto:inputParameter>
  <oslc_auto:inputParameter><oslc_auto:ParameterInstance>
      <oslc:name>executionParameters</oslc:name>
      <rdf:value></rdf:value>
  </oslc_auto:ParameterInstance></oslc_auto:inputParameter>
</oslc_auto:AutomationRequest>
```

Figure 7.4: Analysis execution Automation Request example for ANaConDA.

```
1  <oslc_auto:AutomationResult rdf:about="http://localhost:8080/analysis/services/resources/
       automationResults/*ID*">
2    <oslc_auto:state rdf:resource="http://open-services.net/ns/auto#complete"/>
3    <oslc_auto:verdict rdf:resource="http://open-services.net/ns/auto#passed"/>
4    <oslc_auto:contribution> <oslc_auto:Contribution rdf:about="http://localhost:8080/analysis/
         services/resources/contributions/*ID*-stdout">
5      <rdf:value>ANaConDA 0.4 20200202 (git fc5f069-dirty)
6          using libdie 0.3 20200202 (git 95d8ccd)
7    ...
8    Write-Read race detected on memory address 0x55e2a3a35160
9    ...
10 </rdf:value>
11     <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
12   <oslc_auto:contribution><oslc_auto:Contribution rdf:about="http://localhost:8080/analysis/
         services/resources/contributions/*ID*-returnCode">
13     <rdf:value>0</rdf:value>
14     <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#integer"/>
15   </oslc_auto:Contribution></oslc_auto:contribution>
16   <oslc_auto:contribution><oslc_auto:Contribution rdf:about="http://localhost:8080/analysis/
         services/resources/contributions/*ID*-stderr">
17     <rdf:value>0.85user 0.10system 0:00.95elapsed 100%CPU (0avgtext+0avgdata 56372
           maxresident)k
18 0inputs+32outputs (0major+42875minor)pagefaults 0swaps
19 </rdf:value>
20     <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
21   </oslc_auto:Contribution></oslc_auto:contribution>
22   <oslc_auto:contribution><oslc_auto:Contribution rdf:about="http://localhost:8080/analysis/
         services/resources/contributions/*ID*-race_detected">
23     <rdf:value>true</rdf:value>
24     <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#boolean"/>
25   </oslc_auto:Contribution></oslc_auto:contribution>
26 </oslc_auto:AutomationResult>
```

Figure 7.5: Analysis execution Automation Result example showing Contributions of interest for ANaConDA. Most properties, such as name, description, valueType, etc. have been omitted to save space.

```
1 implements=cz.vutbr.fit.group.verifit.oslc.analysis.outputFilters.IFilter
2 class=pluginFilters.customPluginFilters.AnacondaRaceDetection
3 tool=anaconda
```

Figure 7.6: Output filter properties file for the ANaConDA output filter

A custom output filter was defined for ANaConDA which searches contents of standard output contributions and creates a new contribution. The output filter needs to be defined first and then it can be used using the `outputFilter` input parameter when creating the Automation Request for executing analysis. Figure 7.6 shows the properties file defining the output filter plug-in for jSEM. Line 1 will always be the same when defining a filter. Line 2 names the filter class. And line 3 specifies which tool is the filter meant for.

Figure 7.7 shows the main `filter` function of the defined output filter. The output filter function first calls a built-in filter (line 4) which loads contents of standard output Contribution files and then searches `stdout` for a data race report from ANaConDA (lines 5-14). Then, a new Contribution resource is added to the current ones (lines 15-21) which holds a boolean value representing whether a data race report was found or not. The output filter class also needs to specify a name for the filter (line 2) which will be used as its identifier for clients.

```
1  public class AnacondaRaceDetection implements IFilter, IExtension {
2    final String name = "AnacondaRaceDetection";
3    public void filter(List<Map<String, String>> inoutContributions) {
4      new AddStdoutAndStderrValues().filter(inoutContributions);
5      Boolean dataRaceFound = false;
6      for (Map<String, String> contrib : inoutContributions) {
7        String title = contrib.get("title");
8        if (title.equals("stdout")) {
9          String contentsOfTheStdout = contrib.get("value");
10         if (contentsOfTheStdout.contains("race detected on memory address")) {
11           dataRaceFound = true;
12         }
13       }
14     }
15     Map<String, String> contrib = new HashMap<String, String>();
16     contrib.put("id", "race_detected");
17     contrib.put("title", "DataRaceDetected");
18     contrib.put("description", "Holds the result of data race analysis.");
19     contrib.put("value", dataRaceFound.toString());
20     contrib.put("valueType", "http://www.w3.org/2001/XMLSchema#boolean");
21     inoutContributions.add(contrib);
22   }
23 }
```

Figure 7.7: Class definition for the ANaConDA output filter

7.1.2 Valgrind

The command-line interface of Valgrind [39] looks like this:

```
valgrind [options] <sut> <sut inputs>
```

The last two parameters are the same as for ANaConDA — `sut` expects a SUT launch command, and `sut inputs` are parameters to be passed to the SUT. Valgrind has a number of options which are non-positional arguments, some of which are just flags, some need values, and some need values but only allow a few specific ones. An Automation Plan which defines the full interface with all options would not fit on one page so Figure 7.8 only shows part of the defined Automation Plan.

Parameter definitions which were excluded are some of the `[options]`. These are represented using the `other-options` parameter (line 11) which can be used to submit any option as a string. These could also be easily defined by copy pasting similar ones which are included and adjusting their names and other properties. The `tool` parameter (line 16) is used to select one of the analysers offered by Valgrind and should be placed at command-line position `1` with all other non-positional option parameters. This parameter only allows a given set of values, which are `helgrind` and `memcheck` in our testing, and needs a value prefix `--tool=` to be passed properly to the command-line (as e.g. `--tool=helgrind`). The `launchSUT` parameter (line 24) is used to represent the `<sut>` parameter from Valgrind's interface and will cause the SUT launch command to be placed at command-line position `2`. The `executionParameters` (line 30) is used to represent the `<sut inputs>` parameter from Valgrind's interface and is used to pass parameters to the executed SUT at the last command-line position. Finally, the `help` parameter (line 4) is defined in this example to demonstrate how to represent such parameters using an empty allowed value and a value prefix.

The properties file defined for Valgrind is very similar to ANaConDA, just with a different path to the tool launch command (line 1). Shown in Figure 7.9

```
1 toolLaunchCommand=/usr/bin/valgrind
2 oneInstanceOnly=False
```

Figure 7.9: `valgrind.properties` file used in our experiments with Valgrind.

Before executing analysis, a SUT resource needs to be created using the Compilation adapter. This can be done in exactly the same way as was shown in Section 7.1.1 and will result in obtaining a SUT resource URI which will be used as one of the input parameters for requesting analysis.

```
1  <oslc_auto:AutomationPlan>
2    <dcterms:identifier>valgrind</dcterms:identifier>
3    <oslc_auto:parameterDefinition><oslc_auto:ParameterDefinition>
4      <oslc:name>help</oslc:name>
5      <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
6      <fit:commandlinePosition rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</fit
           :commandlinePosition>
7      <fit:valuePrefix rdf:datatype="http://www.w3.org/2001/XMLSchema#string">-h</fit:
           valuePrefix>
8      <oslc:allowedValue></oslc:allowedValue>
9    </oslc_auto:ParameterDefinition></oslc_auto:parameterDefinition>
10   <oslc_auto:parameterDefinition><oslc_auto:ParameterDefinition>
11     <oslc:name>other-options</oslc:name>
12     <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
13     <fit:commandlinePosition rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</fit
           :commandlinePosition>
14   </oslc_auto:ParameterDefinition></oslc_auto:parameterDefinition>
15   <oslc_auto:parameterDefinition><oslc_auto:ParameterDefinition>
16     <oslc:name>tool</oslc:name>
17     <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
18     <fit:commandlinePosition rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</fit
           :commandlinePosition>
19     <fit:valuePrefix rdf:datatype="http://www.w3.org/2001/XMLSchema#string">--tool=</
           fit:valuePrefix>
20     <oslc:allowedValue>helgrind</oslc:allowedValue>
21     <oslc:allowedValue>memcheck</oslc:allowedValue>
22   </oslc_auto:ParameterDefinition></oslc_auto:parameterDefinition>
23   <oslc_auto:parameterDefinition><oslc_auto:ParameterDefinition>
24     <oslc:name>launchSUT</oslc:name>
25     <fit:commandlinePosition rdf:datatype="http://www.w3.org/2001/XMLSchema#int">2</fit
           :commandlinePosition>
26     <oslc:defaultValue>True</oslc:defaultValue>
27     <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
28   </oslc_auto:ParameterDefinition></oslc_auto:parameterDefinition>
29   <oslc_auto:parameterDefinition><oslc_auto:ParameterDefinition>
30     <oslc:name>executionParameters</oslc:name>
31     <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
32     <fit:commandlinePosition rdf:datatype="http://www.w3.org/2001/XMLSchema#int">3</fit
           :commandlinePosition>
33     <oslc:defaultValue></oslc:defaultValue>
34   </oslc_auto:ParameterDefinition></oslc_auto:parameterDefinition>
35 </oslc_auto:AutomationPlan>
```

Figure 7.8: `valgrind.rdf` file used in our experiments with Valgrind. The file had to be significantly stripped down to fit on one page. Parts that were omitted include enclosing <rdf> tags xmlns definitions, most optional option flags, and non-functional properties like titles and descriptions.

```
1  POST TO: http://host:port/analysis/services/resources/createAutomationRequest
2  <oslc_auto:AutomationRequest>
3    <dcterms:title>Valgrind analysis request</dcterms:title>
4    <oslc_auto:executesAutomationPlan rdf:resource="http://host:port/analysis/services/resources
         /automationPlans/valgrind" />
5    <oslc_auto:inputParameter><oslc_auto:ParameterInstance>
6      <oslc:name>SUT</oslc:name>
7      <rdf:value>http://host:port/compilation/services/resources/sUTs/*ID*</rdf:value>
8    </oslc_auto:ParameterInstance></oslc_auto:inputParameter>
9    <oslc_auto:inputParameter><oslc_auto:ParameterInstance>
10     <oslc:name>tool</oslc:name>
11     <rdf:value>helgrind</rdf:value>
12   </oslc_auto:ParameterInstance></oslc_auto:inputParameter>
13   <oslc_auto:inputParameter><oslc_auto:ParameterInstance>
14     <oslc:name>executionParameters</oslc:name>
15     <rdf:value>"Hello World!"</rdf:value>
16   </oslc_auto:ParameterInstance></oslc_auto:inputParameter>
17 </oslc_auto:AutomationRequest>
```

Figure 7.10: Analysis execution Automation Request example for Valgrind.

The next step is to POST an Automation Request to the Analysis adapter to request execution of analysis using Valgrind. Example of such a request with input parameters for Valgrind is shown in Figure 7.10.

Input parameters used in the request are: `SUT` (line 6) as a link to the SUT to analyze; `tool` (line 10) which tells Valgrind which one of its analysis tools to use; and `executionParameters` (line 14) which holds input parameters for the analysed SUT. Some default input parameter values will be used and included in the created Automation Request as output parameters. These will include `outputFileRegex=.^` (match nothing), `timeout=0` (no timeout), `zipOutputs=false`, `toolCommand=true` (use tool launch command), and `launchSUT=True` (place the SUT launch command at a specified command-line position). An output filter could be defined similar to the one defined for ANaConDA in Section 7.1.1, however its contents are not included here to save space.

The string executed by the adapter built out of all the input parameters will look like this:

```
/usr/bin/valgrind --tool=helgrind ./sut „Hello World!"
```

The Automation Result created for this analysis request will look very similar to the one shown for ANaConDA in Section 7.1.1. The only difference for Valgrind will be different standard outputs and no custom contributions since no custom output filter was used. An example of what the standard output properties would look like for Valgrind is shown in Figure 7.11.

```
1  <oslc_auto:contribution> <oslc_auto:Contribution rdf:about="http://host:port/analysis/
       services/resources/contributions/*ID*-stderr">
2    <rdf:value>==11869== Helgrind, a~thread error detector
3    ==11869== Copyright (C) 2007-2017, and GNU GPL'd, by OpenWorks LLP et al.
4    ==11869== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
5    ==11869== Command: ./sut "Hello World!"
6    ==11869==
7    ==11869==
8    ==11869== For counts of detected and suppressed errors, rerun with: -v
9    ==11869== Use --history-level=approx or =none to gain increased speed, at
10   ==11869== the cost of reduced accuracy of conflicting-access information
11   ==11869== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
12   </rdf:value>
13 </oslc_auto:Contribution></oslc_auto:contribution>
14 <oslc_auto:contribution> <oslc_auto:Contribution rdf:about="http://host:port/analysis/
       services/resources/contributions/*ID*-stdout">
15   <rdf:value>Hello World!
16   </rdf:value>
17 </oslc_auto:Contribution></oslc_auto:contribution>
```

Figure 7.11: Standard output Contribution resources contained in an Automation Result produced by Valgrind. Most properties, such as name, description, valueType, etc. have been omitted to save space.

7.1.3 Facebook Infer

The last case study in this work is Facebook Infer [10] which differs from the previous two in that it is a static analysis tool. The command-line interface of Facebook Infer looks roughly like this:

```
infer [sub-command] [options] [-- sut build]
```

, where `[sub-command]` stands for optionally specifying one of the Infer sub-commands, such as `run`, `analyze`, or `capture`; and `sut build` stands for optionally supplying the build command of the analysed SUT. Infer has a large number of optional `[options]` parameters so we decided to not define each of them as a separate parameter definition because that would take a long time. Fortunately though, all option parameters should be accepted by all sub-commands so the entire Infer's interface could be defined by defining a separate parameter definition for each option the same way we did for ANaConDA in Section 7.1.1 and for Valgrind in Section 7.1.2. The Automation Plan used for Infer in our experiments is shown in Figure 7.12.

The `sub-command` parameter (line 12) is defined at command-line position `1` and has a number of allowed values defined. The `help` parameter (line 5) defined for Infer is special in that Infer displays help information as paged by default (e.g. using `less` on Linux) which causes it to not be encodable inside of a Contribution resource. To make help output non-paged, the value prefix defined for the `help` parameter needs to actually contain two commands: „`--help --help-format plain`“. Defining value prefixes this way can be useful in other scenarios as well. The `options` (line 23) parameter represents all Infer options to be submitted as a single string to save space. Every single option should be defined as its own parameter definition as mentioned before. Since Infer performs static analysis, it needs

```
1
2  <oslc_auto:AutomationPlan>
3    <dcterms:identifier>infer</dcterms:identifier>
4    <oslc_auto:parameterDefinition><oslc_auto:ParameterDefinition>
5        <oslc:name>help</oslc:name>
6        <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
7        <fit:commandlinePosition rdf:datatype="http://www.w3.org/2001/XMLSchema#int">2</
             fit:commandlinePosition>
8        <fit:valuePrefix rdf:datatype="http://www.w3.org/2001/XMLSchema#string">--help
             --help-format plain</fit:valuePrefix>
9        <oslc:allowedValue></oslc:allowedValue>
10   </oslc_auto:ParameterDefinition></oslc_auto:parameterDefinition>
11   <oslc_auto:parameterDefinition><oslc_auto:ParameterDefinition>
12       <oslc:name>sub-command</oslc:name>
13       <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
14       <fit:commandlinePosition rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</
             fit:commandlinePosition>
15       <oslc:allowedValue>analyze</oslc:allowedValue>
16       <oslc:allowedValue>capture</oslc:allowedValue>
17       <oslc:allowedValue>compile</oslc:allowedValue>
18       <oslc:allowedValue>run</oslc:allowedValue>
19       <oslc:allowedValue></oslc:allowedValue>
20           ...
21   </oslc_auto:ParameterDefinition></oslc_auto:parameterDefinition>
22   <oslc_auto:parameterDefinition><oslc_auto:ParameterDefinition>
23       <oslc:name>options</oslc:name>
24       <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
25       <fit:commandlinePosition rdf:datatype="http://www.w3.org/2001/XMLSchema#int">2</
             fit:commandlinePosition>
26       <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
27   </oslc_auto:ParameterDefinition></oslc_auto:parameterDefinition>
28   <oslc_auto:parameterDefinition><oslc_auto:ParameterDefinition>
29       <oslc:defaultValue>False</oslc:defaultValue>
30       <oslc:name>SUTbuildCommand</oslc:name>
31       <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
32       <fit:commandlinePosition rdf:datatype="http://www.w3.org/2001/XMLSchema#int">3</
             fit:commandlinePosition>
33       <fit:valuePrefix rdf:datatype="http://www.w3.org/2001/XMLSchema#string">-- </fit:
             valuePrefix>
34   </oslc_auto:ParameterDefinition></oslc_auto:parameterDefinition>
35 </oslc_auto:AutomationPlan>
```

Figure 7.12: `infer.rdf` file used in our experiments with Facebook Infer. The file had to be significantly stripped down to fit on one page. Parts that were omitted include enclosing <rdf> tags xmlns definitions, most optional option flags, and non-functional properties like titles and descriptions. The `options` parameter represents all Infer options as a free string to save space.

```
1 toolLaunchCommand=/usr/local/bin/infer
2 oneInstanceOnly=False
```

Figure 7.13: `infer.properties` file used in our experiments with Facebook Infer.

to use the SUT build command which can be done using the `SUTbuildCommand` parameter (line 30). This is a special parameter which instructs the adapter to insert the SUT build command at a specified command-line position. This parameter is not always used so its default value is set to `false` and can be supplied by clients to set it to `true`. Also a prefix value „`--` “ is needed in order for the SUT build command to be properly processed by Infer. This causes a build command `make` to be placed on the command-line as „`-- make`“.

The properties file defined for Facebook Infer is shown in Figure 7.13 and only differs from the other tools in having a different launch command (line 1).

Prior to execution analysis a SUT resource needs to be created using the Compilation provider in almost the same way as in Section 7.1.1. The one difference (apart from picking a different example SUT) is that we need to use the `compile` input parameter set to `false`. This will instruct the Compilation adapter to not compile the SUT so that Infer can launch the build process itself to track which source files to analyse.

Once a SUT has been created, clients can request analysis. An Automation Request for analysing a SUT using Facebook Infer is shown in Figure 7.14.

Infer interface parameters used in this request are: `sub-command` (line 10) to choose an Infer sub-command to run, and `options` (line 14) to pass the `--reactive` option. Adapter common parameters used are: `SUT` (line 6) a link to the SUT to analyse; `SUTbuildCommand` (line 18) set to `true` to insert the SUT build command into the executed string; and a pair of parameters `zipOutputs` (line 22) and `outputFileRegex` (line 26). The `zipOutputs` parameter set to `true` instructs the adapter to create a ZIP file containing all file contributions of the analysis and add it as a new Contribution resource to the Automation Result. In order for this parameter to have any effect, the `outputFileRegex` parameter needs to be used with value `.*infer-out.*` to instruct the adapter to match all files created or modified during analysis as contributions. This is needed for Facebook Infer because analysis produces an `infer-out` directory which contains analysis results which might be important for the client. Some default input parameter values will be used and included in the created Automation Request as output parameters. These will include `timeout=0` (no timeout) and `toolCommand=true`.

The string executed by the adapter built out of all the input parameters will look like this:

```
/usr/local/bin/infer run --reactive -- make
```

The Automation Result produced by the adapter will again be similar to the one created in Section 7.1.1. The main differences will be contents of the standard output contributions, and a number of contributions representing files created or modified during analysis including a ZIP file containing all of them. Interesting Contribution resources are shown in Figure 7.15.

```
1  POST TO: http://host:port/analysis/services/resources/createAutomationRequest
2  <oslc_auto:AutomationRequest>
3    <dcterms:title>Infer analysis request</dcterms:title>
4    <oslc_auto:executesAutomationPlan rdf:resource="http://host:port/analysis/services/resources
         /automationPlans/infer" />
5    <oslc_auto:inputParameter><oslc_auto:ParameterInstance>
6      <oslc:name>SUT</oslc:name>
7      <rdf:value>http://host:port/compilation/services/resources/sUTs/*ID*</rdf:value>
8    </oslc_auto:ParameterInstance></oslc_auto:inputParameter>
9    <oslc_auto:inputParameter><oslc_auto:ParameterInstance>
10       <oslc:name>sub-command</oslc:name>
11       <rdf:value>run</rdf:value>
12   </oslc_auto:ParameterInstance></oslc_auto:inputParameter>
13   <oslc_auto:inputParameter><oslc_auto:ParameterInstance>
14       <oslc:name>options</oslc:name>
15       <rdf:value>--reactive</rdf:value>
16   </oslc_auto:ParameterInstance></oslc_auto:inputParameter>
17   <oslc_auto:inputParameter><oslc_auto:ParameterInstance>
18       <oslc:name>SUTbuildCommand</oslc:name>
19       <rdf:value>true</rdf:value>
20   </oslc_auto:ParameterInstance></oslc_auto:inputParameter>
21   <oslc_auto:inputParameter><oslc_auto:ParameterInstance>
22       <oslc:name>zipOutputs</oslc:name>
23       <rdf:value>True</rdf:value>
24   </oslc_auto:ParameterInstance></oslc_auto:inputParameter>
25   <oslc_auto:inputParameter><oslc_auto:ParameterInstance>
26       <oslc:name>outputFileRegex</oslc:name>
27       <rdf:value>.*infer-out.*</rdf:value>
28   </oslc_auto:ParameterInstance></oslc_auto:inputParameter>
29 </oslc_auto:AutomationRequest>
```

Figure 7.14: Analysis execution Automation Request example for Facebook Infer.

```
1  <oslc_auto:contribution> <oslc_auto:Contribution rdf:about="http://host:port/analysis/
       services/resources/contributions/*ID*-stderr">
2  <rdf:value>javac Resources.java Pointers.java Hello.java
3    Summary of the reports
4
5          RESOURCE_LEAK: 2
6        NULL_DEREFERENCE: 1
7  </rdf:value>
8  </oslc_auto:Contribution></oslc_auto:contribution>
9  <oslc_auto:contribution> <oslc_auto:Contribution rdf:about="http://host:port/analysis/
       services/resources/contributions/*ID*-stdout">
10   <rdf:value>Capturing in make/cc mode...
11   Found 3 (out of 3) source files to analyze in ...
12   Hello.java starting
13   Pointers.java starting
14   Resources.java starting
15   Pointers.java DONE
16   Resources.java DONE
17   Hello.java DONE
18
19   Analysis finished in 622mss
20
21   Found 3 issues
22
23   Hello.java:28: error: NULL_DEREFERENCE
24           object ‘a‘ last assigned on line 26 could be null and is dereferenced at line 28.
25           26. Pointers.A a = Pointers.mayReturnNull(rng.nextInt());
26           27. // FIXME: should check for null before calling method()
27           28. &gt; a.method();
28           29. }
29           30.
30
31   ...
32   </rdf:value>
33 </oslc_auto:Contribution></oslc_auto:contribution>
34 <oslc_auto:contribution><oslc_auto:Contribution rdf:about="http://host:port/analysis/services
       /resources/contributions/*ID*-zipedOutputs">
35           <fit:filePath>path/to/adapter/compilation/SUT/*ID*/outN.zip</fit:filePath>
36 </oslc_auto:Contribution></oslc_auto:contribution>
```

Figure 7.15: Standard output Contribution resources and file contributions contained in an Automation Result produced by Facebook Infer. Most properties, such as name, description, valueType, etc. have been omitted to save space.

The `*ID*-zipedOutputs` Contribution (line 34) is the one that was created by the `zipOutputs` parameter. It can be directly downloaded by sending a GET request to its URI with an `Accept` header set to `application/octet-stream`.

7.2 Usage in Practice

The most important indicator of the adapter's functionality and usefulness is that the adapter is being used in practice with four different analysis tools.

Honeywell uses the adapter for automated test case generation and other analyses using their tool, HiLiTE, through a web client and as part of their requirements verification tool. The main argument for using the adapter was its support for multiple platforms including Windows while providing an OSLC interface because a server was needed that would automate test case generation on the Windows platform. Another useful aspect of the adapter was the web based nature of the interface which allows analysis and test case generation tools to be provided as services. Cooperation with Honeywell brought multiple improvement ideas to the adapter's development and motivated improvements to the adapter's usability, such as better build and run scripts. Honeywell really appreciates the simple extensibility and configurability of the adapter and extensive support from the author to solve all issues. The adapter's powershell scripts successfully went through Honeywell's cyber security audit. Only one unused variable $USRPATH was detected for removal. Honeywell plans to fund future adapter extensions to add:

1. a more complex queuing system to allow priorities for automation requests,
2. improved architecture to make the adapter secure to also handle export control artifacts and external customers — limit SUT and execution commands, and a fail-safe system to remain secure even when the client is compromised,
3. extend existing basic authentication to support external customers and licensing,
4. complete cyber security audit.

The VeriFIT research group [2] from BUT FIT is using the adapter to prepare an analysis server running ANaConDA, Spectra, Perun, and more in the future. Such analysis server is a contribution to the AuFoVer [5] project and is enabled by the adapter created in this work.

Furthermore, an Eclipse plugin for executing analysis from the IDE using the adapter is currently being developed by a VeriFIT member. This will allow the adapter to be used directly from the Eclipse IDE to run analysis which could see the adapter getting even more use cases in the future.

Chapter 8

Conclusion

This work provided an introduction to OSLC, particularly to its Automation domain, and to Eclipse Lyo, particularly to Lyo Designer and Lyo Code Generator.

The goal of this work was to provide a way of adding an OSLC interface to a software analysis tool. This was achieved by designing and implementing an OSLC adapter which can be configured to be used with most command-line analysis tools. In order to design the adapter and its functionality property, a list of analysis tool requirements had to be created based on experiments with a number of different analysis tools. The adapter was designed as a toolchain of two sub-adapters to separate two distinct Automation scenarios: SUT management and analysis execution. The Analysis adapter can be configured to work with most command-line analysis tools and provides features that fulfill all analysis tool requirements in our experiments, including resource persistency and queries.

The functionality of the adapter is backed up by an automated test suite both on Linux and Windows, and manual experiments were conducted with a number of different analysis tools to verify their usability with the adapter. The most important achievement in my opinion is that the adapter created in this work is already being used in Honeywell with positive feedback, and the adapter is also being used or experimented with in two different use cases by researchers from the VeriFIT group. This work also contributes to the projects AuFoVer [5] and Arrowhead Tools [4], and a paper was published at Excel@FIT 2021 [44].

The implemented adapter does cover all analysis tool requirements defined in this work, and in our experiments all analysis tools were fully usable through the adapter. Since the tested tools included representatives of both dynamic and static analysis, I am confident that the adapter is well able to work with almost any command-line analysis tool, however, it would not be right to claim that the adapter can work with *all* analysis tools because that would require first hand experience with *all* analysis tools which I do not have the expertise nor the time for.

Possible future improvements to the adapter include suggestions from Honeywell, such as a more complex queuing system which supports priority requests, more authentication options, and security. Other future ideas include a proper user interface using a standalone web application to provide the adapter's functionality to human clients more conveniently (and not just machine clients); and a *coordinator adapter*[1] which would aggregate multiple servers running the current adapter as workers in a single interface and distribute analysis between them based on available analysis tools, load balancing, or other policies.

[1] formerly a *master* adapter

Bibliography

[1] Berners Lee, T. *Linked Data* [online]. 2006 [cit. 2021-27-04]. Available at: https://www.w3.org/DesignIssues/LinkedData.html.

[2] *Automated Analysis and Verification Research Group - VeriFIT* [online]. [cit. 2021-27-04]. Available at: http://www.fit.vutbr.cz/research/groups/verifit/.

[3] *AQUAS: Aggregated Quality Assurance for Systems* [online]. 2017 [cit. 2021-27-04]. Available at: https://www.fit.vut.cz/research/project/1041/.

[4] *H2020 ECSEL Arrowhead Tools - Arrowhead Tools for Engineering of Digitalisation Solutions* [online]. 2019 [cit. 2021-27-04]. Available at: https://www.fit.vut.cz/research/project/1299/.

[5] *TAČR AuFoVer - Automating Formal Verification* [online]. 2019 [cit. 2021-27-04]. Available at: https://www.vutbr.cz/en/rad/projects/detail/29833.

[6] Carothers, G., Prud'hommeaux, E., Beckett, D. and Berners Lee, T. *RDF 1.1 Turtle*. 2014 [cit. 2021-27-04]. Available at: http://www.w3.org/TR/2014/REC-turtle-20140225/.

[7] Eclipse Foundation. *Eclipse Jetty* [online]. [cit. 2021-27-04]. Available at: https://www.eclipse.org/jetty/.

[8] El khoury, J. Lyo code generator: A model-based code generator for the development of OSLC-compliant tool interfaces. *SoftwareX*. 2016. DOI: 10.1016/j.softx.2016.08.004.

[9] El khoury, J. *An Analysis of the OASIS OSLC Integration Standard, for a Cross-disciplinary Integrated Development Environment : Analysis of market penetration, performance and prospects*. 978-91-7873-525-9. KTH, Mechatronics, 2020. 55 p. QC 20200430.

[10] Facebook, Inc. *Facebook-Infer*. [cit. 2021-27-04]. Available at: https://fbinfer.com/.

[11] Fiedor, J. *ANaConDA Framework* [online]. [cit. 2021-27-04]. Available at: http://www.fit.vutbr.cz/research/groups/verifit/tools/anaconda/.

[12] Fiedor, J. *JSEM - Simple Extension Manager for Java* [online]. [cit. 2021-27-04]. Available at: https://pajda.fit.vutbr.cz/verifit/jsem.

[13] Fiedor, J. and Vojnar, T. ANaConDA: A Framework for Analysing Multi-threaded C/C++ Programs on the Binary Level. In: Qadeer, S. and Tasiran, S., ed. *Runtime Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, p. 35–41. ISBN 978-3-642-35632-2.

[14] Fiedor, T. *Perun* [online]. [cit. 2021-27-04]. Available at: https://github.com/tfiedor/perun.

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

[16] Flanagan, C. and Freund, S. *RoadRunner*. [cit. 2021-27-04]. Available at: https://github.com/stephenfreund/RoadRunner.

[17] Harris, S. and Seaborne, A. *SPARQL 1.1 Query Language*. 2013 [cit. 2021-27-04]. Available at: https://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[18] Letko, Z., Vojnar, T. and Křena, B. AtomRace: Data Race and Atomicity Violation Detector and Healer. In: *Proceedings of the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*. New York, NY, USA: Association for Computing Machinery, 2008. PADTAD '08. DOI: 10.1145/1390841.1390848. ISBN 9781605580524. Available at: https://doi.org/10.1145/1390841.1390848.

[19] *Eclipse Lyo* [online]. [cit. 2021-27-04]. Available at: https://www.eclipse.org/lyo/.

[20] *Lyo Designer Wiki* [online]. [cit. 2021-27-04]. Available at: https://github.com/eclipse/lyo.designer/wiki.

[21] *Lyo Domains* [online]. [cit. 2021-27-04]. Available at: https://github.com/eclipse/lyo.domains/.

[22] *Lyo RIO* [online]. [cit. 2021-27-04]. Available at: https://github.com/eclipse/lyo.rio.

[23] *Lyo Store* [online]. [cit. 2021-27-04]. Available at: https://github.com/eclipse/lyo.store.

[24] *Lyo Test Suite* [online]. [cit. 2021-27-04]. Available at: https://github.com/eclipse/lyo.testsuite.

[25] *Open Services for Lifecycle Collaboration* [online]. [cit. 2021-27-04]. Available at: https://open-services.net/.

[26] *Open Services for Lifecycle Collaboration Core Specification Version 2.0* [online]. [cit. 2021-27-04]. Edited by John Arwe. 30 May 2013. Available at: https://archive.open-services.net/bin/view/Main/OslcCoreSpecification.html.

[27] *OSLC Automation Version 2.1 Part 1: Specification* [online]. [cit. 2021-27-04]. Edited by Fabio Ribeiro. 03 March 2019. OASIS Working Draft 01. Available at: https://rawgit.com/oasis-tcs/oslc-domains/master/auto/automation-spec.html.

[28] *OSLC Core Version 3.0. Part 1: Overview* [online]. [cit. 2021-27-04]. Edited by Jim Amsden. 31 May 2018. OASIS Committee Specification Draft 03 / Public Review Draft 03. Available at: http://docs.oasis-open.org/oslc-core/oslc-core/v3.0/csprd03/part1-overview/oslc-core-v3.0-csprd03-part1-overview.html.

[29] *OSLC Developer Guide* [online]. [cit. 2021-27-04]. Available at: https://oslc.github.io/developing-oslc-applications/.

[30] *OSLC Specifications* [online]. [cit. 2021-27-04]. Available at: https://open-services.net/specifications/.

[31] Postman. *Newman* [online]. [cit. 2021-27-04]. Available at: https://www.npmjs.com/package/newman.

[32] Postman. *Postman* [online]. [cit. 2021-27-04]. Available at: https://www.getpostman.com/.

[33] RDF Working Group. *Resource Description Framework (RDF)* [online]. 2004. 2014 [cit. 2021-27-04]. Available at: https://www.w3.org/RDF/.

[34] Ryman, A. *Open Services for Lifecycle Collaboration Core Specification Version 2.0 Query Syntax* [online]. [cit. 2021-27-04]. Available at: https://archive.open-services.net/bin/view/Main/OSLCCoreSpecQuery.

[35] SmartBear. *Swagger* [online]. [cit. 2021-27-04]. Available at: https://swagger.io/.

[36] Smrčka, A. *Testos - Spectra* [online]. [cit. 2021-27-04]. Available at: https://www.fit.vutbr.cz/research/groups/verifit/tools/testos-spectra/.cs.

[37] The Apache Software Foundation. *Apache Jena Fuseki* [online]. [cit. 2021-27-04]. Available at: https://jena.apache.org/documentation/fuseki2/.

[38] The Apache Software Foundation. *Apache Maven* [online]. [cit. 2021-27-04]. Available at: https://maven.apache.org/.

[39] Valgrind^{TM} Developers. *Valgrind* [online]. [cit. 2021-27-04]. Available at: https://valgrind.org/.

[40] Vašíček, O. *OSLC Adapter for ANaConDA Framework* [Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Aleš Smrčka, Ph.D.]. Brno, CZ, 2019. Bachelor's thesis. Vysoké učení technické v Brně, Fakulta informačních technologií.

[41] Vašíček, O. *OSLC Adapter for Perun* [FIT VUT v Brně]. 2020. Projektová praxe, Brno.

[42] Vašíček, O. *Universal OSLC Analysis Adapter - GitLab* [online]. 2020. 2021 [cit. 2021-27-04]. Available at: https://pajda.fit.vutbr.cz/verifit/oslc-generic-analysis.

[43] Vašíček, O. *Universal OSLC Analysis Adapter - Wiki* [online]. 2020. 2021 [cit. 2021-27-04]. Available at: https://pajda.fit.vutbr.cz/verifit/oslc-generic-analysis/-/wikis/home.

[44] Vašíček, O. OSLC Adapter for Software Analysis. In: [Excel@FIT 2021. Brno University of Technology, Faculty of Information Technology.]. 2021 [cit. 2021-27-04]. Available at: http://excel.fit.vutbr.cz/submissions/2021/017/17.pdf.

[45] W3C. *Linked Data* [online]. [cit. 2021-27-04]. Available at: https://www.w3.org/standards/semanticweb/data.

[46] *Triplestore* [online]. [cit. 2021-27-04]. Available at: https://en.wikipedia.org/wiki/Triplestore.

Appendix A

Repository and Usage Guide

The adapter's repository is publicly available at the faculty GitLab [42]:

- Public repository https://pajda.fit.vutbr.cz/verifit/oslc-generic-analysis

For a detailed usage guide which explains configuration options, all input parameters and features, utility scripts etc. please refer to the adapter's Wiky [43].

- Wiky page https://pajda.fit.vutbr.cz/verifit/oslc-generic-analysis/-/wikis/home
- A copy of the Wiky is also included with the adapter in the `tutorials` directory

Particularly the *Tutorial* section which shows all steps of using the adapter including screenshots and examples ready to be copy-pasted.

- Configuration: https://pajda.fit.vutbr.cz/verifit/oslc-generic-analysis/-/wikis/Tutorial/1.-Configuration
- Build and run: https://pajda.fit.vutbr.cz/verifit/oslc-generic-analysis/-/wikis/Tutorial/2.-Build-and-Run
- Running analysis: https://pajda.fit.vutbr.cz/verifit/oslc-generic-analysis/-/wikis/Tutorial/3.-Analysis-Using-SwaggerUI

The *Usage Guide* section which explains all input parameters, analysis tool configuration, and output filter configuration.

- Basics and input parameters: https://pajda.fit.vutbr.cz/verifit/oslc-generic-analysis/-/wikis/Usage-Guide/1.-Basics
- Analysis tool definition: https://pajda.fit.vutbr.cz/verifit/oslc-generic-analysis/-/wikis/Usage-Guide/2.-Analysis-Tool-Definition
- Output filter definition: https://pajda.fit.vutbr.cz/verifit/oslc-generic-analysis/-/wikis/Usage-Guide/3.-Plugin-Output-Filters

And the *Setup Guide* section which briefly introduces how to setup and run the adapter.

- Instalation and configuration: https://pajda.fit.vutbr.cz/verifit/oslc-generic-analysis/-/wikis/Setup-Guide/Installation-and-Configuration
- Running: https://pajda.fit.vutbr.cz/verifit/oslc-generic-analysis/-/wikis/Setup-Guide/Running

Appendix B

Detailed Example of Running Analysis

This appendix contains a simple but complete example of running analysis using the adapter created in this work. The example uses default adapter configuration which means that ports `8080`, `8081`, and `8082` will be used and hostname `localhost`. The example shows how to run an analysis of an SUT using ANaConDA [11] including full XML contents of all resources or responses used in the process for completeness. Unfortunately, most XML files span multiple pages due to their size and that is why they are only included in this appendix.

The example consists of these steps:

1. Creating an SUT
   (a) GET the SUT creation Automation Plan
   (b) Request SUT creation by POSTing an Automation Request to the Compilation adapter
   (c) Retrieve the Automation Result of SUT creation
   (d) Retrieve the created SUT resource
2. Executing analysis
   (a) GET the analysis execution Automation Plan
   (b) Request analysis execution by POSTing an Automation Request to the Analysis adapter
   (c) Retrieve the Automation Result of the analysis

B.1 Creating an SUT

Before an analysis can be executed, an SUT needs to be transferred to the analysis server and compiled. This process consists of four steps: getting an Automation Plan to learn about input parameters, creating an Automation Request to request SUT creation, polling the Automation Result until its finished, and finally getting the created SUT resource.

B.1.1 GET the SUT creation Automation Plan

In order to learn how to create Automation Requests for creating SUTs, clients first need to see what the Automation Plan looks like. The Compilation adapter currently has a single Automation Plan which has identifier `0` (zero). This Automation Plan can be retrieved by sending a GET request to the URI of the Automation Plan. Automation Plan URIs look like this:

```
http://host:port/compilation/services/resources/automationPlans/*ID*
```

To retrieve the SUT creation Automation Plan, send a GET request with an `Accept` header to:

```
http://localhost:8081/compilation/services/resources/automationPlans/0
```

The adapter's response will contain the requested Automation Plan which is described and shown below. In case the Automation Plan is not found, most likely because of an incorrect URI, then the adapter will respond with code `400 Not Found`. The Automation Plan defines all input parameters for Automation Requests using ParameterDefinition resources. The most important parameters for a basic example are: one of the `source*` parameters to choose a way of transferring a SUT to the server (lines 34, 43, 52, 61); `launchCommand` to specify an SUT launch command to be used for analysis (line 90); and `buildCommand` to specify an SUT build command to be used to compile the SUT (line 25). Then there are other functional parameters to control the SUT creation process such as `compile` (line 80) and `unpackZip` (line 70).

```
1  <!-- HEADER: Accept=application/rdf+xml
2       RESPONSE CODE: 200 OK -->
3
4  <?xml version="1.0" encoding="UTF-8"?>
5  <rdf:RDF
6    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
7    xmlns:dcterms="http://purl.org/dc/terms/"
8    xmlns:oslc_data="http://open-services.net/ns/servicemanagement/1.0/"
9    xmlns:oslc="http://open-services.net/ns/core#"
10   xmlns:foaf="http://xmlns.com/foaf/0.1/#"
11   xmlns:fit="http://fit.vutbr.cz/group/verifit/oslc/ns/universal-analysis#"
12   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
13   xmlns:oslc_auto="http://open-services.net/ns/auto#">
14
15   <oslc_auto:AutomationPlan rdf:about="http://localhost:8081/compilation/services/resources/
         automationPlans/0">
16     <dcterms:identifier>0</dcterms:identifier>
17     <dcterms:creator rdf:resource="https://pajda.fit.vutbr.cz/xvasic"/>
18     <dcterms:created rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">
           2021-04-26T18:06:11.694Z</dcterms:created>
19     <dcterms:title rdf:parseType="Literal">SUT Deploy</dcterms:title>
20     <dcterms:description rdf:parseType="Literal">Download and compile an SUT on the server
           so it can be executed later.Use exactly one of the "source.*" parameters.</dcterms:
           description>
21     <dcterms:modified rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">
           2021-04-26T18:06:11.694Z</dcterms:modified>
22
```

```
23      <oslc_auto:parameterDefinition>
24        <oslc_auto:ParameterDefinition>
25          <oslc:name>buildCommand</oslc:name>
26          <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
27          <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
28          <dcterms:description rdf:parseType="Literal">How to build the SUT. The specified
                command will be launched from the root directory of the downloaded SUT.
                Examples: make | ./build.sh | gcc -g -o my_sut. If this command is missing or
                empty then compilation will not be performed (e.g. for static analysis tools)</
                dcterms:description>
29        </oslc_auto:ParameterDefinition>
30      </oslc_auto:parameterDefinition>
31
32      <oslc_auto:parameterDefinition>
33        <oslc_auto:ParameterDefinition>
34          <oslc:name>sourceUrl</oslc:name>
35          <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
36          <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
37          <dcterms:description rdf:parseType="Literal">The SUT will be downloaded from a URL.
                 Example: https://pajda.fit.vutbr.cz/xvasic/oslc-generic-analysis.git</dcterms:
                description>
38        </oslc_auto:ParameterDefinition>
39      </oslc_auto:parameterDefinition>
40
41      <oslc_auto:parameterDefinition>
42        <oslc_auto:ParameterDefinition>
43          <oslc:name>sourceBase64</oslc:name>
44          <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
45          <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
46          <dcterms:description rdf:parseType="Literal">The SUT is encoded in base64 as this
                parameter's value. IMPORTANT the value needs to specify a filename (to match
                with buildCommand). The filename should be on the first line of the value, then the
                base64 encoded file as the second line (separated by a "\n"</dcterms:description>
47        </oslc_auto:ParameterDefinition>
48      </oslc_auto:parameterDefinition>
49
50      <oslc_auto:parameterDefinition>
51        <oslc_auto:ParameterDefinition>
52          <oslc:name>sourceFilePath</oslc:name>
53          <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
54          <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
55          <dcterms:description rdf:parseType="Literal">The SUT will be copied from a path in
                the filesystem.</dcterms:description>
56        </oslc_auto:ParameterDefinition>
57      </oslc_auto:parameterDefinition>
58
59      <oslc_auto:parameterDefinition>
60        <oslc_auto:ParameterDefinition>
61          <oslc:name>sourceGit</oslc:name>
62          <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
63          <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
64          <dcterms:description rdf:parseType="Literal">The SUT will be retrieved from a Git
                repository.</dcterms:description>
65        </oslc_auto:ParameterDefinition>
```

```
66      </oslc_auto:parameterDefinition>
67
68      <oslc_auto:parameterDefinition>
69        <oslc_auto:ParameterDefinition>
70          <oslc:name>unpackZip</oslc:name>
71          <oslc:defaultValue>false</oslc:defaultValue>
72          <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
73          <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#boolean"/>
74          <dcterms:description rdf:parseType="Literal">Set this parameter to "true" to have the
                adapter unpack the SUT using ZIP after fetching it.</dcterms:description>
75        </oslc_auto:ParameterDefinition>
76      </oslc_auto:parameterDefinition>
77
78      <oslc_auto:parameterDefinition>
79        <oslc_auto:ParameterDefinition>
80          <oslc:name>compile</oslc:name>
81          <oslc:defaultValue>true</oslc:defaultValue>
82          <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
83          <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#boolean"/>
84          <dcterms:description rdf:parseType="Literal">Set this parameter to "false" to disable
                SUT compilation.</dcterms:description>
85        </oslc_auto:ParameterDefinition>
86      </oslc_auto:parameterDefinition>
87
88      <oslc_auto:parameterDefinition>
89        <oslc_auto:ParameterDefinition>
90          <oslc:name>launchCommand</oslc:name>
91          <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
92          <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
93          <dcterms:description rdf:parseType="Literal">How to launch the SUT once its build.
                The specified command will be launched from the root directory of the downloaded
                SUT. Examples: make run | ./run.sh | ./my_sut</dcterms:description>
94        </oslc_auto:ParameterDefinition>
95      </oslc_auto:parameterDefinition>
96
97    </oslc_auto:AutomationPlan>
98  </rdf:RDF>
```

B.1.2 Request SUT creation

To request an SUT to be created, a client needs to POST an Automation Request to the adapter's creation factory. The Automation Request creation factory is:

```
http://host:port/compilation/services/resources/createAutomationRequest
```

The full request is described and shown below. An example SUT to be analysed using ANaConDA is a Bank example which performs concurrent transactions on a bank account without proper synchronization and then checks whether the final balance is as expected. The example SUT can be retrieved from a URL hosted at my student website using the `sourceUrl` input parameter (line 16). The downloaded file is a `zip` file so the adapter needs to unpack it first which is controlled by the `unpackZip` input parameter (line 23) set to `true`. The example is compiled using make, so the `buildCommand` (line 30) will be set to `make`;

and the `launchCommand` (line 37) will be set to `./bank`. And the `executesAutomationPlan` property (line 12) needs to contain a link to the Automation Plan to be executed.

```
1  <!-- HEADER: Content-Type=application/rdf+xml, Accept=application/rdf+xml -->
2
3  <?xml version="1.0" encoding="UTF-8"?>
4  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
5    xmlns:dcterms="http://purl.org/dc/terms/"
6    xmlns:oslc="http://open-services.net/ns/core#"
7    xmlns:oslc_auto="http://open-services.net/ns/auto#">
8
9    <oslc_auto:AutomationRequest>
10     <dcterms:title>analysis test</dcterms:title>
11     <dcterms:description>analysis test</dcterms:description>
12     <oslc_auto:executesAutomationPlan rdf:resource="http://localhost:8081/compilation/
           services/resources/automationPlans/0" />
13
14     <oslc_auto:inputParameter>
15       <oslc_auto:ParameterInstance>
16         <oslc:name>sourceUrl</oslc:name>
17         <rdf:value>http://www.stud.fit.vutbr.cz/~xvasic25/bank.zip</rdf:value>
18       </oslc_auto:ParameterInstance>
19     </oslc_auto:inputParameter>
20
21     <oslc_auto:inputParameter>
22       <oslc_auto:ParameterInstance>
23         <oslc:name>unpackZip</oslc:name>
24         <rdf:value>true</rdf:value>
25       </oslc_auto:ParameterInstance>
26     </oslc_auto:inputParameter>
27
28     <oslc_auto:inputParameter>
29       <oslc_auto:ParameterInstance>
30         <oslc:name>buildCommand</oslc:name>
31         <rdf:value>make</rdf:value>
32       </oslc_auto:ParameterInstance>
33     </oslc_auto:inputParameter>
34
35     <oslc_auto:inputParameter>
36       <oslc_auto:ParameterInstance>
37         <oslc:name>launchCommand</oslc:name>
38         <rdf:value>./bank</rdf:value>
39       </oslc_auto:ParameterInstance>
40     </oslc_auto:inputParameter>
41   </oslc_auto:AutomationRequest>
42 </rdf:RDF>
```

The adapter will respond with the same Automation Request resource enriched with more properties and a URI. The response is described and shown below. If the Automation Request submitted by client is not valid, then the adapter will return a response with code `400 Bad Request` and an `oslc:Error` resource with a meaningful error message. The created Automation Request contains all the specified input parameters (lines 27-53) and properties (lines 16-18); and some extra properties added by the adapter (lines 20-25), such as creation and modification tags, a state, and an identifier. The most important property

is usually the `producedAutomationResult` property (line 23) which holds a link to the Automation Result created for this request. The next step is to retrieve this Automation Result using the URI stored in this property.

```
1  <!-- RESPONSE CODE: 201 Created -->
2
3  <?xml version="1.0" encoding="UTF-8"?>
4  <rdf:RDF
5    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
6    xmlns:dcterms="http://purl.org/dc/terms/"
7    xmlns:oslc_data="http://open-services.net/ns/servicemanagement/1.0/"
8    xmlns:oslc="http://open-services.net/ns/core#"
9    xmlns:foaf="http://xmlns.com/foaf/0.1/#"
10   xmlns:fit="http://fit.vutbr.cz/group/verifit/oslc/ns/universal-analysis#"
11   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
12   xmlns:oslc_auto="http://open-services.net/ns/auto#">
13
14   <oslc_auto:AutomationRequest rdf:about="http://localhost:8081/compilation/services/
         resources/automationRequests/1">
15
16     <oslc_auto:executesAutomationPlan rdf:resource="http://localhost:8081/compilation/
           services/resources/automationPlans/0"/>
17     <dcterms:title rdf:parseType="Literal">analysis test</dcterms:title>
18     <dcterms:description rdf:parseType="Literal">analysis test</dcterms:description>
19
20     <dcterms:identifier>1</dcterms:identifier>
21     <oslc_auto:state rdf:resource="http://open-services.net/ns/auto#inProgress"/>
22     <oslc_auto:desiredState rdf:resource="http://open-services.net/ns/auto#complete"/>
23     <oslc_auto:producedAutomationResult rdf:resource="http://localhost:8081/compilation/
           services/resources/automationResults/1"/>
24     <dcterms:modified rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">
           2021-04-26T18:37:36.974Z</dcterms:modified>
25     <dcterms:created rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">
           2021-04-26T18:37:36.974Z</dcterms:created>
26
27     <oslc_auto:inputParameter>
28       <oslc_auto:ParameterInstance>
29         <oslc:name>unpackZip</oslc:name>
30         <rdf:value>true</rdf:value>
31       </oslc_auto:ParameterInstance>
32     </oslc_auto:inputParameter>
33
34     <oslc_auto:inputParameter>
35       <oslc_auto:ParameterInstance>
36         <oslc:name>launchCommand</oslc:name>
37         <rdf:value>./bank</rdf:value>
38       </oslc_auto:ParameterInstance>
39     </oslc_auto:inputParameter>
40
41     <oslc_auto:inputParameter>
42       <oslc_auto:ParameterInstance>
43         <oslc:name>buildCommand</oslc:name>
44         <rdf:value>make</rdf:value>
45       </oslc_auto:ParameterInstance>
```

```
46      </oslc_auto:inputParameter>
47
48      <oslc_auto:inputParameter>
49        <oslc_auto:ParameterInstance>
50          <oslc:name>sourceUrl</oslc:name>
51          <rdf:value>http://www.stud.fit.vutbr.cz/~xvasic25/bank.zip</rdf:value>
52        </oslc_auto:ParameterInstance>
53      </oslc_auto:inputParameter>
54
55    </oslc_auto:AutomationRequest>
56  </rdf:RDF>
```

B.1.3 Retrieve the Automation Result

After an Automation Request was created, clients need to poll for the Automation Result using the URI found in the `producedAutomationResult` property of the created Automation Request. Clients need to poll the result and look at its `state` property so see if the execution finished which is signified by value `complete`. Automation Results which are not yet complete will not contain any Contribution resources so an example of such resource can be skipped. The Automation Result can be retrieved by sending a GET request to its URI. Below is an example of a complete Automation Result. The Automation Result contains all the input parameters (lines 29-55) as well as output parameters (line 59). Output parameters are parameters which the adapter added automatically using their default values. Then there are Contribution resources (lines 64-135) which contain the `stdout` (line 127) and `stderr` (line 66) outputs of the compilation, its `returnCode` (line 118), total `ExecutionTime` (line 109), and `statusMessage`s (line 85) from the adapter. The most important property for the Compilation adapter is the `createdSUT` property (line 17) which holds a link to the SUT resource.

```
1   <!-- HEADER: Accept=application/rdf+xml
2        RESPONSE CODE: 200 OK -->
3
4   <?xml version="1.0" encoding="UTF-8"?>
5   <rdf:RDF
6     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
7     xmlns:dcterms="http://purl.org/dc/terms/"
8     xmlns:oslc_data="http://open-services.net/ns/servicemanagement/1.0/"
9     xmlns:oslc="http://open-services.net/ns/core#"
10    xmlns:foaf="http://xmlns.com/foaf/0.1/#"
11    xmlns:fit="http://fit.vutbr.cz/group/verifit/oslc/ns/universal-analysis#"
12    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
13    xmlns:oslc_auto="http://open-services.net/ns/auto#">
14
15    <oslc_auto:AutomationResult rdf:about="http://localhost:8081/compilation/services/resources
          /automationResults/1">
16
17      <fit:createdSUT rdf:resource="http://localhost:8081/compilation/services/resources/sUTs/1"
            />
18      <oslc_auto:verdict rdf:resource="http://open-services.net/ns/auto#passed"/>
19      <oslc_auto:state rdf:resource="http://open-services.net/ns/auto#complete"/>
20
21      <dcterms:identifier>1</dcterms:identifier>
```

```
22      <oslc_auto:desiredState rdf:resource="http://open-services.net/ns/auto#complete"/>
23      <dcterms:created rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">
            2021-04-26T18:52:18.824Z</dcterms:created>
24      <dcterms:modified rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">
            2021-04-26T18:52:19.129Z</dcterms:modified>
25      <dcterms:title rdf:parseType="Literal">Result - analysis test</dcterms:title>
26      <oslc_auto:producedByAutomationRequest rdf:resource="http://localhost:8081/compilation/
            services/resources/automationRequests/1"/>
27      <oslc_auto:reportsOnAutomationPlan rdf:resource="http://localhost:8081/compilation/
            services/resources/automationPlans/0"/>
28
29      <oslc_auto:inputParameter>
30        <oslc_auto:ParameterInstance>
31          <oslc:name>unpackZip</oslc:name>
32          <rdf:value>true</rdf:value>
33        </oslc_auto:ParameterInstance>
34      </oslc_auto:inputParameter>
35
36      <oslc_auto:inputParameter>
37        <oslc_auto:ParameterInstance>
38          <oslc:name>buildCommand</oslc:name>
39          <rdf:value>make</rdf:value>
40        </oslc_auto:ParameterInstance>
41      </oslc_auto:inputParameter>
42
43      <oslc_auto:inputParameter>
44        <oslc_auto:ParameterInstance>
45          <oslc:name>launchCommand</oslc:name>
46          <rdf:value>./bank</rdf:value>
47        </oslc_auto:ParameterInstance>
48      </oslc_auto:inputParameter>
49
50      <oslc_auto:inputParameter>
51        <oslc_auto:ParameterInstance>
52          <oslc:name>sourceUrl</oslc:name>
53          <rdf:value>http://www.stud.fit.vutbr.cz/~xvasic25/bank.zip</rdf:value>
54        </oslc_auto:ParameterInstance>
55      </oslc_auto:inputParameter>
56
57      <oslc_auto:outputParameter>
58        <oslc_auto:ParameterInstance>
59          <oslc:name>compile</oslc:name>
60          <rdf:value>true</rdf:value>
61        </oslc_auto:ParameterInstance>
62      </oslc_auto:outputParameter>
63
64      <oslc_auto:contribution>
65        <oslc_auto:Contribution>
66          <dcterms:title rdf:parseType="Literal">stderr</dcterms:title>
67          <rdf:value></rdf:value>
68          <fit:filePath>/path/to/universal-analysis-adapter/compilation/SUT/3/.adapter/
                stderr_compilation_1</fit:filePath>
69          <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
```

```
70          <dcterms:description rdf:parseType="Literal">Error output of the compilation.</
                dcterms:description>
71        </oslc_auto:Contribution>
72      </oslc_auto:contribution>
73
74      <oslc_auto:contribution>
75        <oslc_auto:Contribution>
76          <dcterms:title rdf:parseType="Literal">Fetching Output</dcterms:title>
77          <rdf:value># currently only shows error messages</rdf:value>
78          <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
79          <dcterms:description rdf:parseType="Literal">Output of the program fetching process.<
                /dcterms:description>
80        </oslc_auto:Contribution>
81      </oslc_auto:contribution>
82
83      <oslc_auto:contribution>
84        <oslc_auto:Contribution>
85          <dcterms:title rdf:parseType="Literal">statusMessage</dcterms:title>
86          <rdf:value>SUT fetch successful
87  Executing: make
88  as: /bin/bash ./.adapter/exec_compilation_1.sh "make"
89  In dir: SUT/3
90  Compilation completed successfully
91  SUT resource created
92  </rdf:value>
93          <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
94          <dcterms:description rdf:parseType="Literal">Status messages from the adapter about
                the execution.</dcterms:description>
95        </oslc_auto:Contribution>
96      </oslc_auto:contribution>
97
98      <oslc_auto:contribution>
99        <oslc_auto:Contribution>
100         <dcterms:title rdf:parseType="Literal">SUT</dcterms:title>
101         <rdf:value>http://localhost:8081/compilation/services/resources/sUTs/1</rdf:value>
102         <oslc:valueType rdf:resource="http://localhost:8081/compilation/services/resourceShapes
                /sUT"/>
103         <dcterms:description rdf:parseType="Literal">Created SUT resource. Also linked to by
                the createdSUT property.</dcterms:description>
104       </oslc_auto:Contribution>
105     </oslc_auto:contribution>
106
107     <oslc_auto:contribution>
108       <oslc_auto:Contribution>
109         <dcterms:title rdf:parseType="Literal">executionTime</dcterms:title>
110         <rdf:value>200</rdf:value>
111         <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
112         <dcterms:description rdf:parseType="Literal">Total execution time of the analysis in
                milliseconds.</dcterms:description>
113       </oslc_auto:Contribution>
114     </oslc_auto:contribution>
115
116     <oslc_auto:contribution>
117       <oslc_auto:Contribution>
```

```
118           <dcterms:title rdf:parseType="Literal">returnCode</dcterms:title>
119           <rdf:value>0</rdf:value>
120           <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#integer"/>
121           <dcterms:description rdf:parseType="Literal">Return code of the execution. If non-zero
                  , then the verdict will be #failed.</dcterms:description>
122         </oslc_auto:Contribution>
123       </oslc_auto:contribution>
124
125       <oslc_auto:contribution>
126         <oslc_auto:Contribution>
127           <dcterms:title rdf:parseType="Literal">stdout</dcterms:title>
128           <rdf:value>g++ -Wall -ansi -pedantic -g -c src/bank.cpp
129 g++ -lm -pthread -o bank bank.o
130 </rdf:value>
131           <fit:filePath>/path/to/universal-analysis-adapter/compilation/SUT/3/.adapter/
                  stdout_compilation_1</fit:filePath>
132           <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
133           <dcterms:description rdf:parseType="Literal">Standard output of the compilation.</
                  dcterms:description>
134         </oslc_auto:Contribution>
135       </oslc_auto:contribution>
136
137     </oslc_auto:AutomationResult>
138 </rdf:RDF>
```

B.1.4 Retrieve the created SUT

The last step of the SUT creation process is retrieving the actual created SUT resource by sending a `GET` request to the URI found in the `createdSUT` property of the received Automation Result. This step can technically be skipped because only the SUT URI is needed to execute analysis, not the actual contents of the SUT resource. The SUT resource will contain the specified launch command (line 18) and build command (line 19), a path to its directory (line 21), a boolean flag about being compiled (line 20), and other properties.

```
1  <!-- HEADER: Accept=application/rdf+xml
2           RESPONSE CODE: 200 OK -->
3
4  <?xml version="1.0" encoding="UTF-8"?>
5  <rdf:RDF
6    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
7    xmlns:dcterms="http://purl.org/dc/terms/"
8    xmlns:oslc_data="http://open-services.net/ns/servicemanagement/1.0/"
9    xmlns:oslc="http://open-services.net/ns/core#"
10   xmlns:foaf="http://xmlns.com/foaf/0.1/#"
11   xmlns:fit="http://fit.vutbr.cz/group/verifit/oslc/ns/universal-analysis#"
12   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
13   xmlns:oslc_auto="http://open-services.net/ns/auto#">
14
15   <fit:SUT rdf:about="http://localhost:8081/compilation/services/resources/sUTs/1">
16
17     <dcterms:identifier>1</dcterms:identifier>
18     <fit:launchCommand rdf:parseType="Literal">./bank</fit:launchCommand>
19     <fit:buildCommand rdf:parseType="Literal">make</fit:buildCommand>
```

```
20      <fit:compiled rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean">true</fit:
            compiled>
21      <fit:SUTdirectoryPath rdf:parseType="Literal">/path/to/universal-analysis-adapter/
            compilation/SUT/1</fit:SUTdirectoryPath>
22
23      <dcterms:title rdf:parseType="Literal">SUT - analysis test</dcterms:title>
24      <oslc_auto:producedByAutomationRequest rdf:resource="http://localhost:8081/compilation/
            services/resources/automationRequests/1"/>
25      <dcterms:modified rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">
            2021-04-26T18:52:19.111Z</dcterms:modified>
26      <dcterms:created rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">
            2021-04-26T18:52:19.111Z</dcterms:created>
27
28    </fit:SUT>
29  </rdf:RDF>
```

B.2 Executing analysis

After an SUT was created and the client knows the URI of the SUT, then analysis can be executed using the Analysis adapter. The process of executing analysis consists of these steps: getting an Automation Plan for an analysis tool, creating an Automation Request to execute analysis, and getting the produced Automation Result. The overall process is very similar to the SUT creation process because it also uses the Automation domain. From the user input perspective the only differences are endpoint URIs, input parameters used with Automation Requests, and different Contributions in the Automation Result.

B.2.1 GET the analysis execution Automation Plan

In order to learn how to create Automation Requests for executing analysis, clients first need to see what the Automation Plan looks like. The Analysis adapter contains Automation Plans for different analysis tools based on user configuration. In this example we will be using ANaConDA which has an Automation Plan defined with identifier `anaconda`. This Automation Plan can be retrieved by sending a GET request to the URI of the Automation Plan. Automation Plan URIs look like this:

```
http://host:port/analysis/services/resources/automationPlans/*ID*
```

To retrieve the SUT creation Automation Plan, send a GET request with an `Accept` header to:

```
http://localhost:8080/analysis/services/resources/automationPlans/anaconda
```

The adapter's response will contain the requested Automation Plan which is described and shown below.

In case the Automation Plan is not found, most likely because of an incorrect URI, then the adapter will respond with code `400 Not Found`. The Automation Plan defines all input parameters for Automation Requests using ParameterDefinition resources. Input parameters specific to ANaConDA's interface all have a `commandlinePosition` property and can be found towards the start of the request (lines 25-149), and input parameters common to the adapter do not have `commandlinePosition` properties and can be found after the ANaConDA parameters (lines 151-256).

ANaConDA has a number of optional non-positional parameters which are all set to be placed on command-line position `1`. There is a number of flags without values represented as Parameter Definitions with value prefixes and allowed values set to empty, which makes it so that clients can only ever specify these parameters with no or empty values and the adapter then only places their value prefixes on the command line. These are `help` (line 38), `verbose` (line 49), `profile` (line 86), and `time` (line 98). Then, there is a number of non-positional parameters which need a value represented by Parameter Definitions with value prefixes and some of them have a restricted set of allowed values. These are `config` (line 27), `threads` (line 78), `run-type` (line 61). The last three arguments for the ANaConDA interface are positional arguments with their own command-line positions. These are `analyser` (line 110) with specific allowed values, `launchSUT` (line 131) a special parameter for placing the SUT launch command at the command-line, and `executionParameters` (line 142).

The rest of Parameter Definitions are parameters included in all Automation Plans by the adapter. These are `SUT` (line 242) a link to the SUT resource to analyse, `timeout` (line 153) a time limit for execution, `afterCommand` (line 163) a command to execute before analysis, `beforeCommand` (line 233) a command to execute after analysis, `confFile` (line 172) configuration file to be created in the SUT directory, `confDir` (line 251) configuration directory to be created in the SUT directory, `outputFilter` (line 181) output filter selection, `envVariable` (line 194) for setting environmental variables, `outputFileRegex` (line 203) for matching new or modified files as contributions, `zipOutputs` (line 213) for creating a zip of all contribution files, and `toolCommand` (line 223) for toggling use of the analysis tool launch command.

```
1  <!-- HEADER: Accept=application/rdf+xml
2  RESPONSE CODE: 200 OK -->
3
4  <?xml version="1.0" encoding="UTF-8"?>
5  <rdf:RDF
6    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
7    xmlns:dcterms="http://purl.org/dc/terms/"
8    xmlns:oslc_data="http://open-services.net/ns/servicemanagement/1.0/"
9    xmlns:oslc="http://open-services.net/ns/core#"
10   xmlns:foaf="http://xmlns.com/foaf/0.1/#"
11   xmlns:fit="http://fit.vutbr.cz/group/verifit/oslc/ns/universal-analysis#"
12   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
13   xmlns:oslc_auto="http://open-services.net/ns/auto#">
14
15   <oslc_auto:AutomationPlan rdf:about="http://localhost:8080/analysis/services/resources/
         automationPlans/anaconda">
16
17     <dcterms:identifier>anaconda</dcterms:identifier>
18     <dcterms:title rdf:parseType="Literal">ANaConDA</dcterms:title>
19     <oslc_auto:usesExecutionEnvironment rdf:resource="https://pajda.fit.vutbr.cz/anaconda/
           anaconda"/>
20     <dcterms:description rdf:parseType="Literal">Analyse an SUT using ANaConDA</dcterms:
           description>
21     <dcterms:creator rdf:resource="https://pajda.fit.vutbr.cz/xvasic"/>
22     <dcterms:modified rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">
           2021-04-26T18:06:22.259Z</dcterms:modified>
23     <dcterms:created rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">
           2021-04-26T18:06:22.259Z</dcterms:created>
24
```

```
25      <oslc_auto:parameterDefinition>
26        <oslc_auto:ParameterDefinition>
27          <oslc:name>config</oslc:name>
28          <fit:commandlinePosition rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1
                </fit:commandlinePosition>
29          <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
30          <fit:valuePrefix>--config </fit:valuePrefix>
31          <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
32          <dcterms:description rdf:parseType="Literal">A path to a directory containing
                ANaConDA settings</dcterms:description>
33        </oslc_auto:ParameterDefinition>
34      </oslc_auto:parameterDefinition>
35
36      <oslc_auto:parameterDefinition>
37        <oslc_auto:ParameterDefinition>
38          <oslc:name>help</oslc:name>
39          <fit:commandlinePosition rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1
                </fit:commandlinePosition>
40          <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
41          <oslc:allowedValue></oslc:allowedValue>
42          <fit:valuePrefix>--help</fit:valuePrefix>
43          <dcterms:description rdf:parseType="Literal">Print the script usage</dcterms:
                description>
44        </oslc_auto:ParameterDefinition>
45      </oslc_auto:parameterDefinition>
46
47      <oslc_auto:parameterDefinition>
48        <oslc_auto:ParameterDefinition>
49          <oslc:name>verbose</oslc:name>
50          <fit:commandlinePosition rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1
                </fit:commandlinePosition>
51          <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
52          <oslc:allowedValue></oslc:allowedValue>
53          <fit:valuePrefix>--verbose </fit:valuePrefix>
54          <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#boolean"/>
55          <dcterms:description rdf:parseType="Literal">Print detailed information about what the
                script is doing</dcterms:description>
56        </oslc_auto:ParameterDefinition>
57      </oslc_auto:parameterDefinition>
58
59      <oslc_auto:parameterDefinition>
60        <oslc_auto:ParameterDefinition>
61          <oslc:name>run-type</oslc:name>
62          <fit:valuePrefix>--run-type </fit:valuePrefix>
63          <fit:commandlinePosition rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1
                </fit:commandlinePosition>
64          <dcterms:description rdf:parseType="Literal">Execute the program in ANaConDA, PIN
                or no framework (native run)</dcterms:description>
65          <oslc:allowedValue>anaconda</oslc:allowedValue>
66          <oslc:allowedValue>pin</oslc:allowedValue>
67          <oslc:allowedValue>native</oslc:allowedValue>
68          <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
69          <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
70        </oslc_auto:ParameterDefinition>
```

```
71      </oslc_auto:parameterDefinition>
72
73      <oslc_auto:parameterDefinition>
74        <oslc_auto:ParameterDefinition>
75          <oslc:name>threads</oslc:name>
76          <fit:commandlinePosition rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1
                </fit:commandlinePosition>
77          <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
78          <fit:valuePrefix>--threads </fit:valuePrefix>
79          <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
80          <dcterms:description rdf:parseType="Literal">A number of threads the analysed
                program should utilize</dcterms:description>
81        </oslc_auto:ParameterDefinition>
82      </oslc_auto:parameterDefinition>
83
84      <oslc_auto:parameterDefinition>
85        <oslc_auto:ParameterDefinition>
86          <oslc:name>profile</oslc:name>
87          <fit:commandlinePosition rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1
                </fit:commandlinePosition>
88          <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
89          <oslc:allowedValue></oslc:allowedValue>
90          <fit:valuePrefix>--profile </fit:valuePrefix>
91          <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#boolean"/>
92          <dcterms:description rdf:parseType="Literal">Profile the program being analysed using
                the oprofile profiler</dcterms:description>
93        </oslc_auto:ParameterDefinition>
94      </oslc_auto:parameterDefinition>
95
96      <oslc_auto:parameterDefinition>
97        <oslc_auto:ParameterDefinition>
98          <oslc:name>time</oslc:name>
99          <fit:commandlinePosition rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1
                </fit:commandlinePosition>
100         <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
101         <oslc:allowedValue></oslc:allowedValue>
102         <fit:valuePrefix>--time </fit:valuePrefix>
103         <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#boolean"/>
104         <dcterms:description rdf:parseType="Literal">Measure the execution time of the
                program being analysed</dcterms:description>
105       </oslc_auto:ParameterDefinition>
106     </oslc_auto:parameterDefinition>
107
108     <oslc_auto:parameterDefinition>
109       <oslc_auto:ParameterDefinition>
110         <oslc:name>analyser</oslc:name>
111         <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
112         <oslc:allowedValue>simple-contract-validator</oslc:allowedValue>
113         <oslc:allowedValue>statistics-collector</oslc:allowedValue>
114         <oslc:allowedValue>eraser</oslc:allowedValue>
115         <oslc:allowedValue>data-printer</oslc:allowedValue>
116         <oslc:allowedValue>tx-monitor</oslc:allowedValue>
117         <oslc:allowedValue>contract-validator</oslc:allowedValue>
118         <oslc:allowedValue>fasttrack2</oslc:allowedValue>
```

```
119         <oslc:allowedValue>atomrace</oslc:allowedValue>
120         <oslc:allowedValue>hldr-detector</oslc:allowedValue>
121         <oslc:allowedValue>data-validator</oslc:allowedValue>
122         <oslc:allowedValue>param-contract-validator</oslc:allowedValue>
123         <dcterms:description rdf:parseType="Literal">Specify what analyser to use</dcterms:
                description>
124         <oslc:occurs rdf:resource="http://open-services.net/ns/core#Exactly-one"/>
125         <fit:commandlinePosition rdf:datatype="http://www.w3.org/2001/XMLSchema#int">2
                </fit:commandlinePosition>
126       </oslc_auto:ParameterDefinition>
127     </oslc_auto:parameterDefinition>
128
129     <oslc_auto:parameterDefinition>
130       <oslc_auto:ParameterDefinition>
131         <oslc:name>launchSUT</oslc:name>
132         <oslc:readOnly rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean">true</
                oslc:readOnly>
133         <oslc:defaultValue>True</oslc:defaultValue>
134         <fit:commandlinePosition rdf:datatype="http://www.w3.org/2001/XMLSchema#int">3
                </fit:commandlinePosition>
135         <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
136         <dcterms:description rdf:parseType="Literal">If set to true, This parameter definitions
                tells the Automation Plan to place the SUT launch command at this command line
                position</dcterms:description>
137       </oslc_auto:ParameterDefinition>
138     </oslc_auto:parameterDefinition>
139
140     <oslc_auto:parameterDefinition>
141       <oslc_auto:ParameterDefinition>
142         <oslc:name>executionParameters</oslc:name>
143         <oslc:defaultValue></oslc:defaultValue>
144         <fit:commandlinePosition rdf:datatype="http://www.w3.org/2001/XMLSchema#int">4
                </fit:commandlinePosition>
145         <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
146         <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
147         <dcterms:description rdf:parseType="Literal">Set the execution parameters for the
                analyzed program. Write down all parameters as you would in a console.</dcterms:
                description>
148       </oslc_auto:ParameterDefinition>
149     </oslc_auto:parameterDefinition>
150
151     <oslc_auto:parameterDefinition>
152       <oslc_auto:ParameterDefinition>
153         <oslc:name>timeout</oslc:name>
154         <oslc:defaultValue>0</oslc:defaultValue>
155         <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
156         <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#integer"/>
157         <dcterms:description rdf:parseType="Literal">Timeout for the analysis. Zero means no
                timeout.</dcterms:description>
158       </oslc_auto:ParameterDefinition>
159     </oslc_auto:parameterDefinition>
160
161     <oslc_auto:parameterDefinition>
162       <oslc_auto:ParameterDefinition>
```

```
163         <oslc:name>afterCommand</oslc:name>
164         <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
165         <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
166         <dcterms:description rdf:parseType="Literal">A command to run just after analysis is
                executed.</dcterms:description>
167       </oslc_auto:ParameterDefinition>
168     </oslc_auto:parameterDefinition>
169
170     <oslc_auto:parameterDefinition>
171       <oslc_auto:ParameterDefinition>
172         <oslc:name>confFile</oslc:name>
173         <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-many"/>
174         <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
175         <dcterms:description rdf:parseType="Literal">Creates a configuration file inside of the
                SUT directory before running analysis. Can be used multiple times create multiple
                conf files.Format for this parameter: "conf_file_name\nconf_file_txt_contents"</
                dcterms:description>
176       </oslc_auto:ParameterDefinition>
177     </oslc_auto:parameterDefinition>
178
179     <oslc_auto:parameterDefinition>
180       <oslc_auto:ParameterDefinition>
181         <oslc:name>outputFilter</oslc:name>
182         <oslc:defaultValue>default</oslc:defaultValue>
183         <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
184         <oslc:allowedValue>default</oslc:allowedValue>
185         <oslc:allowedValue>removeAllFileValues</oslc:allowedValue>
186         <oslc:allowedValue>AnacondaRaceDetection</oslc:allowedValue>
187         <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
188         <dcterms:description rdf:parseType="Literal">Use this parameter to select which output
                filter should be used to processContributions of this Automation Request.
                AllowedValues are loaded based on defined PluginFilters.</dcterms:description>
189       </oslc_auto:ParameterDefinition>
190     </oslc_auto:parameterDefinition>
191
192     <oslc_auto:parameterDefinition>
193       <oslc_auto:ParameterDefinition>
194         <oslc:name>envVariable</oslc:name>
195         <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-many"/>
196         <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
197         <dcterms:description rdf:parseType="Literal">Environment variable to be set for
                execution. Can be used multiple times for multiple variables.Expected value format: "
                variable_name\nvariable_value"</dcterms:description>
198       </oslc_auto:ParameterDefinition>
199     </oslc_auto:parameterDefinition>
200
201     <oslc_auto:parameterDefinition>
202       <oslc_auto:ParameterDefinition>
203         <oslc:name>outputFileRegex</oslc:name>
204         <oslc:defaultValue>.^</oslc:defaultValue>
205         <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
206         <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
```

```
207         <dcterms:description rdf:parseType="Literal">Files that change during execution and
                match this regex will be added as contributions to the Automation Result. The regex
                 needs to match the whole filename.</dcterms:description>
208       </oslc_auto:ParameterDefinition>
209     </oslc_auto:parameterDefinition>
210
211     <oslc_auto:parameterDefinition>
212       <oslc_auto:ParameterDefinition>
213         <oslc:name>zipOutputs</oslc:name>
214         <oslc:defaultValue>false</oslc:defaultValue>
215         <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
216         <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#boolean"/>
217         <dcterms:description rdf:parseType="Literal">If set to true, then all file contributions
                will be ZIPed and provided as a single zip contribution</dcterms:description>
218       </oslc_auto:ParameterDefinition>
219     </oslc_auto:parameterDefinition>
220
221     <oslc_auto:parameterDefinition>
222       <oslc_auto:ParameterDefinition>
223         <oslc:name>toolCommand</oslc:name>
224         <oslc:defaultValue>true</oslc:defaultValue>
225         <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
226         <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#boolean"/>
227         <dcterms:description rdf:parseType="Literal">Used to omit the analysis tool launch
                command while executing analysis. True means the tool will be used and False means
                 the tool command will not be used. (eg. "./tool ./sut args" vs "/sut args").</
                dcterms:description>
228       </oslc_auto:ParameterDefinition>
229     </oslc_auto:parameterDefinition>
230
231     <oslc_auto:parameterDefinition>
232       <oslc_auto:ParameterDefinition>
233         <oslc:name>beforeCommand</oslc:name>
234         <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
235         <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
236         <dcterms:description rdf:parseType="Literal">A command to run just before analysis is
                executed.</dcterms:description>
237       </oslc_auto:ParameterDefinition>
238     </oslc_auto:parameterDefinition>
239
240     <oslc_auto:parameterDefinition>
241       <oslc_auto:ParameterDefinition>
242         <oslc:name>SUT</oslc:name>
243         <oslc:occurs rdf:resource="http://open-services.net/ns/core#Exactly-one"/>
244         <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
245         <dcterms:description rdf:parseType="Literal">Refference to an SUT resource to analyse.
                SUTs are created using the compilation provider.</dcterms:description>
246       </oslc_auto:ParameterDefinition>
247     </oslc_auto:parameterDefinition>
248
249     <oslc_auto:parameterDefinition>
250       <oslc_auto:ParameterDefinition>
251         <oslc:name>confDir</oslc:name>
252         <oslc:occurs rdf:resource="http://open-services.net/ns/core#Zero-or-One"/>
```

```
253         <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
254         <dcterms:description rdf:parseType="Literal">Creates a configuration directory inside of
                the SUT directory before running analysis from a base64 encoded string.Format for
                this parameter: "path_to_unzip_to\nbase64_encoded_zip_file"</dcterms:
                description>
255       </oslc_auto:ParameterDefinition>
256     </oslc_auto:parameterDefinition>
257
258   </oslc_auto:AutomationPlan>
259 </rdf:RDF>
```

B.2.2 Request analysis execution

To request an analysis using ANaConDA to be executed, a client needs to POST an Automation Request to the adapter's creation factory. The Automation Request creation factory is:

`http://host:port/analysis/services/resources/createAutomationRequest`

An example analysis using ANaConDA on the previously created Bank example uses `atomrace` [18] to detect data races in the bank execution. The full request is described and shown below. Important input parameters used specific to ANaConDA are: `analyser` (line 23) to specify an analyser to use, and `executionParameters` (line 30) to pass parameters to the SUT Common adapter input parameters used are: `outputFilter` (line 16) to select a custom output filter to create ANaConDA specific Contributions, and `SUT` (line 37) which holds a link to the previously created SUT to be analysed. The adapter will respond with the same Automation Request resource enriched with more properties and a URI.

```
1  <!-- HEADER: Content-Type=application/rdf+xml, Accept=application/rdf+xml -->
2
3  <?xml version="1.0" encoding="utf-8" ?>
4  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
5    xmlns:dcterms="http://purl.org/dc/terms/"
6    xmlns:oslc="http://open-services.net/ns/core#"
7    xmlns:oslc_auto="http://open-services.net/ns/auto#">
8
9    <oslc_auto:AutomationRequest>
10     <dcterms:title>Anaconda atomarace bank analysis</dcterms:title>
11     <dcterms:description>...</dcterms:description>
12     <oslc_auto:executesAutomationPlan rdf:resource="http://localhost:8080/analysis/services/
           resources/automationPlans/anaconda" />
13
14   <oslc_auto:inputParameter>
15     <oslc_auto:ParameterInstance>
16       <oslc:name>outputFilter</oslc:name>
17       <rdf:value>AnacondaRaceDetection</rdf:value>
18     </oslc_auto:ParameterInstance>
19   </oslc_auto:inputParameter>
20
21   <oslc_auto:inputParameter>
22     <oslc_auto:ParameterInstance>
23       <oslc:name>analyser</oslc:name>
24       <rdf:value>atomrace</rdf:value>
```

```
25      </oslc_auto:ParameterInstance>
26    </oslc_auto:inputParameter>
27
28    <oslc_auto:inputParameter>
29      <oslc_auto:ParameterInstance>
30        <oslc:name>executionParameters</oslc:name>
31        <rdf:value>"Hello World!"</rdf:value>
32      </oslc_auto:ParameterInstance>
33    </oslc_auto:inputParameter>
34
35    <oslc_auto:inputParameter>
36      <oslc_auto:ParameterInstance>
37        <oslc:name>SUT</oslc:name>
38        <rdf:value>http://localhost:8081/compilation/services/resources/sUTs/1</rdf:value>
39      </oslc_auto:ParameterInstance>
40    </oslc_auto:inputParameter>
41
42    </oslc_auto:AutomationRequest>
43  </rdf:RDF>
```

The adapter's response is described and shown below. If the Automation Request submitted by client is not valid, then the adapter will return a response with code `400 Bad Request` and an `oslc:Error` resource with a meaningful error message. The created Automation Request contains all the specified input parameters (lines 27-53) and properties (lines 16-18), and then some properties added by the adapter (lines 20-25) such as creation and modification tags, a state, and an identifier. The most important property is usually the `producedAutomationResult` property (line 23) which holds a link to the Automation Result created for this request. The next step is to retrieve this Automation Result using the URI stored in this property.

```
1   <!-- RESPONSE CODE: 201 Created -->
2
3   <?xml version="1.0" encoding="UTF-8"?>
4   <rdf:RDF
5     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
6     xmlns:dcterms="http://purl.org/dc/terms/"
7     xmlns:oslc_data="http://open-services.net/ns/servicemanagement/1.0/"
8     xmlns:oslc="http://open-services.net/ns/core#"
9     xmlns:foaf="http://xmlns.com/foaf/0.1/#"
10    xmlns:fit="http://fit.vutbr.cz/group/verifit/oslc/ns/universal-analysis#"
11    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
12    xmlns:oslc_auto="http://open-services.net/ns/auto#">
13
14    <oslc_auto:AutomationRequest rdf:about="http://localhost:8080/analysis/services/resources/
          automationRequests/1">
15
16      <dcterms:title rdf:parseType="Literal">Anaconda atomarace bank analysis</dcterms:title>
17      <dcterms:description rdf:parseType="Literal">...</dcterms:description>
18      <oslc_auto:executesAutomationPlan rdf:resource="http://localhost:8080/analysis/services/
            resources/automationPlans/anaconda"/>
19
20      <dcterms:identifier>1</dcterms:identifier>
21      <oslc_auto:state rdf:resource="http://open-services.net/ns/auto#inProgress"/>
22      <oslc_auto:desiredState rdf:resource="http://open-services.net/ns/auto#complete"/>
```

```
23      <oslc_auto:producedAutomationResult rdf:resource="http://localhost:8080/analysis/services
            /resources/automationResults/1"/>
24      <dcterms:modified rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">
            2021-04-26T19:44:07.599Z</dcterms:modified>
25      <dcterms:created rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">
            2021-04-26T19:44:07.599Z</dcterms:created>
26
27      <oslc_auto:inputParameter>
28        <oslc_auto:ParameterInstance>
29          <oslc:name>SUT</oslc:name>
30          <rdf:value>http://localhost:8081/compilation/services/resources/sUTs/1</rdf:value>
31        </oslc_auto:ParameterInstance>
32      </oslc_auto:inputParameter>
33
34      <oslc_auto:inputParameter>
35        <oslc_auto:ParameterInstance>
36          <oslc:name>analyser</oslc:name>
37          <rdf:value>atomrace</rdf:value>
38        </oslc_auto:ParameterInstance>
39      </oslc_auto:inputParameter>
40
41      <oslc_auto:inputParameter>
42        <oslc_auto:ParameterInstance>
43          <oslc:name>outputFilter</oslc:name>
44          <rdf:value>AnacondaRaceDetection</rdf:value>
45        </oslc_auto:ParameterInstance>
46      </oslc_auto:inputParameter>
47
48      <oslc_auto:inputParameter>
49        <oslc_auto:ParameterInstance>
50          <oslc:name>executionParameters</oslc:name>
51          <rdf:value>"Hello World!"</rdf:value>
52        </oslc_auto:ParameterInstance>
53      </oslc_auto:inputParameter>
54
55    </oslc_auto:AutomationRequest>
56  </rdf:RDF>
```

B.2.3 Retrieve the Automation Result

After an Automation Request was created, clients need to poll for the Automation Result using the URI found in the `producedAutomationResult` property of the created Automation Request. Clients need to poll the result and look at its `state` property so see if the execution finished which is signified by value `complete`. Automation Results which are not yet complete will not contain any Contribution resources so there is no need to shows an example. Here is an example of a complete Automation Result instead. The Automation Result can be retrieved by sending a GET request to its URI. The result is described and shown below. The Automation Result contains all the input parameters (lines 28-61) as well as output parameters (lines 63-89). Output parameters are parameters which the adapter added automatically using their default values. Then, there are Contribution resources (lines 91-187) which contain the `stdout` (line 93) and `stderr` (line 141) outputs of the analysis, its `returnCode` (line 130), total `executionTime` (line 153), and `statusMessages`

(line 164) from the adapter. The status message includes the exact string executed by the adapter which is useful in case the execution failed due to an invalid executed string. The `vedict` (line 17) property will be `passed` or `failed` based on the return code of the analysis, or it can be `error` in case of exceptions during the execution process in the adapter. The custom output filter used for ANaConDA, created a custom Contribution called `DataRaceDetected` (line 180) with a value of `true` which holds the semantic result of the executed analysis as defined by clients during configuration.

```
1  <!-- HEADER: Accept=application/rdf+xml
2  RESPONSE CODE: 200 OK -->
3
4  <?xml version="1.0" encoding="UTF-8"?>
5  <rdf:RDF
6    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
7    xmlns:dcterms="http://purl.org/dc/terms/"
8    xmlns:oslc_data="http://open-services.net/ns/servicemanagement/1.0/"
9    xmlns:oslc="http://open-services.net/ns/core#"
10   xmlns:foaf="http://xmlns.com/foaf/0.1/#"
11   xmlns:fit="http://fit.vutbr.cz/group/verifit/oslc/ns/universal-analysis#"
12   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
13   xmlns:oslc_auto="http://open-services.net/ns/auto#">
14
15   <oslc_auto:AutomationResult rdf:about="http://localhost:8080/analysis/services/resources/
         automationResults/1">
16
17     <oslc_auto:verdict rdf:resource="http://open-services.net/ns/auto#passed"/>
18     <oslc_auto:state rdf:resource="http://open-services.net/ns/auto#complete"/>
19
20     <dcterms:identifier>1</dcterms:identifier>
21     <oslc_auto:desiredState rdf:resource="http://open-services.net/ns/auto#complete"/>
22     <dcterms:title rdf:parseType="Literal">Result - Anaconda atomarace bank analysis</
           dcterms:title>
23     <oslc_auto:reportsOnAutomationPlan rdf:resource="http://localhost:8080/analysis/services/
           resources/automationPlans/anaconda"/>
24     <dcterms:created rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">
           2021-04-26T19:53:52.925Z</dcterms:created>
25     <dcterms:modified rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">
           2021-04-26T19:53:53.986Z</dcterms:modified>
26     <oslc_auto:producedByAutomationRequest rdf:resource="http://localhost:8080/analysis/
           services/resources/automationRequests/1"/>
27
28     <oslc_auto:inputParameter>
29       <oslc_auto:ParameterInstance>
30         <oslc:name>executionParameters</oslc:name>
31         <rdf:value>"Hello World!"</rdf:value>
32       </oslc_auto:ParameterInstance>
33     </oslc_auto:inputParameter>
34
35     <oslc_auto:inputParameter>
36       <oslc_auto:ParameterInstance>
37         <oslc:name>SUT</oslc:name>
38         <rdf:value>http://localhost:8081/compilation/services/resources/sUTs/1</rdf:value>
39       </oslc_auto:ParameterInstance>
40     </oslc_auto:inputParameter>
```

```
41
42      <oslc_auto:inputParameter>
43        <oslc_auto:ParameterInstance>
44          <oslc:name>analyser</oslc:name>
45          <rdf:value>atomrace</rdf:value>
46        </oslc_auto:ParameterInstance>
47      </oslc_auto:inputParameter>
48
49      </oslc_auto:inputParameter>
50        <oslc_auto:ParameterInstance>
51          <oslc:name>outputFileRegex</oslc:name>
52          <rdf:value>.^</rdf:value>
53        </oslc_auto:ParameterInstance>
54      </oslc_auto:inputParameter>
55
56      <oslc_auto:inputParameter>
57        <oslc_auto:ParameterInstance>
58          <oslc:name>outputFilter</oslc:name>
59          <rdf:value>AnacondaRaceDetection</rdf:value>
60        </oslc_auto:ParameterInstance>
61      </oslc_auto:inputParameter>
62
63      <oslc_auto:outputParameter>
64        <oslc_auto:ParameterInstance>
65          <oslc:name>launchSUT</oslc:name>
66          <rdf:value>True</rdf:value>
67        </oslc_auto:ParameterInstance>
68      </oslc_auto:outputParameter>
69
70      <oslc_auto:outputParameter>
71        <oslc_auto:ParameterInstance>
72          <oslc:name>zipOutputs</oslc:name>
73          <rdf:value>false</rdf:value>
74        </oslc_auto:ParameterInstance>
75      </oslc_auto:outputParameter>
76
77      <oslc_auto:outputParameter>
78        <oslc_auto:ParameterInstance>
79          <oslc:name>timeout</oslc:name>
80          <rdf:value>0</rdf:value>
81        </oslc_auto:ParameterInstance>
82      </oslc_auto:outputParameter>
83
84      <oslc_auto:outputParameter>
85        <oslc_auto:ParameterInstance>
86          <rdf:value>true</rdf:value>
87          <oslc:name>toolCommand</oslc:name>
88        </oslc_auto:ParameterInstance>
89      </oslc_auto:outputParameter>
90
91      <oslc_auto:contribution>
92        <oslc_auto:Contribution rdf:about="http://localhost:8080/analysis/services/resources/
              contributions/1-stdout">
93          <dcterms:title rdf:parseType="Literal">stdout</dcterms:title>
```

```
94            <rdf:value>ANaConDA 0.4 20200202 (git fc5f069-dirty)
95      using libdie 0.3 20200202 (git 95d8ccd)
96
97    Bank week started.
98    i: 0 Account(id=0,balance=0,synchronized=1)
99    i: 1 Account(id=1,balance=0,synchronized=1)
100   i: 2 Account(id=2,balance=0,synchronized=0)
101   i: 3 Account(id=3,balance=0,synchronized=0)
102   Data race on memory address 0x55d216b26160 detected.
103     Thread 4 written to &lt;unknown&gt;
104     accessed at line 85 in file /path/to/universal-analysis-adapter/compilation/SUT/1/src/bank.
            cpp
105     Thread 3 written to &lt;unknown&gt;
106     accessed at line 85 in file /path/to/universal-analysis-adapter/compilation/SUT/1/src/bank.
            cpp
107
108     Thread 4 backtrace:
109
110     Thread created at &lt;unknown&gt;
111
112     Thread 3 backtrace:
113
114     Thread created at &lt;unknown&gt;
115
116   End of the week.
117   Bank records = 937, accounts balance = 1161.
118   ERROR: records don't match !!!
119   </rdf:value>
120           <dcterms:modified rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">
                2021-04-26T19:53:53.986Z</dcterms:modified>
121           <fit:filePath>/path/to/universal-analysis-adapter/compilation/SUT/1/.adapter/
                stdout_analysis_1</fit:filePath>
122           <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
123           <dcterms:created rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">
                2021-04-26T19:53:53.986Z</dcterms:created>
124           <dcterms:description rdf:parseType="Literal">Standard output of the analysis.</
                dcterms:description>
125         </oslc_auto:Contribution>
126       </oslc_auto:contribution>
127
128       <oslc_auto:contribution>
129         <oslc_auto:Contribution rdf:about="http://localhost:8080/analysis/services/resources/
              contributions/1-returnCode">
130           <dcterms:title rdf:parseType="Literal">returnCode</dcterms:title>
131           <rdf:value>0</rdf:value>
132           <dcterms:modified rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">
                2021-04-26T19:53:53.986Z</dcterms:modified>
133           <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#integer"/>
134           <dcterms:created rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">
                2021-04-26T19:53:53.986Z</dcterms:created>
135           <dcterms:description rdf:parseType="Literal">Return code of the execution. If non-zero
                , then the verdict will be #failed.</dcterms:description>
136         </oslc_auto:Contribution>
137       </oslc_auto:contribution>
```

```
138
139       <oslc_auto:contribution>
140         <oslc_auto:Contribution rdf:about="http://localhost:8080/analysis/services/resources/
                contributions/1-stderr">
141           <dcterms:title rdf:parseType="Literal">stderr</dcterms:title>
142           <rdf:value></rdf:value>
143           <dcterms:modified rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">
                  2021-04-26T19:53:53.986Z</dcterms:modified>
144           <fit:filePath>/path/to/universal-analysis-adapter/compilation/SUT/1/.adapter/
                  stderr_analysis_1</fit:filePath>
145           <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
146           <dcterms:created rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">
                  2021-04-26T19:53:53.986Z</dcterms:created>
147           <dcterms:description rdf:parseType="Literal">Error output of the analysis.</dcterms:
                  description>
148         </oslc_auto:Contribution>
149       </oslc_auto:contribution>
150
151       <oslc_auto:contribution>
152       <oslc_auto:Contribution rdf:about="http://localhost:8080/analysis/services/resources/
              contributions/1-executionTime">
153           <dcterms:title rdf:parseType="Literal">executionTime</dcterms:title>
154           <rdf:value>1032</rdf:value>
155           <dcterms:modified rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">
                  2021-04-26T19:53:53.986Z</dcterms:modified>
156           <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#integer"/>
157           <dcterms:created rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">
                  2021-04-26T19:53:53.986Z</dcterms:created>
158           <dcterms:description rdf:parseType="Literal">Total execution time of the analysis in
                  milliseconds.</dcterms:description>
159         </oslc_auto:Contribution>
160       </oslc_auto:contribution>
161
162       <oslc_auto:contribution>
163         <oslc_auto:Contribution rdf:about="http://localhost:8080/analysis/services/resources/
                contributions/1-statusMessage">
164           <dcterms:title rdf:parseType="Literal">statusMessage</dcterms:title>
165           <rdf:value>Executing analysis: /path/to/anaconda/tools/run.sh atomrace ./bank "Hello
                  World!"
166       as: /bin/bash ./.adapter/exec_analysis_1.sh "/path/to/anaconda/tools/run.sh atomrace ./
              bank "Hello World!""
167       In dir: /path/to/universal-analysis-adapter/compilation/SUT/1
168   Analysis completed successfully
169   File Contributions added
170   </rdf:value>
171           <dcterms:modified rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">
                  2021-04-26T19:53:53.986Z</dcterms:modified>
172           <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
173           <dcterms:created rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">
                  2021-04-26T19:53:53.986Z</dcterms:created>
174           <dcterms:description rdf:parseType="Literal">Status messages from the adapter about
                  the execution.</dcterms:description>
175         </oslc_auto:Contribution>
176       </oslc_auto:contribution>
```

```
177
178       <oslc_auto:contribution>
179         <oslc_auto:Contribution rdf:about="http://localhost:8080/analysis/services/resources/
                contributions/1-race_detected">
180           <dcterms:title rdf:parseType="Literal">DataRaceDetected</dcterms:title>
181           <rdf:value>true</rdf:value>
182           <dcterms:modified rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">
                  2021-04-26T19:53:53.986Z</dcterms:modified>
183           <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#boolean"/>
184           <dcterms:created rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">
                  2021-04-26T19:53:53.986Z</dcterms:created>
185           <dcterms:description rdf:parseType="Literal">Holds the result of data race analysis.</
                  dcterms:description>
186         </oslc_auto:Contribution>
187       </oslc_auto:contribution>
188
189     </oslc_auto:AutomationResult>
190   </rdf:RDF>
```

Appendix C

Container For GitLab CI

The adapter repository includes a `.gitlab-ci.yml` file which is used by GitLab CI to automatically execute the adapter's test suite. The file can also be used as instructions for creating your own container to test the adapter. The required steps are starting with a `maven:3.6.3-jdk-8` docker image, installing `Newman` and its dependencies, and then cloning the adapter's repository.

This is what the `.gitlab-ci.yml` file looks like:

```
1  # how to in local docker:
2  # run the image:
3  # $ docker run -it --entrypoint /bin/bash maven:3.6.3-jdk-8
4  #
5  # install newman:
6  # $ apt-get update && apt-get install -y nodejs npm && npm install -g newman
7  #
8  # clone repository
9  # $ git clone https://pajda.fit.vutbr.cz/verifit/oslc-generic-analysis.git
10
11 image: maven:3.6.3-jdk-8
12
13 # Cache downloaded dependencies and plugins between builds.
14 cache:
15   paths:
16     - .m2/repository
17
18 stages:
19   - build
20   - test
21
22 build:
23   stage: build
24   script: /bin/bash ./build.sh
25
26 test-start:
27   stage: test
28   script: /bin/bash ./dev_tools/ci_test_start.sh
29
30 test-newman-core:
31   stage: test
32   before_script:
```

```
33      - apt-get update
34      - apt-get install -y nodejs npm
35      - npm install -g newman
36    script: /bin/bash ./dev_tools/ci_test_suite.sh
37
38  test-newman-keepLastN:
39    stage: test
40    before_script:
41      - apt-get update
42      - apt-get install -y nodejs npm
43      - npm install -g newman
44    script: /bin/bash ./dev_tools/ci_test_keep_last_n.sh
```

Lines 1-9 give instructions on how to start a docker container locally. Line 11 specifies the base image to use. Lines 13-16 enable cache-ing of downloaded Maven artifacts to avoid downloading them for each test. Lines 18-20 specify GitLab CI stages to be executed. Lines 22-24 represent the first test which is building the adapter. Lines 26-28 represent testing if the adapter starts correctly. Lines 30-36 run the main test suite. Lines 38-44 run a secondary test suite for different configuration.