UNITE: An Adapter for Transforming Analysis Tools to Web Services via OSLC

Ondřej Vašíček
ivasicek@fit.vutbr.cz
Brno University of Technology
Honeywell International
Brno, Czech Republic

Jan Fiedor
ifiedor@fit.vutbr.cz
Brno University of Technology
Honeywell International
Brno, Czech Republic

Tomáš Kratochvíla
Tomas.Kratochvila@Honeywell.com
Honeywell International
Brno, Czech Republic

Bohuslav Křena
krena@fit.vutbr.cz
Brno University of Technology
Brno, Czech Republic

Aleš Smrčka
smrcka@fit.vutbr.cz
Brno University of Technology
Brno, Czech Republic

Tomáš Vojnar
vojnar@fit.vutbr.cz
Brno University of Technology
Brno, Czech Republic

ABSTRACT

This paper describes UNITE, a new tool intended as an adapter for transforming non-interactive command-line analysis tools to OSLC-compliant web services. UNITE aims to make such tools easier to adopt and more convenient to use by allowing them to be accessible, both locally and remotely, in a unified way and to be easily integrated into various development environments. Open Services for Lifecycle Collaboration (OSLC) is an open standard for tool integration and was chosen for this task due to its robustness, extensibility, support of data from various domains, and its growing popularity. The work is motivated by allowing existing analysis tools to be more widely used with a strong emphasis on widening their industrial usage. We have implemented UNITE and used it with multiple existing static as well as dynamic analysis and verification tools, and then successfully deployed it internationally in the industry to automate verification tasks for development teams in Honeywell. We discuss Honeywell's experience with using UNITE and with OSLC in general. Moreover, we also provide the Unite Client (UNIC) for Eclipse to allow users to easily run various analysis tools directly from the Eclipse IDE.

CCS CONCEPTS

• **Software and its engineering** → **Development frameworks and environments**; **Software verification and validation**; • **Applied computing** → *Service-oriented architectures* .

KEYWORDS

Transformation to web services, OSLC, OSLC Automation, Software analysis, Tool integration, Eclipse Lyo

ACM Reference Format:
Ondřej Vašíček, Jan Fiedor, Tomáš Kratochvíla, Bohuslav Křena, Aleš Smrčka, and Tomáš Vojnar. 2022. UNITE: An Adapter for Transforming Analysis Tools to Web Services via OSLC. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3540250.3558939

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3558939

1 INTRODUCTION

With the ever-growing complexity of mission-critical software, automated analysis and verification became an integral part of the software development process. Yet many analysis tools, especially new ones, struggle to find their applications in practice despite the analyses they offer being of high interest. According to our experience from the industry, the reasons for this are often of a quite practical nature: their setup and configuration can be difficult even for experienced developers, they can have expensive licenses, they may require a lot of computational resources not available locally, the analyses they perform often run for a long time, their integration into existing development ecosystems can be complicated, etc.

To help resolve such problems and to contribute to a wider, especially industrial, usage of analysis and verification tools, we introduce UNITE—a *universal analysis adapter based on the OSLC standard*—that can be used to easily transform command-line analysis tools into OSLC-compliant web services[1]. By doing so, one can offload these tools onto servers where they can be installed, configured, and maintained by specialists (possibly authors of the tools who can make them available over the internet—but without requiring them to be experts on web services), the tools can use the resources available on the servers, can run their analyses 24/7, and can be easily integrated with other tools and environments via the standardized OSLC interface. The only task left for the users is then to provide the inputs, call a service, and wait for the results.

We tested UNITE with publicly available static and dynamic analysis tools such as VALGRIND [21], FACEBOOK INFER [5], ANACONDA [14, 15], PERUN [16], and THETA [31], as well as with in-house industrial solutions such as HILITE [4]. All of the tools were used through UNITE successfully resulting in a subsequent pilot deployment of UNITE in Honeywell for a number of tools. Furthermore, UNITE is currently being tested by several partners within the Arrowhead Tools EU ECSEL project. Based on practical experience with UNITE in Honeywell (discussed later in this paper), we believe

[1] UNITE is open-source and publicly available at [32].

that our approach can save both time and costs while widening the use of analysis tools which can lead to more errors being detected earlier in the development process.

For users who are not familiar with OSLC or do not have an OSLC-compliant client at their disposal, we further provide the Unite Client (UniC) for Eclipse that allows tools hosted through Unite as web services to be easily used from the Eclipse IDE. To ease the adoption of Unite and to demonstrate its capabilities, we also provide a virtual machine [35] with several analysis tools and step-by-step usage instructions (including videos).

Plan of the Paper. Below, we first briefly introduce the OSLC standard and the fundamental idea of Unite. Then, we describe the architecture of Unite and illustrate how it can be used to transform an analysis tool into an OSLC-compliant web service. Further, we demonstrate how to execute analysis through Unite using its REST API directly, as well as a more user-friendly approach using the Unite Client (UniC) for Eclipse. Then, we discuss existing applications of Unite with various analysis tools, describe Unite's deployment in Honeywell, and share Honeywell's experiences with using Unite. Finally, we discuss related work and future work.

2 OPEN SERVICES FOR LIFECYCLE COLLABORATION (OSLC)

OSLC [22] is an open standard for integrating tools across the entire software development lifecycle. Tools are integrated through self-describing RESTful APIs, use standard HTTP for communication, and use serialized resource representation like RDF, XML, or JSON. The OSLC standard consists of a number of specifications for different domains, such as Requirements Management, Change Management, or Automation (suitable for automated usage of development tools) [25]. These specifications define *resource shapes* for the *resources* that make up the interfaces of tools in each domain. Resource shapes can be imagined as *classes* (i.e., type definitions) and resources as their corresponding *objects* (i.e., instances). For example, *test cases* are a resource in the domain covered by the Quality Management specification. All specifications are built on top of the OSLC Core specification [23], which defines common resource shapes and communication principles (HTTP, REST, and other technologies). OSLC participants are servers and clients producing and/or consuming resources from their domain.

As visible already from the above, a strong point of OSLC is that it is built upon standards—both for communication (REST, HTTP) as well as data representation (RDF, XML, JSON), with a strong emphasis on linked data. It is robust and extensible, meaning new resource shapes can be added, yet adding them does not prevent OSLC participants not knowing these resource shapes from exchanging and sharing the data. It supports data from various domains, providing a unified representation of data from the whole development process (requirements, models, artifacts, tests, etc.), which, in turn, enables interoperability. Its emphasis on linked data allows the data to be linked with meta-models, ontologies, and other formal specifications, allowing precise description of their semantics. All of the above made OSLC very popular in both industry and academia, and makes OSLC-compliant tools very sought after as of late.

3 WHAT IS UNITE?

Since the fundamental idea of Unite might be hard to understand for those not familiar with REST or OSLC, we now provide a non-technical explanation of what Unite *is*, which is intended to help form a clearer picture.

Imagine you have just learned about a new analysis tool. The tool has a command-line interface, and you want to use it to analyse a program. In order to do that, you will likely need to go through the following steps:

(1) Download and install the analysis tool on your computer.
(2) Transfer the program to be analysed to your computer and compile it, if needed.
(3) Execute the analysis tool on it using the command-line (by executing a command with the tool's parameters).
(4) Wait for the analysis to finish, and then look at the standard outputs or files produced by the analysis.

To be able to perform the above steps, you needed to install and run the analysis tool on your personal computer, which can be difficult or even impossible due to the constraints of your hardware or software (e.g., operating system, libraries, other needed tools, insufficient memory, licensing, etc.). If you then wanted your colleagues to start using the new tool as well, then they would also need to go through the whole setup process themselves. Of course, one could also think of installing the tool on a server. Then, however, there is a need for each user to access the tool remotely. Doing so manually, typically using *ssh*, is not a very user-friendly way and can lead to problems with transferring the program to be analysed to the server, running and monitoring a remote analysis task, and transferring the results back. The installation could also be made available from some file server to the local computer, but then one can again hit incompatibilities with running it on the local computer or a lack of local resources.

Now, let us see how the analysis tool can be made available using Unite by transforming it to an OSLC-compliant web service. Unite runs as a server, executes analysis tools based on requests from local or remote clients, and then makes outputs of those executions available to clients as analysis results. The steps can be divided between two actors—a *provider* and a *consumer*. The providers host their analysis tools as web services on their servers, while the consumers use the web services to run analyses on their programs. A provider can be the author of the analysis tool or an administrator of a server within an organization interested in running the tool. A consumer can then be anyone who has access to the service. Each of the actors needs to perform the following steps:

(1) The providers directly manage their servers:
  (a) They download and install the analysis tool on a server,
  (b) install Unite and configure it for the analysis tool, and
  (c) start Unite to make the tool available as a web service.
(2) The consumers interact with Unite as a web service:
  (a) They send a request to transfer the program to be analysed to the server and to optionally compile it,
  (b) send a request to run an analysis on the program using a chosen analysis tool,
  (c) wait for the analysis to finish by polling its state, and then get the standard or file outputs from the server.

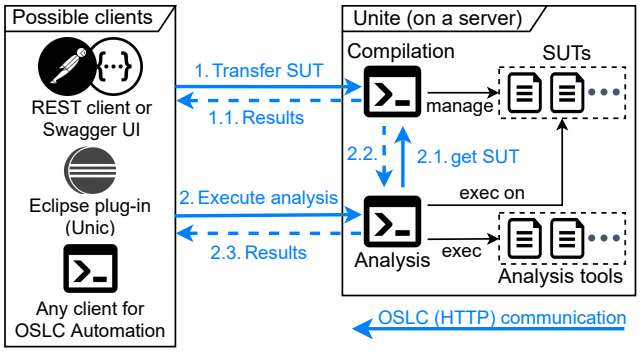

Figure 1: An overview of Unite's architecture. The left side shows different ways to interact with Unite. The right side shows Unite running on a server made up of two components. The compilation sub-adapter manages SUTs and the analysis sub-adapter executes analysis tools on them.

This way, the analysis tool may run on a remote server managed by someone else than you (assuming you are the consumer in this example), possibly an expert on the tool, maybe its author. This means you do not need to go through the setup process or be limited by the hardware or software of your personal computer. You only need to know the address of the web service and be able to interact with it. The same applies to all your colleagues who might also be interested in the new analysis tool. On the other hand, the author of the analysis tool or the administrator installing the tool on a server does not have to be an expert in developing web services.

Note that interacting with the service may be done through its OSLC-compliant RESTful API, which is mainly meant for other developers who want to integrate the service into their own ecosystems (for example, to execute analyses from their test management tool or from a version control system like Git whenever a new version is committed). For regular users, we provide UniC, a plugin for the Eclipse IDE, which abstracts users from the REST API and OSLC, and all they need to do is to click a context menu to execute an analysis in a much more user-friendly way.

The standardized interaction is what sets Unite apart from other solutions. While it is possible to write a script which uses, e.g., ssh, to interact with a tool on a remote server, it will be specific to this tool, and every tool would need its own script for this purpose. With Unite, any OSLC or REST client knows how to interact with any tool. Unite defines what the requests and results look like, how to choose the tool to execute, how to specify the parameters (if the tool has any), and everything remains the same regardless of the tool we need to interact with.

4 ARCHITECTURE OVERVIEW

Unite is a standalone tool-chain of two servers referred to as *sub-adapters* as shown in Figure 1 and further explained throughout this section. The solution was divided into two sub-adapters to separate two different steps needed for analysing a system under test (SUT) on a remote server and to allow each of the sub-adapters to be reusable in other use-cases and to be replaceable, e.g., by parts of an existing tool-chain. Both sub-adapters are also connected with a database to enable resource queries and persistence.

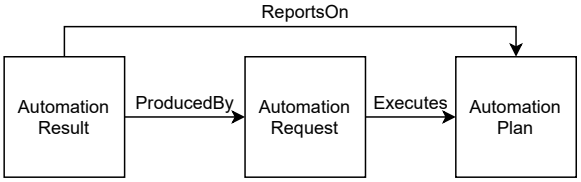

Figure 2: The OSLC Automation domain contains three main resources. Automation Plans represent units of automation offered by a server and define their input parameters. Automation Requests are what clients create to request execution of one of the available Automation Plans. Automation Results are then produced by the server and contain outputs of the execution.

The first simpler sub-adapter is the *compilation sub-adapter*. Clients interact with it to transfer SUT files to the server and optionally compile them. After transferring, SUT files are stored in a local folder on the server and compiled (if requested) by executing a compilation command. SUT folders then work as workspaces for the execution of analysis tools (i.e., analyses are executed in the SUT folders to preserve files produced for consecutive analyses).

The second more complex sub-adapter is the *analysis sub-adapter*. Clients interact with it to analyse a previously created SUT using a chosen analysis tool. We consider the adapter universal as it can be configured for any non-interactive command-line analysis tool (see Section 5 for more details) by essentially representing its command-line interface as input parameters contained in an OSLC request. The analysis sub-adapter communicates with the compilation sub-adapter to get information about SUT resources to be analysed, such as the folders they are located in or their execution commands. The analysis is performed by asynchronously executing the analysis tool in the obtained SUT folder using the server's local shell (PowerShell or CMD on Windows, Bash on Linux). Concurrent analysis requests will result in concurrent analysis tool executions. A FIFO queuing system can be enabled for analysis tools that should not run in multiple instances concurrently.

Both sub-adapters use a RESTful API as their interface modelled w.r.t. the OSLC Automation specification [24]. These interfaces were generated from graphical models of Unite's architecture in the Eclipse Lyo Designer [9] and use three main resources shown in Figure 2. Unite then implements these interfaces to provide its functionality. Users interact with Unite using the interfaces of one of the sub-adapters (depending on the current interaction). In order to do that, they can use any OSLC Automation client, a simple REST client, or Unite's integrated Swagger UI [29]. We also provide the Unite Client (UniC) for Eclipse (see Section 7), a plug-in for the Eclipse IDE that allows analysis to be executed directly from the IDE using context menus, as a more user-friendly way to use Unite.

Note that Unite's applicability is broader, despite that it focuses mainly on analysis tools in terms of its features (such as facilitating the typical workflow of getting an SUT, compiling it, and then executing analyses on it). Indeed, Unite can be configured to execute any tool, not just analysis tools, that has a command-line interface and produces its outputs to the standard output or by creating files. The sole limitations we are aware of are interactive tools that require non-deterministic user inputs during their execution, such

as interactive theorem provers. As UNITE is implemented in Java, it should be usable on any OS, although, we have so-far explicitly tested it on Windows and Linux only.

It is important to note that, due to its nature, UNITE should be deployed in docker, in a virtual machine, or in a trusted environment as it allows clients to execute any SUT directly on the server. An SUT could potentially be malicious and tamper with the server.

5 CONFIGURING UNITE TO GET A WEB SERVICE FROM AN ANALYSIS TOOL

This section demonstrates how to configure UNITE for a new analysis tool. The configuration is a step that needs to be performed by the analysis tool *provider* in order to make it available as a web service on a server using UNITE. We use ANACONDA [14, 15], a dynamic analysis framework, as an example analysis tool.

Configuring UNITE for a new analysis tool is performed by creating two files in its configuration directory: (1) an *automation plan* defined using an RDF file, called *anaconda.rdf* in our example, and (2) a definition of *technical parameters* given as a Java properties file, called *anaconda.properties* in our example. Configuration files can be modified or added anytime during UNITE's deployment. UNITE only needs to be restarted in order for the new configuration to take effect. Further optional configuration includes defining output filters which are used for processing outputs of the analysis tool. All these configuration possibilities are described later in this section.

Automation Plans. The first above mentioned configuration file—the RDF file—needs to contain a definition of an *automation plan* resource. Automation plans represent units of automation offered by OSLC Automation servers. Their main properties are *parameter definitions* that define input parameters which can be submitted by clients when requesting execution of an available automation plan. UNITE uses automation plans to represent individual available analysis tools[2], and their parameter definitions typically correspond to command-line parameters of a given analysis tool. Clients can then browse available automation plans to discover which analysis tools are available and what their input parameters look like.

Each automation plan needs to contain a unique identifier which is used as a part of its URI. Then, a number of parameters can be defined based on the command-line parameters of the analysis tool. The defined parameters can mimic the tool's command-line parameters to give clients full control of the analysis tool, or they can be simplified to abstract clients from the details of the tool's interface. Each parameter definition needs to have a name and a command-line position which instructs UNITE how to order their values when constructing the string to execute the analysis tool (more in Section 6). Other possible properties include default values, a list of allowed values, occurrence restrictions, or a value prefix for named command-line parameters. When a client requests execution of an automation plan, UNITE automatically checks that all the submitted input parameters are correct and that no required ones are missing. The simplest automation plan can contain just a single parameter which is used for clients to supply all the tool's parameters in a single string that is then placed on the command-line and parsed as it would be when using the command-line directly.

[2]There may be multiple automation plans for one tool, e.g., with different default parameters to represent individual pre-configured analyses.

```
1 <AutomationPlan>
2 <identifier>anaconda</identifier>
3 <parameterDefinition>
4   <name>analyser</name>
5   <cmdlinePos>1</cmdlinePos>
6   <occurs res="Exactly-one"/>
7   <allowedValue>atomrace</allowedValue>
8   <allowedValue>fasttrack</allowedValue>
9 </parameterDefinition><parameterDefinition>
10  <name>SutLaunchCommand</name>
11  <cmdlinePos>2</cmdlinePos>
12  <occurs res="Exactly-one"/>
13  <defaultValue>True</defaultValue>
14 </parameterDefinition><parameterDefinition>
15  <name>SutInputs</name>
16  <cmdlinePos>3</cmdlinePos>
17  <occurs res="Zero-or-One"/>
18 </parameterDefinition></AutomationPlan>
```

(a) A simplified automation plan defined for ANACONDA in *anaconda.rdf*. Namespaces, URLs, and some properties were omitted for space and readability reasons. The automation plan defines three parameters. The first parameter (lines 4–8) is compulsory and is used to pick one of the available analysers. The second parameter (lines 10–13) is a special compulsory parameter which instructs UNITE to place the SUT launch command at the specified command-line position. The third parameter (lines 15–17) is an optional parameter used to specify inputs for the executed SUT.

```
1 toolLaunchCommand=/anaconda/run.sh
2 #toolSpecificArgs=--no-color
3 oneInstanceOnly=False
```

(b) Technical properties defined in *anaconda.properties*. There are three properties. The first one tells UNITE how to execute ANACONDA. The second one holds parameters to be always placed on the command-line, such as removing colored outputs, which is currently not applicable for ANACONDA (the line is commented out). The third one tells UNITE whether multiple instances of the tool can run at the same time or whether to enable queuing.

Figure 3: An example configuration for ANACONDA

Figure 3a shows a simplified automation plan for ANACONDA and its command-line interface, which looks like this:

```
run.sh <analyser> <sut-launch-command> <sut-inputs>
```

There are some parameter definitions with reserved names and special functionality recognized by UNITE, such as *SutLaunchCommand* that instructs the adapter to look up the SUT launch command as a property of an SUT resource and insert it to the specified command-line position. Also, a number of common parameter definitions are added to all automation plans by UNITE automatically. These include parameters like an execution timeout or output processing options. The most important common parameter added to all automation plans is the *SUT* parameter that is compulsory and is used to reference the SUT resource to be analysed.

Technical Properties. The second, above mentioned configuration file configures *technical properties* of the analysis tool that need not or should not be visible to clients. Most importantly, it specifies

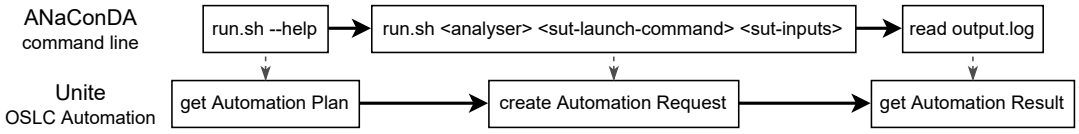

Figure 4: From a command-line execution of ANaConDA to executing it via Unite.

the path to the executable of the analysis tool. An example of such a file is shown in Figure 3b along with a description of its contents.

Output Filters. Further optional configuration allows custom *output filters* to be defined using a plug-in system. These allow outputs of each analysis tool to be processed in a tailored way. A filter is a Java class implementing a method which receives a collection of *contributions* (parts of OSLC automation results) which represent individual outputs of the analysis tool, such as standard outputs and/or produced files. The filter can then process the contributions in any way (e.g., search for error reports in the standard output) and output a collection of untouched, modified, or entirely new contributions. Output filters are an optional feature and Unite includes built-in filters for use by clients who do not need custom ones. For more details on output filters refer to Unite's Wiki [33].

6 RUNNING ANALYSIS USING UNITE'S API

Once Unite is configured for an analysis tool by a *provider* on a server, it can be used by *consumers* using their clients to execute analyses on their SUTs. As shown in Figure 1, there are multiple clients that can be used to communicate with Unite due to its interface being a REST API. In this section, we explain how to execute analysis with Unite by interacting with its REST API directly, i.e., by sending and receiving HTTP requests (POST, GET, etc.) and responses with RDF data content. This is most useful for understanding how Unite works, especially for developers who want to integrate analysis execution using Unite into their products. The typical workflow when a client wants to execute an analysis using Unite consists of two steps with three sub-steps each:

(1) *Transfer the source files* of your SUT to the Unite server by interacting with the *compilation sub-adapter*:
   (a) *Retrieve the Automation Plan* for SUT creation by sending a GET request to its URI. Inspect the Automation Plan to learn about input parameters defined in it.
   (b) *Create an Automation Request* with input parameters (e.g., the source of SUT files) based on the retrieved Automation Plan by sending a POST request to the appropriate URI. This will start the SUT creation process.
   (c) Poll the state of your Automation Request until it finishes. Then, *retrieve the produced Automation Result* by sending a GET request to its URI. The result will contain the URI of your newly created SUT resource.
(2) *Execute analysis* on the previously created SUT by interacting with the *analysis sub-adapter*:
   (a) *Retrieve the Automation Plan*, which corresponds to your desired analysis tool, by sending a GET request to its URI. Inspect the Automation Plan to learn about input parameters defined in it.
   (b) *Create an Automation Request* with input parameters (e.g., tool parameters) based on the retrieved Automation Plan by sending a POST request to the appropriate URI. This will start the analysis execution.
   (c) Poll the state of your Automation Request until it finishes. Then, *retrieve the produced Automation Result* by sending a GET request to its URI. The result will contain outputs of your analysis.

Step 1 is performed by interacting with the compilation sub-adapter, and Step 2 by interacting with the analysis sub-adapter. However, both sub-adapters use an OSLC Automation interface which makes both steps almost the same (up to different automation plans, input parameters, and contents of results). As discussed in Section 5, an OSLC Automation interface contains a number of *automation plans* (i.e., available units of automation). A client needs to pick the one they want to execute and GET it to see its parameter definitions. Then, they can create an *automation request* (by sending a POST request to the appropriate creation factory), referencing the selected automation plan while specifying input parameter values (each input parameter corresponds to a parameter definition). This will cause the destination sub-adapter to execute the requested automation plan asynchronously and to produce an *automation result*. Either the automation result or the automation request then has to be polled[3] (i.e., sending multiple GET requests) by the client to monitor their *state* property. An in-progress request can be canceled by updating or deleting it. Once complete, the automation result will contain outputs of the executed automation plan including a *verdict* property which signifies whether the execution finished successfully or whether there were any errors (e.g., a non-zero exit code, out of memory, crash, etc.). All requests and results are stored in a SPARQL triplestore to allow queries using the OSLC Query syntax [27] and optional persistence[4].

For a better understanding of how OSLC Automation works, see Figure 4 which shows how executing analysis using ANaConDA on the command-line translates to doing it through Unite.

We now provide more details on creating automation requests to the two sub-adapters of Unite. Note that typical requests can be pre-prepared and reused (either as is or just with some small changes for a given specific analysis run).

Requests to the Compilation Sub-Adapter. The compilation sub-adapter has a single automation plan for creating SUT resources only. As some SUT resources may be executable, the parameters of this automation plan include not only the source files of the SUT, but also the compilation and launch commands needed to compile and execute it. Supported file sources include a Git repository, a general URL, a base64-encoded string, or a path to a file already located

[3] Multiple automation requests can run and can be polled concurrently.
[4] SPARQL is well suited for RDF resources and is supported by OSLC tooling [19].

on the server (e.g., transferred by the client manually by other means). After an automation request is created by a client, the compilation sub-adapter will create a local folder as a workspace for the SUT, automatically fetch its source files from the specified source, and optionally execute a compilation command on the SUT. The automation result will then contain a newly created SUT resource identified by a URI along with outputs of the compilation and transfer processes. See Figure 5a for a simplified example of an automation request used for creating an SUT.

Requests to the Analysis Sub-Adapter. The analysis sub-adapter can have multiple automation plans corresponding to individual available analysis tools and/or to various pre-configured analyses for a single tool. When creating an automation request, the client needs to pick the right automation plan according to their desired analysis tool. The parameters of the selected automation plan will include a URI of the previously created SUT resource to be analysed, input parameters for the analysis tool w.r.t. its configuration (from Section 5), and parameters for controlling additional features of Unite. The analysis sub-adapter will then take the values of the submitted input parameters, corresponding to command-line parameters of the analysis tool, and build a single string to execute using the provided values of the parameters, their defined command-line positions, value prefixes, and default values. The string will be executed asynchronously using the local shell. The automation result can contain all outputs of the analysis as *contributions*, which can include standard outputs, any files produced by the analysis, or contributions created by an output filter. See Figures 5b and 5c for simplified examples of an analysis execution request and of the result produced for that request. If the parameter values from Figure 5b are positioned according to the configuration from Figure 3a, one gets the following command to be executed:

```
/anaconda/run.sh atomrace ./hello HelloWorld
```

7 USING THE ECLIPSE IDE TO RUN ANALYSIS

In the previous section, we described how any REST client can be used to interact with Unite. Naturally, using a REST client requires the user to write and send an OSLC-compliant request to Unite in order to successfully communicate with it. Having an OSLC client alleviates a lot of the work and many companies already have their own internal OSLC clients they could use to integrate Unite into their OSLC ecosystem. Yet many users do not have such a client at their disposal and remain stuck with only a REST client to use.

To address this issue and simplify the usage of Unite, we provide the Unite Client (UniC) for Eclipse. Users can install this client from an Eclipse update site as any other Eclipse plugin. UniC can be configured to remotely execute any tool transformed by Unite. This is thanks to the UniC's extensible architecture in which users can define *jobs* made up of built-in *tasks* or their own custom *tasks*. After the configuration, users can invoke the tool directly from the menu of the Eclipse IDE, feed it with input data, and fetch the results back into Eclipse, potentially transforming them into Eclipse markers for a more user-friendly representation.

As an example, we provide a UniC configuration for the Facebook Infer static analyser [5] (available from the same update site as UniC). The user needs to add the VeriFIT update site [13] and install the *Infer on BUT server (eTE Task)* feature from it. UniC and all

```
1 <AutomationRequest><!--POST TO compilation-->
2 <execAutomationPlan res="../URI/SutCreation"/>
3 <inputParameter>
4   <name>srcUrl</name><val>http://sut.zip</val>
5 </inputParameter><inputParameter>
6   <name>unpackZip</name><val>true</val>
7 </inputParameter><inputParameter>
8   <name>buildCommand</name><val>make</val>
9 </inputParameter><inputParameter>
10   <name>launchCommand</name><val>./hello</val>
11 </inputParameter></AutomationRequest>
```

(a) A simplified automation request used for creating an SUT. The request contains a link to the automation plan to execute (line 2) and four input parameters. The first one (line 4) specifies where to fetch the SUT source files from, in this case to download them from a URL. The second one (line 6) specifies that the downloaded SUT files are in a ZIP archive which needs to be unpacked before compilation. The last two (lines 8, 10) specify the SUTs build and launch commands.

```
1 <AutomationRequest><!-- POST TO analysis -->
2 <execAutomationPlan res="../URI/anaconda"/>
3 <inputParameter>
4   <name>analyser</name><val>atomrace</val>
5 </inputParameter><inputParameter>
6   <name>SUT</name><val>../URI/of/the/SUT</val>
7 </inputParameter><inputParameter>
8   <name>SutInputs</name><val>HelloWorld</val>
9 </inputParameter></AutomationRequest>
```

(b) A simplified automation request used for executing analysis on an SUT using ANaConDA. It contains a link to the automation plan to execute (line 2), which is the one for ANaConDA from Figure 3a, and three input parameters. The parameters specify the type of analysis to perform (line 4), the SUT to analyse (line 6), whose value must be a link to the SUT resource (created using the request from Figure 5a), and the input data given to the SUT (line 8).

```
1 <AutomationResult>
2 <producedByAutoRequest res="../URI/requestN"/>
3 <reportsOnAutoPlan res="../URI/anaconda"/>
4 <state res="complete"> <verdict res="passed">
5 <contribution>
6 <title>stdout<title><val>HelloWorld...</val>
7 </contribution><contribution>
8 <title>stderr<title><val>No dataraces...</val>
9 </contribution><contribution>
10 <title>statusMsg</title><val>Executing:
11  /anaconda/run.sh atomrace ./hello HelloWorld
12 In dir: /path/to/SUT/directory
13 Completed successfully<val></AutomationResult>
```

(c) A simplified automation result produced for the request from Figure 5b. It contains links to the associated automation plan and request (lines 2–3) and properties which signify that the analysis finished without issues (line 4). Then, there are three contributions containing the standard outputs produced by ANaConDA (lines 6, 8), and Unite's reports on the execution process (lines 10–13).

Figure 5: Examples of requests sent to Unite and of the result produced for one of them. Namespaces, URLs, and some properties were omitted for space and readability reasons.

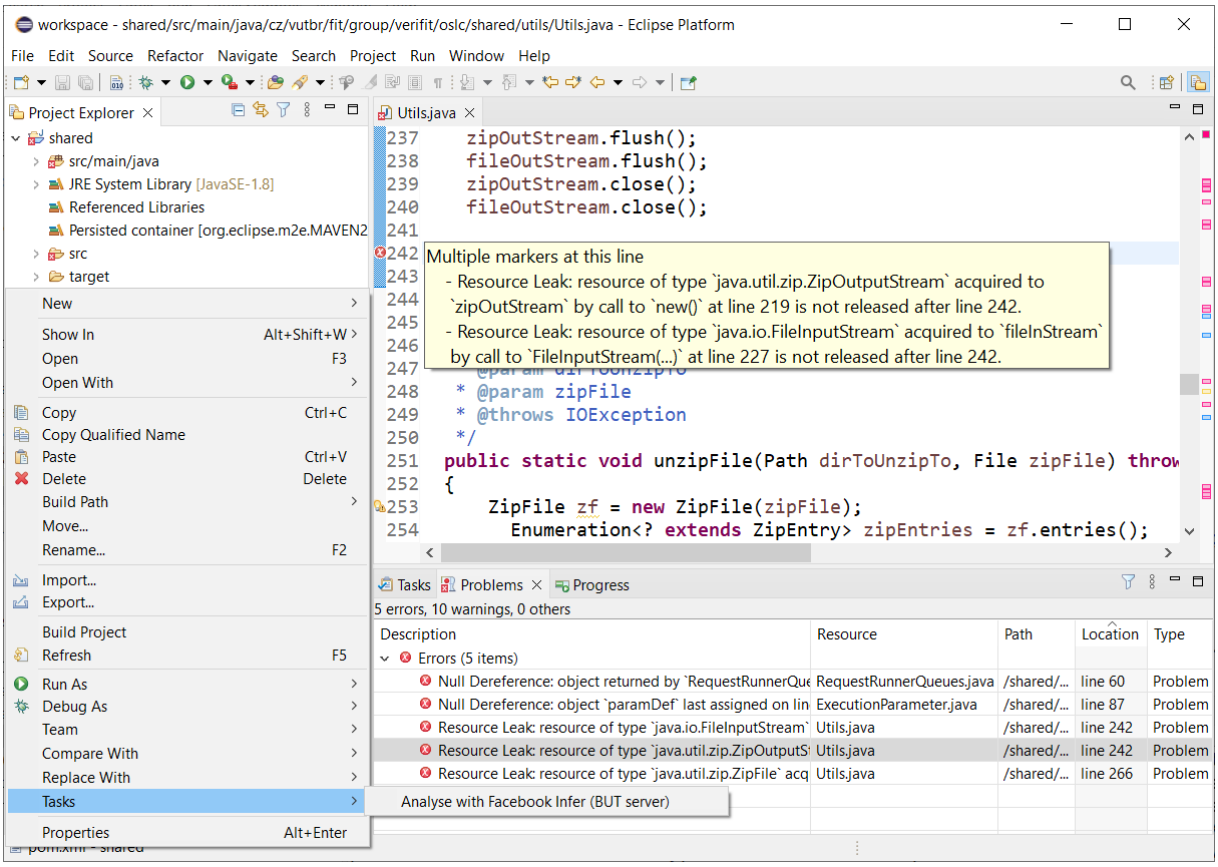

Figure 6: Execution of FACEBOOK INFER and visualization of its results in the Eclipse IDE.

of its dependencies will be installed automatically with this feature. After the installation, the *Analyse with Facebook Infer (BUT server)* entry will show up in a popup menu under the *Tasks* category as shown in the bottom-left part of Figure 6.

As INFER requires a build command in order to analyse a project, the user must first specify it in the project's properties, UNITE section, SUT page. After that, the user can right-click on an Eclipse project, select the *Analyse with Facebook Infer (BUT server)* task under the *Tasks* category, and let UNIC handle everything. When the analysis is finished, UNIC will fetch the results and transform them into Eclipse markers, adding one marker for each error found by INFER as shown in the Problems view in the bottom-right part of Figure 6. These markers are linked with specific lines in concrete source code files where the error was found, and the user can jump to these locations from the Problems view. The code editors also show these errors as editor markers with their description in a hover text as can be seen in the center of Figure 6.

The UNIC configuration for INFER suffices with built-in tasks, which are part of UNIC and its libraries. Any tool developer using UNITE to transform their tool into an OSLC-compliant web service can adapt this configuration to allow UNIC to execute their tool and provide this configuration to the users through an update site.

8 CASE STUDIES

We now discuss several instances of using UNITE with different tools which confirm that UNITE is indeed usable with a broad spectrum of non-interactive command-line analysis tools.

In our experimental evaluation, we considered a number of static as well as dynamic analysis and verification tools. The tested tools included VALGRIND [21], the FACEBOOK INFER static analyser [5], GREP (as a representative of a general Unix utility, also perceivable as the simplest static analyser), the dynamic concurrency analyser ANACONDA [14, 15] (used as an illustrative example in Sections 5 and 6), the dynamic performance analyser PERUN [16], the model checking framework THETA [31], and Honeywell's proprietary test vector generation tool HILITE [4].

- ANACONDA and VALGRIND as dynamic analysers have the same and most typical usage workflow as they first need the SUT to be compiled (as shown in Figure 5a) and then the analysis can be executed (as shown in Figure 5b) which immediately produces the final results of their analyses as standard outputs that can be retrieved by clients (as shown in Figure 5c).
- PERUN differs from the other dynamic analysers due to its tight integration with the Git versioning system (indeed, PERUN aims at detecting performance regressions between program versions). Further, PERUN needs to run multiple subsequent analyses on the same compiled SUT, introducing a need for

running multiple analyses in the same workspace, i.e., with files produced by one analysis persisting for the following analyses.

- Infer as a static analyser does not need the SUT to be built, it suffices with its source code and its compilation command[5]. This can be achieved by simply disabling compilation during SUT creation. Infer's analysis then produces standard outputs but also a directory which contains detailed reports and logs.
- Similarly, Theta only needs the SUT to be preprocessed, not compiled, which can be treated as a special case of compilation with specific arguments for the compiler. Theta is then executed on the preprocessed source files producing standard outputs or files (e.g., containing counter-examples produced by model-checking). All produced files and directories can be included on request in analysis results offered by Unite.
- Grep demonstrates that Unite is usable with common non-interactive UNIX utilities. Compilation would be disabled for Grep just like it was for Infer. One could even use a pipeline of UNIX utilities or a script instead of Grep in order to have a more complex functionality executed by Unite.
- Finally, HiLiTE takes requirements and models as its inputs and generates test vectors. As such, HiLiTE's inputs are not an SUT (in the sense of being a set of source files to compile and execute) but general files. Unite can handle such files in the same way as a normal SUT with disabled compilation. Note that the compilation step can be used to perform some form of preprocessing on the inputs for example. HiLiTE also needs some environmental variables to be set during execution which can be facilitated using one of the common input parameters provided by Unite for all configured analysis tools.

Despite all these differences, Unite allowed us to make all of the tools available as OSLC-compliant web services and to utilize them remotely[6]. Moreover, Unite is already being used by several academic as well as industrial parties other than the authors.

9 INDUSTRIAL EXPERIENCE

Unite is currently deployed with a number of tools in Honeywell on both Windows and Linux platforms. The most important ones are HiLiTE [4] and its versions. Other tools which are being experimented with include Symbiotic [28], DIVINE [3], and NuSMV [7]. Honeywell also has prior experience with OSLC from using a number of other verification tools using their own custom OSLC servers. In this section, we share Honeywell's experience with using Unite as well as with OSLC in general.

Most of the experience comes from using Unite with HiLiTE which is a qualified tool that generates test vectors from Matlab models and from high-level and low-level requirements. Moreover, HiLiTE is also applied in connection with Honeywell's usage of its hard real-time variant of CLIPS, which is a tool for building expert systems originally developed by NASA [37]. CLIPS rules are converted to requirements and then processed by HiLiTE to generate test vectors. Unite was deployed (and still is) with various versions of HiLiTE on 5 Honeywell servers for around 8 months. Approximately 43 engineers from Brno in the Czech Republic (further referred to as the *Brno team*) are using two different clients to run HiLiTE through Unite. Around 18 of them are new users who do not have any prior experience with HiLiTE and would have likely not used it otherwise. In addition, teams from Asia (further referred to as the *Asia teams*) are using a different client to also run HiLiTE through Unite. Their experiences were gathered using various interviews and data collected from servers, and are discussed throughout the rest of this section.

Experience from Brno. The Brno team uses two different clients. The first one is a web UI client made specifically for accessing HiLiTE as an OSLC-compliant service. The second one is ForReq [2], a desktop GUI client which remotely executes a range of tools and aggregates their outputs. ForReq was previously unable to execute HiLiTE due to HiLiTE being limited to Windows and there was no existing solution for running remote verification tools on Windows prior to Unite. The new integration is also more robust compared to the previous integrations of ForReq with other tools which were only sending remote commands to a server and any changes in how the tool should be executed forced changes in the integration code of all the clients. On top of that, the files to analyse needed to be transferred to the server by other means. Unite provides an OSLC interface for both transferring files and performing analyses which unifies the communication. In addition, changes to how the HiLiTE tool is executed do not mandate changes to the OSLC interface, simply updating the tool's automation plan suffices, making it very robust. Using Unite effectively removed all issues with new HiLiTE versions breaking the integration code and solved problems of using multiple HiLiTE versions concurrently by defining configuration and automation plans for each version.

The Brno team has further estimated their average time savings. They saved approximately three 90-minute long Matlab installations per month, and approximately fourteen 15-minute long installations of HiLiTE and its other dependencies. This came at a cost of having to install and configure Unite with HiLiTE once per server. Using the web UI client comes at no cost as it does not need any installation on the user side, and using ForReq essentially comes at no extra cost as well because the team members already use it for other purposes. In practical experience, Unite can be installed in less than 30 minutes and requires only Maven and Java to be available for the installation. After the installation, configuring Unite for an analysis tool in the easiest way (i.e., handling all parameters of the tool as a single string) can be done in a few minutes. This process can be made even faster when using Docker[7], since potential issues with installing Java and Maven can be avoided.

The execution overhead of Unite is very small. Processing an analysis request before executing the analysis tool itself takes a few seconds at most, as does producing an analysis result after the analysis tool finishes its execution. The analysis tool itself runs without any overhead. Overall, the overhead is negligible compared to execution times of typical analysis tools which can take hours to finish their analyses. Transferring large SUTs to the server could delay the analyses, yet each SUT can be transferred only once

[5] Infer may determine which files to analyse from the build process of the SUT, for which it needs to know how to build it even when not using the resulting executable.
[6] Interested readers are welcome to check our claims through a Unite virtual machine [35], which allows one to see almost all the functionality claimed in this paper, using a REST client (Postman [26]) with step-by-step instructions, or by watching a video. The exceptions missing in the VM are HiLiTE (proprietary) and Theta (only tested after VM creation). The video by itself outside of the VM is available at [34].
[7] Dockerfiles and compose files for Unite are available in its GitLab repository [32].

and then reused for multiple analyses. Unite's performance under heavy load (i.e., a large number of concurrent requests) was not a major concern during its design due to the fact that a large number of analysis tools being executed concurrently will typically overload the server regardless of Unite's performance. Unite can handle concurrent analysis requests and polling of request states without issues as long as the executed analyses do not overload the server. Unite does feature a FIFO queuing system for analysis request executions which can be enabled to prevent such server overload.

Having verification tools as OSLC-compliant web services enabled gathering of data throughout the development process. It is now possible to track who uses which tool, how often, how long the execution takes, how many errors were detected, and so on. It is also possible to compare execution times and outputs of different versions of the same verification tool on the same inputs to detect and fix performance degradations or newly introduced bugs. For example, HiLiTE is typically executed thousands of times a month using Unite and on average over 40 % of verified requirements need to be reworked to pass the verification.

Experience from Asia. A number of teams in India are currently using a command-line client to invoke HiLiTE remotely in CI/CD pipelines. Due to all the issues discussed throughout this paper, such as installation and compatibility, HiLiTE's usage prior to Unite was limited to only a few engineers from the India teams who were using it directly on their personal computers. Introducing Unite allowed HiLiTE to be used in CI/CD pipelines by removing the need to run CI/CD agents and HiLiTE on the same server, which was often impossible due to licensing, compatibility, and other issues. Having HiLiTE integrated into CI/CD pipelines allows users who are not even aware of this tool to still utilize its functionality and have their Matlab models analysed. This significantly widened HiLiTE's user base in the teams from India.

However, the disadvantage of this approach is that the analyses are done only when the pipeline is triggered, which occurs when changes to the Matlab models are published by the users. Naturally, many users want to analyse their modified models before they publish them. Moreover, they may also want to try different versions of HiLiTE and/or different configurations of its analyses, which is difficult with pre-configured pipelines. Therefore, the India teams are currently starting to use UniC to allow their users to perform analyses using various versions of HiLiTE and different configurations of its analyses directly from their IDEs.

Furthermore, the simplicity of configuring UniC for multiple versions of HiLiTE with multiple configurations of its analyses, which can be done remotely from Eclipse update sites, also sparked the interest of several teams in China. The China teams cannot access HiLiTE directly because of export control restrictions, therefore, members of the India teams were required to perform the analyses on their behalf. With HiLiTE being accessible as a service and with UniC's ease of use, the China teams can now do on-demand analyses themselves by simply picking their desired version and configuration of HiLiTE from a menu in their IDE and waiting for the results instead of having to rely on the India teams.

An important advantage of using Unite was saving effort, time, and costs. Most of the costs savings come from reduced licensing costs and operator costs. Having only a few central installations of, e.g., Matlab, on servers reduces licensing costs compared to each team member having their own local license. While having one shared Matlab server is already common practice, an operator is often required to process analysis requests on the server. An operator is a team member whose job is to wait for tasks from the rest of the team and to then run these tasks on the shared server to provide outputs of the tools, such as generated test-cases in case of HiLiTE, back to the rest of the team. Having an operator abstracts the rest of the team from manually going through the process of using a tool on a remote server. With Unite, each team member can run their own tasks easily using one of the available clients effectively eliminating the need for a dedicated operator.

As is evident from the described experience, the most important advantage of using Unite is its impact on widening the user base thanks to making analysis tools easier to adopt and use. In practice, 18 out of the total of 43 engineers from the Brno team were new HiLiTE users, the number of users in the India teams increased substantially from just a few previous users[8], and teams from China were unable to use HiLiTE themselves before at all. All these new users have not used HiLiTE before and would have most likely not used it at all if it was not for Unite. This is indeed the main goal of Unite and its main selling point. Having more people actively use verification tools can be very beneficial to the overall development process as they can detect defects which would otherwise (hopefully) be detected only in the testing phase much later.

Indeed, the more people actively use a tool the more errors can potentially be detected by it, making it more useful. The more useful a tool is the more new users are likely to be interested in using it which leads to more users, more defects detected, and more usefulness of the tool. Detecting defects as early as possible in the development process of a product is crucial, because the later an error is discovered the more expensive it is to fix it. According to the study [36] conducted within the AVSI SAVI project around 70 % of errors in large systems are introduced as early as in the specification of system requirements, yet over 50 % of those errors are only discovered during integration testing much later in the development process. The cost of fixing an error in the integration testing phase is around 16-times higher than in the initial requirements specification phase and grows much higher in later stages. In Honeywell, for example, generating test cases with HiLiTE based on CLIPS rules, as was mentioned at the start of this section, is performed exclusively through Unite. This enables early detection of defects in requirements specifications and expert systems reducing the time and the cost of reworking them.

Honeywell's overall experience with Unite and verification tools as OSLC-compliant web services is the following:

- Unite indeed is in practice easy to configure for new analysis tools, makes them easier to adopt and use, and allows multiple versions of them to be available at the same time.
- Practical experience confirmed that converting analysis tools to web services can have a significant positive effect on widening the user base of a tool.

[8]Unfortunately, we cannot provide a precise number here since we cannot easily compute how many users access HiLiTE through the CI/CD pipelines in India.

- Moreover, having analysis tools available on servers as web services saves time on tool installations, and reduces licensing costs and operator costs.
- Web services provided by Unite are highly reliable and a verification tool can now work for months without issues.
- Unite is still being actively developed, therefore, there is still room for improvement in terms of tool maturity such as better troubleshooting options, and in terms of missing features such as user management or better security.
- Finally, there is currently no automated system in place which would allow clients to discover available servers and, therefore, each client needs to be manually configured to know the appropriate server address. Note that this issue will be fixed very soon (see future work in Section 11).

Finally, we discuss Honeywell's experience with using OSLC in general. Using OSLC is beneficial because it is an open-source, distributed, and very rich integration technology. It is also by design flexible and easily extensible with new resources or entire specifications. This means it has potential to become widely adopted as it is well suitable for a wide range of use cases. However, as is common with protocols that use serialized XML data, OSLC communication transfers a lot of information consuming bandwidth and making it not efficient for file management and repositories, or for heavy event-triggered traffic. This, however, is not much of an issue in the case of Unite as there is not such a high volume of analysis requests and their contents are fairly small. An exception is transferring large SUTs or large analysis tool outputs which can cause issues in poor network conditions.

10 RELATED WORK

We are not aware of any existing solutions for universally transforming any analysis tool, or at least a certain class of tools, to an OSLC-compliant web service other than Unite. The closest seem to be two other implementations of the OSLC Automation specification that were used in the AMASS project [8] for remote integration of tools into the CHESS toolkit [6]. The first implementation was developed by FBK ES to provide adapters for their tools [12]. We were unable to find any publications about these adapters, but were able to inspect their source codes and documentation at EATA [11] along with AMASS deliverables. These adapters seem to serve the same role as Unite and UniC but only for FBK tools and not for universal use of a wide range of tools by others. According to their documentation, the adapters can be extended with new tools, however, doing so requires modifications of the source code both on the server side and client side. In comparison, adding new analysis tools to Unite and UniC can be done through configuration files with no changes to the source codes. The second implementation used in the AMASS project was a Verification Server for Linux developed by Honeywell which was also briefly mentioned in Section 9. Unite currently coexists with this server and is set to replace it in the near future. The server can be extended with new tools, but doing so requires changes to the source code for every tool, and it is not publicly available. In comparison, Unite is configurable without modification of the source code, is portable (not limited to Linux), and implements the OSLC specification more completely (e.g., includes a service-provider catalogue, resource queries, etc.).

There are of course other projects whose aim covers transformation of tools into web services. Out of those, to the best of our knowledge, the one that is perhaps the closest (though not really too close) to Unite is jETI [20]. jETI is an integration platform used to combine multiple remote tools so that they can work together to provide a more complex combined functionality. jETI does not use OSLC and focuses at orchestration of the tools, which need to be used within the platform. Unite is most similar to jETI's *tool adapters* which are a small part of jETI and seem to offer a basic functionality only. Unite allows the tools to be integrated into any OSLC-compliant platforms while giving clients detailed information about the tool execution process and full control over it.

There are other implementations of adapters for the OSLC Automation specification in general. These are focused on a specific tool or only on a couple of tools. For example, the Lyo project [18] contains a JenkinsPlugin [17], although, it is a limited sample implementation according to its website and has not been updated for a few years. For a list of any other OSLC-compliant products, such as the IBM Jazz platform and Atlassian products, refer to [10].

11 CONCLUSION AND FUTURE WORK

We introduced Unite—a universal analysis adapter based on the OSLC standard. It allows one to add OSLC-compliant interfaces to non-interactive command-line analysis tools, transforming them to remotely accessible web services. Based on the practical experience with using Unite in Honeywell, we are confident that Unite can significantly ease the adoption and usage of various analysis tools, can substantially widen their user base, and can bring a number of benefits ranging from savings of time and costs to better chances of early detection and correction of defects in products.

As for future work, we are currently finishing integration of both Unite and UniC with the Eclipse Arrowhead Framework [1], which can facilitate automatic discovery of available analysis tools instead of manual client configuration. Further, Unite's implementation currently lacks user management features. This means that all clients have access to all requests, all SUTs, and all results. These features will have to be addressed when the need arises. Then, we plan on creating a publicly available web client (not an Eclipse IDE plugin, like UniC) which would allow users, who do not have experience with REST APIs, to more easily execute analysis using Unite directly. We are also planning instantiations of Unite for further tools (to show developers it is indeed easy), with the closest targets being the past-time-LTL-based runtime verification tool Spectra [30] and further experiments with the symbolic-execution-based analyser Symbiotic [28]. Furthermore, we are planning to create a *master* version of Unite which would aggregate multiple Unite instances across different servers to allow advanced, distributed computation techniques such as load balancing. Finally, we intend to upgrade the core architecture of Unite to the architecture currently used in UniC to make it even more extensible and to support more complex configuration scenarios.

ACKNOWLEDGMENTS

The work was supported by the project 20-07487S of the Czech Science Foundation, the FIT BUT internal project FIT-S-20-6427, and the H2020 ECSEL project Arrowhead Tools.

REFERENCES

[1] AHT 2022. Eclipse Arrowhead Framework. Retrieved 2022-05-18 from https://projects.eclipse.org/projects/iot.arrowhead

[2] Jiri Barnat, Jan Beran, Lubos Brim, Tomas Kratochvíla, and Petr Ročkai. 2012. Tool Chain to Support Automated Formal Verification of Avionics Simulink Designs. In *Formal Methods for Industrial Critical Systems*. Springer Berlin Heidelberg, 78–92. https://doi.org/10.1007/978-3-642-32469-7$_6$

[3] Jiří Barnat, Luboš Brim, Ivana Černá, Pavel Moravec, Petr Ročkai, and Pavel Šimeček. 2006. DiVinE – A Tool for Distributed Verification. In *Computer Aided Verification*, Thomas Ball and Robert B. Jones (Eds.). Springer Berlin Heidelberg, 278–281. https://doi.org/10.1007/11817963$_2$6

[4] Devesh Bhatt, Gabor Madl, David Oglesby, and Kirk Schloegel. 2010. Towards Scalable Verification of Commercial Avionics Software. In *Proc. of Infotech@Aerospace'10*. AIAA. https://doi.org/10.2514/6.2010-3452

[5] Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *Proc. of NFM'11 (LNCS, Vol. 6617)*. Springer. https://doi.org/10.1007/978-3-642-20398-5$_3$3

[6] Antonio Cicchetti, Federico Ciccozzi, Silvia Mazzini, Stefano Puri, Marco Panunzio, Alessandro Zovi, and Tullio Vardanega. 2012. CHESS: A Model-Driven Engineering Tool Environment for Aiding the Development of Complex Industrial Systems. In *Proc. of ASE'12*. ACM. https://doi.org/10.1145/2351676.2351748

[7] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. 2000. NUSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer* 2, 4 (March 2000), 410–425. https://doi.org/10.1007/s100090050046

[8] Jose Luis De la Vara, Eugenio Parra, Alejandra Ruiz, and Barbara Gallina. 2019. AMASS: A Large-Scale European Project to Improve the Assurance and Certification of Cyber-Physical Systems. In *Proc. of PROFES'19 (LNCS, Vol. 11915)*. Springer. https://doi.org/10.1007/978-3-030-35333-9$_4$9

[9] Jad El-khoury. 2016. Lyo Code Generator: A Model-based Code Generator for the Development of OSLC-compliant Tool Interfaces. *SoftwareX* 5 (2016), 190–194. https://doi.org/10.1016/j.softx.2016.08.004

[10] Jad El-khoury. 2020. *An Analysis of the OASIS OSLC Integration Standard, for a Cross-disciplinary Integrated Development Environment: Analysis of market penetration, performance and prospects.* Technical Report. KTH, Mechatronics.

[11] FBK ES 2022. EATA. Retrieved 2022-05-18 from https://gitlab.fbk.eu/ESProjects/EATA

[12] FBK ES 2022. FBK ES Tools. Retrieved 2022-05-18 from https://es.fbk.eu/index.php/category/tools/

[13] Jan Fiedor. 2022. VeriFIT Update Site. Retrieved 2022-05-18 from https://verifit1.fit.vutbr.cz/eclipse/

[14] J. Fiedor, M. Mužikovská, A. Smrčka, O. Vašíček, and T. Vojnar. 2018. Advances in the ANaConDA Framework for Dynamic Analysis and Testing of Concurrent C/C++ Programs. In *Proc. of ISSTA'18*. ACM. https://doi.org/10.1145/3213846.3229505

[15] Jan Fiedor and Tomáš Vojnar. 2013. ANaConDA: A Framework for Analysing Multi-threaded C/C++ Programs on the Binary Level. In *Proc. of RV'12 (LNCS, Vol. 7687)*. Springer. https://doi.org/10.1007/978-3-642-35632-2$_5$

[16] Tomáš Fiedor, Jiří Pavela, and et. al. 2022. *Perun*. Retrieved 2022-05-18 from https://github.com/tfiedor/perun

[17] Lyo 2015. Lyo JenkinsPlugin. Retrieved 2022-05-18 from https://wiki.eclipse.org/Lyo/JenkinsPlugin

[18] Lyo 2022. Eclipse Lyo. Retrieved 2022-05-18 from https://www.eclipse.org/lyo/

[19] Lyo 2022. Lyo Store. Retrieved 2022-05-18 from https://github.com/eclipse/lyo/tree/master/store

[20] Tiziana Margaria, Ralf Nagel, and Bernhard Steffen. 2005. jETI: A Tool for Remote Tool Integration. In *Proc. of TACAS'05 (LNCS, Vol. 3440)*. Springer. https://doi.org/10.1007/978-3-540-31980-1$_3$8

[21] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proc. of PLDI '07*. ACM. https://doi.org/10.1145/1273442.1250746

[22] OASIS. 2022. Open Services for Lifecycle Collaboration. https://open-services.net/.

[23] OSLC 2018. OSLC Core Version 3.0. Part 1: Overview. Retrieved 2022-05-18 from http://docs.oasis-open.org/oslc-core/oslc-core/v3.0/csprd03/part1-overview/oslc-core-v3.0-csprd03-part1-overview.html Edited by Jim Amsden. OASIS Committee Specification Draft 03 / Public Review Draft 03..

[24] OSLC 2022. OSLC Automation Version 2.1 Part 1: Specification. Retrieved 2022-05-18 from https://rawgit.com/oasis-tcs/oslc-domains/master/auto/automation-spec.html Edited by Fabio Ribeiro. OASIS Working Draft 01..

[25] OSLC 2022. OSLC Specifications. Retrieved 2022-05-18 from https://open-services.net/specifications/

[26] Postman. 2022. *Postman App: API platform (as well as a REST client)*. Retrieved 2022-05-18 from https://www.getpostman.com/

[27] Arthur Ryman. 2014. Open Services for Lifecycle Collaboration Core Specification Version 2.0 Query Syntax. Retrieved 2022-05-18 from https://archive.open-services.net/bin/view/Main/OSLCCoreSpecQuery

[28] Jiří Slabý, Jan Strejček, and Marek Trtík. 2013. Symbiotic: Synergy of Instrumentation, Slicing, and Symbolic Execution. In *Proc. of TACAS'13 (LNCS, Vol. 7795)*. Springer. https://doi.org/10.1007/978-3-642-36742-7$_5$0

[29] SmartBear. 2022. *Swagger UI*. Retrieved 2022-05-18 from https://swagger.io/tools/swagger-ui/

[30] Aleš Smrčka. 2022. Testos – Spectra: Runtime Verification of Past-time LTL. Retrieved 2022-02-21 from https://www.fit.vutbr.cz/research/groups/verifit/tools/testos-spectra/

[31] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. 2017. Theta: a Framework for Abstraction Refinement-Based Model Checking. In *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, Daryl Stewart and Georg Weissenbacher (Eds.). 176–179. https://doi.org/10.23919/FMCAD.2017.8102257

[32] Ondřej Vašíček. 2022. Unite – GitLab. Retrieved 2022-05-18 from https://pajda.fit.vutbr.cz/verifit/unite

[33] Ondřej Vašíček. 2022. Unite – Wiki. Retrieved 2022-05-18 from https://pajda.fit.vutbr.cz/verifit/unite/-/wikis/home

[34] Ondřej Vašíček. 2022. Unite Demonstration Video. Retrieved 2022-05-18 from https://nextcloud.fit.vutbr.cz/s/5tq8M7d3RPnNEDW

[35] Ondřej Vašíček, Jan Fiedor, Bohuslav Křena, Aleš Smrčka, and Tomáš Vojnar. 2022. *Unite Demonstration VM (2022)*. https://doi.org/10.5281/zenodo.6074820

[36] Don Ward. 2012. AVSI: Moving SAVI to the Launch Pad. *NDIA 2012* (2012).

[37] Robert M. Wygant. 1989. CLIPS — A powerful development and delivery expert system tool. *Computers & Industrial Engineering* 17, 1 (1989), 546–549. https://doi.org/10.1016/0360-8352(89)90121-6