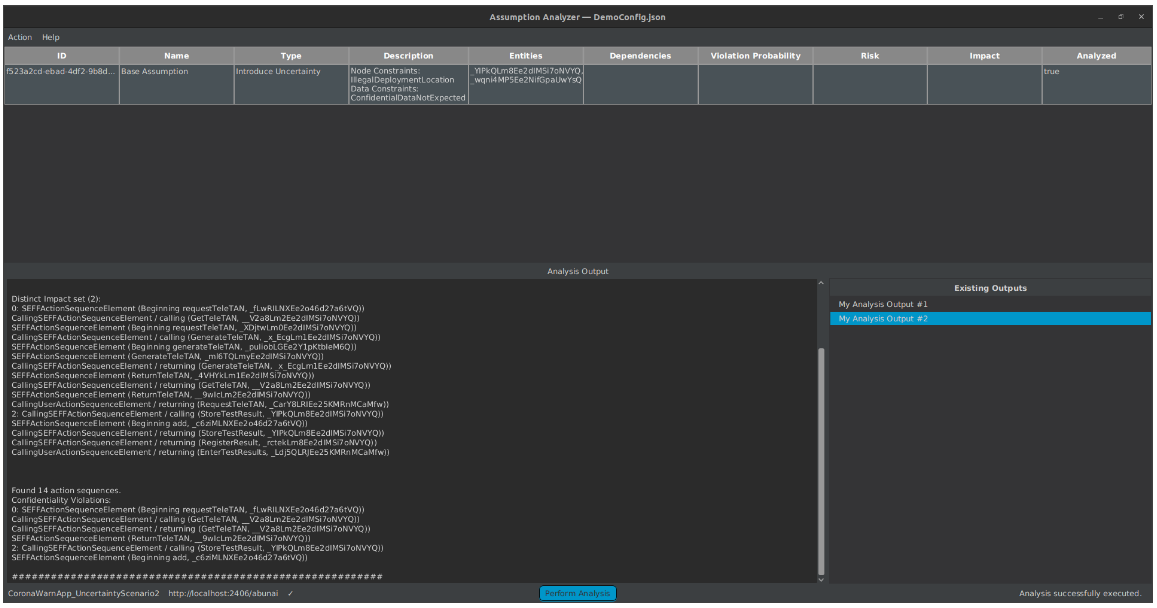
Praktikum

Werkzeuge für Agile Modellierung



ID	Name	Type	Description	Entities	Dependencies	Violation Probability	Risk	Impact	Analyzed
f523a2cd-ebad-4df2-9b8d...	Base Assumption	Introduce Uncertainty	Node Constraints: IllegalDeploymentLocation Data Constraints: ConfidentialDataNotExpected	_YIPkQLm8Ee2dIMSi7oNVYQ, _wqni4MP5Ee2NifGpaUwYsQ					true

```
Distinct Impact set (2):
0: SEFFActionSequenceElement (Beginning requestTeleTAN, _fLwRILNXEe2o46d27a6tVQ))
CallingSEFFActionSequenceElement / calling (GetTeleTAN, __V2a8Lm2Ee2dIMSi7oNVYQ))
SEFFActionSequenceElement (Beginning requestTeleTAN, _XDjtwLm0Ee2dIMSi7oNVYQ))
CallingSEFFActionSequenceElement / calling (GenerateTeleTAN, _x_EcgLm1Ee2dIMSi7oNVYQ))
SEFFActionSequenceElement (Beginning generateTeleTAN, _puliobLGEe2Y1pKtbleM6Q))
SEFFActionSequenceElement (GenerateTeleTAN, _ml6TQLmyEe2dIMSi7oNVYQ))
CallingSEFFActionSequenceElement / returning (GenerateTeleTAN, _x_EcgLm1Ee2dIMSi7oNVYQ))
SEFFActionSequenceElement (ReturnTeleTAN, _4VHYkLm1Ee2dIMSi7oNVYQ))
CallingSEFFActionSequenceElement / returning (GetTeleTAN, __V2a8Lm2Ee2dIMSi7oNVYQ))
SEFFActionSequenceElement (ReturnTeleTAN, __9wIcLm2Ee2dIMSi7oNVYQ))
CallingUserActionSequenceElement / returning (RequestTeleTAN, _CarY8LRIEe25KMRnMCaMfw))
2: CallingSEFFActionSequenceElement / calling (StoreTestResult, _YIPkQLm8Ee2dIMSi7oNVYQ))
SEFFActionSequenceElement (Beginning add, _c6ziMLNXEe2o46d27a6tVQ))
CallingSEFFActionSequenceElement / returning (StoreTestResult, _YIPkQLm8Ee2dIMSi7oNVYQ))
CallingSEFFActionSequenceElement / returning (RegisterResult, _rctekLm8Ee2dIMSi7oNVYQ))
CallingUserActionSequenceElement / returning (EnterTestResults, _Ldj5QLRJEe25KMRnMCaMfw))



Found 14 action sequences.
Confidentiality Violations:
0: SEFFActionSequenceElement (Beginning requestTeleTAN, _fLwRILNXEe2o46d27a6tVQ))
CallingSEFFActionSequenceElement / calling (GetTeleTAN, __V2a8Lm2Ee2dIMSi7oNVYQ))
CallingSEFFActionSequenceElement / returning (GetTeleTAN, __V2a8Lm2Ee2dIMSi7oNVYQ))
SEFFActionSequenceElement (ReturnTeleTAN, __9wIcLm2Ee2dIMSi7oNVYQ))
2: CallingSEFFActionSequenceElement / calling (StoreTestResult, _YIPkQLm8Ee2dIMSi7oNVYQ))
SEFFActionSequenceElement (Beginning add, _c6ziMLNXEe2o46d27a6tVQ))

##########################################################
```

```
CoronaWarnApp_UncertaintyScenario2   http://localhost:2406/abunai   ✓
Analysis successfully executed.
```

Anforderungsprüfung

Tim Bächle

Institute of Information Security and Dependability (KASTEL)
Advisor: M.Sc. Sophie Corallo

Contents

1 Motivation and Objective

It is often difficult to verify that every requirement for a software system is met. This is particularly the case in the context of large and complex systems, where different parts are commonly developed by independent teams. Consequently, evaluating whether the system as a whole meets a requirement is not a straightforward task.

While requirements dictate many quality attributes of a system, nowadays, security has become one of the most important. Among other things, this is due to the fact that violations with regard to security requirements can have particularly severe consequences. From a legal perspective, security is often mandated through strict laws, whose violation is sanctioned with substantial fines. However, more damaging to a company is sometimes the loss of trust in the system by its customers and various stakeholders.

Thus, identifying potential security violations is a top priority during any development process. Given that the cost of implementing changes within software tends to grow as development time increases, it is obvious that such violations should be identified as early as possible.

So-called security analyses try to address this problem by allowing developers to analyze the system with respect to various security dimensions as early as the first architecture model. For instance, the security analysis Abunai [4] aids developers in identifying illegal data flows within the system, which could lead to confidentiality violations. Internally, Abunai achieves this by analyzing the system's component model.

With analyses commonly specializing in different security dimensions (such as confidentiality), it comes as no surprise that they differ in various aspects. This can mean that technical aspects, such as the installation process, differ, but, more importantly, that analyses are operated differently. As a result, to analyze a system with regard to multiple security dimensions, a developer first has to familiarize himself with the chosen analyses. Evidently, this is not only a tedious process but also consumes time that would be better spent on analyzing the system. To address this problem, we propose Assumption Analyzer, a stand-alone application that aims to bridge the gap between a developer and various security analyses. The underlying idea of this application is to provide a universal yet concise user interface, allowing users to interact with (potentially remote) security analysis. To allow Assumption Analyzer to be decoupled from individual analyses, we also propose a simple REST API specification, which can easily be added to an existing security analysis. This API allows security analyses to act as independent microservices, and, crucially, enables Assumption Analyzer to connect to different analyses without requiring information about their internals.

As of now, Assumption Analyzer is still a prototype. Therefore, only the security analysis Abunai [4] (specializing in confidentiality) has been extended with the aforementioned REST API. In addition to the source code for Assumption Analyzer [3], we provide both the source code for the customized analysis [5] as well as a Docker[1] container image [1], which simplifies the installation.

[1]`https://www.docker.com/`

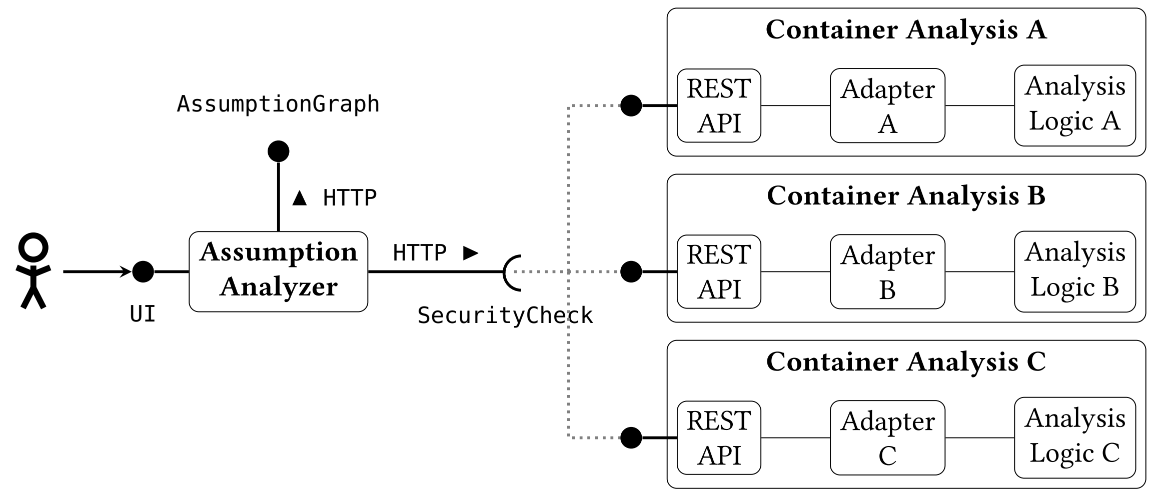


Figure 1: The architecture of Assumption Analyzer

2 Architecture

The architecture of Assumption Analyzer is depicted in Figure 1. It is designed around the idea of a stand-alone application providing a uniform front-end for various security analyses (cf. analyses `A`, `B`, and `C` from Figure 1). To allow for easy integration with different analyses, Assumption Analyzer proposes a simple REST API, called `SecurityCheck`, that provides a layer of abstraction between Assumption Analyzer and the specifics of individual security analyses. This API is outlined in Table 1 and can easily be added to an existing analysis. In the exemplary case of Abunai [4], this was achieved using Java Spark,[2] a micro-framework for creating REST APIs with minimal need for boilerplate code. This overall approach allows a security analysis to be completely decoupled from Assumption Analyzer and any immediate user interaction. Moreover, considering that the entire communication is built upon HTTP, it does not constrain analyses to be located on the same machine as the front-end. Instead, it is, for instance, possible to have an Assumption Analyzer instance access analyses hosted on remote machines. However, for this abstraction layer, it is important to note that every security analysis requires a dedicated adapter (cf. Figure 1) that translates between the data specified within Assumption Analyzer and the data required for the particular analysis. As an outline for the implementation of such an adapter as well as the REST API, we provide the interface `SecurityCheckAdapter.java`[3] and abstract class `RestConnector.java`[4] (both located in [5]).

From a technical perspective, Assumption Analyzer regularly checks the connection to the analysis specified by the user as part of his particular application configuration via the `/test` endpoint. Upon analysis execution, Assumption Analyzer first transmits the files found within the specified model folder to the security analysis via the

[2] https://sparkjava.com/

[3] https://github.com/TDot305/UncertaintyImpactAnalysis/blob/8a3184ea7f54d50ac85b73b6067d9bdbabc07229/tests/dev.abunai.impact.analysis.tests/src/rest/general/SecurityCheckAdapter.java

[4] https://github.com/TDot305/UncertaintyImpactAnalysis/blob/8a3184ea7f54d50ac85b73b6067d9bdbabc07229/tests/dev.abunai.impact.analysis.tests/src/rest/general/RestConnector.java

Endpoint	Functionality
`/test`	Allows Assumption Analyzer to test the connection to a particular analysis microservice. **Inputs**: None. **Outputs**: A plain text response with HTTP code 200 on connection success.
`/set/model/:modelName`	Allows Assumption Analyzer to transmit a component model with the name `modelName` to the analysis micoservice. The individual files that make up the model are sent as a multipart request, where each part represents one file. **Note**: For this initial prototype, Assumption Analyzer has been tailored towards Palladio Component Models (PCMs). **Inputs**: The model name (i.e., name of the model directory) via the endpoint path and the files constituting the views of the PCM as the multipart objects. **Outputs**: A plain text response with HTTP code 200 on transmission success.
`/run`	Allows Assumption Analyzer to execute the actual analysis associated with a particular analysis microservice. The request contains an `AnalysisParameter` argument and expects an `AnalysisOutput` instance as its response. **Inputs**: A JSON serialization of an `AnalysisParameter` (declaration in [2]) instance, i.e., a serialized object specifying the PCM model path and the collection of `SecurityCheckAssumptions` that should be used for the analysis. **Outputs**: A JSON serialization of an `AnalysisOutput` (declaration in [2]) instance, i.e., a serialized object specifying a textual log of the analysis output as well as the collection of potentially altered `SecurityCheckAssumptions` (for instance, an assumption could set the analyzed property of the `SecurityCheckAssumptions` to `true`).

Table 1: The `SecurityCheck` REST API proposed by Assumption Analyzer

`/set/model/:modelName` endpoint. This ensures that even on a remote machine, the analysis has access to the model files. While some of the files might suffice for the analysis, Assumption Analyzer always transmits all files as it is not aware of the way a particular security analysis processes incoming data.

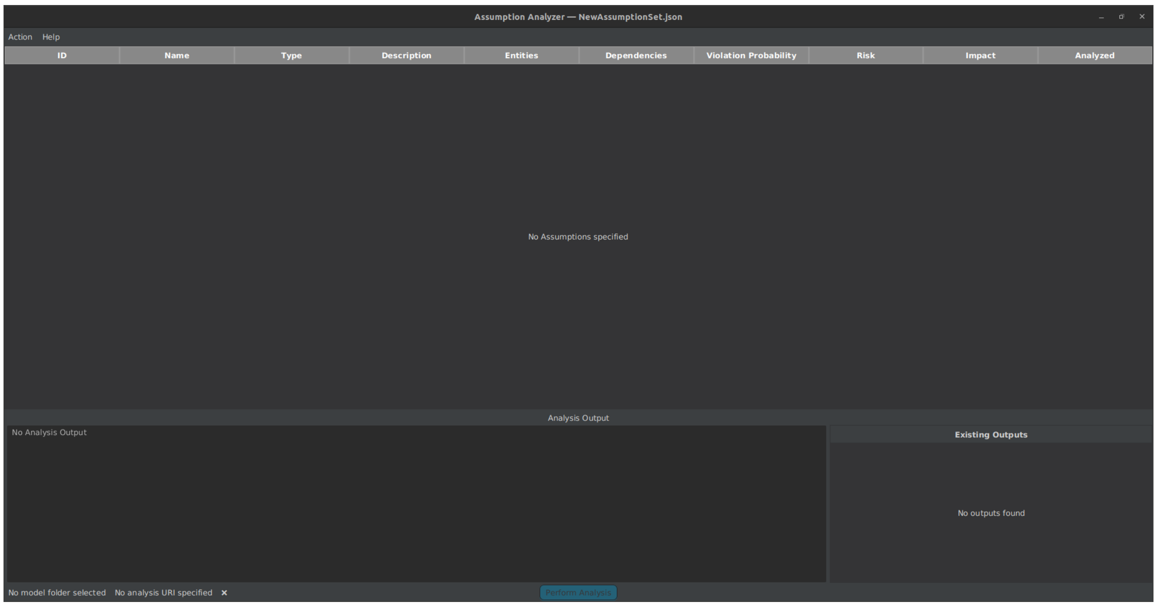
After the model has been successfully transferred, Assumption Analyzer initiates a second transfer via the `/run` endpoint, containing all parameters that are potentially relevant for specifying an analysis scenario. This includes both the path of the model and the collection of `SecurityCheckAssumptions` on which the analysis should take place (cf. Table 1). These parameters are referred to as potentially relevant, as, similarly to the transferred model files, not all properties contained in the `SecurityCheckAssumptions` are relevant for every analysis.

The REST API of the receiving analysis microservice then forwards this data specifying the analysis scenario to a dedicated adapter (cf. Figure 1). This adapter is aware of the internal logic of the particular security analysis and is, therefore, able to extract the data that is actually relevant for the security analysis and transform it into a suitable format. Lastly, the adapter triggers the actual analysis of the scenario described by this data and passes the result, in the form of a textual log and potentially altered `SecurityCheckAssumptions` (cf. Table 1), back to the REST API. The REST API then sends an appropriate answer, containing the serialized analysis result, back to Assumption Analyzer.

Conceptually, `SecurityCheckAssumptions` only contain a subset of the properties that can be specified for an assumption within Assumption Analyzer. This is due to the fact that some of the properties (e.g., relationships between individual assumptions) are tailored towards a higher-level analysis. To allow access to these additional properties, Assumption Analyzer provides an additional `AssumptionGraph` REST API (cf. Figure 1). This API gives full access to the current application configuration, including both the specified collection of assumptions with all their properties (implemented as `GraphAssumptions`) and any previous analysis outputs. As a result, in the future, other tools could leverage the information collected within textscAssumption Analyzer.

3 Usage

Upon opening Assumption Analyzer, the user is prompted with the screen shown in Figure 2. This screen acts as the center point of the application and prominently shows both the specified assumptions (upper section) as well as potential analysis results (lower section).



No Assumptions specified

No Analysis Output

No outputs found

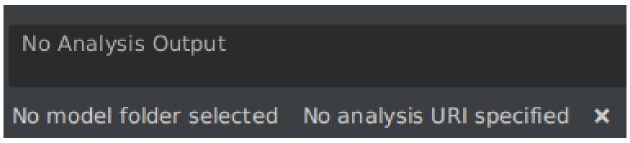
No model folder selected No analysis URI specified

Figure 2: Assumption Analyzer on start-up

3.1 Typical Workflow

To perform an analysis with Assumption Analyzer, the workflow usually looks as follows:

1. First, the user specifies both the PCM on which the analysis should take place as well as the URI on which the microservice of the desired security analysis can be found.
   a) The PCM, on which the analysis should take place, is specified via the leftmost field shown in Figure 3. It is located in the lower left corner of Assumption Analyzer's main screen (cf. Figure 2). Upon a click, the user is prompted to select the folder in which the desired model resides.



No Analysis Output

No model folder selected No analysis URI specified

Figure 3: The labels for model and analysis specification

   b) Similarly, the URI of the desired analysis (e.g., `http://localhost:2406/abunai`) is specified via a click on the field directly adjacent to the right. This field not only shows the currently selected URI but also a small indicator for the connection status to the analysis. The ✗ symbol indicates an interrupted connection, while the ✓ symbol indicates a healthy connection.

2. The user then adds the desired assumptions. To do so, the user clicks on the `Action` button in the menu bar at the top of Assumption Analyzer's main screen and selects the `New Assumption` menu item located within. This brings up the additional screen shown in Figure 4.

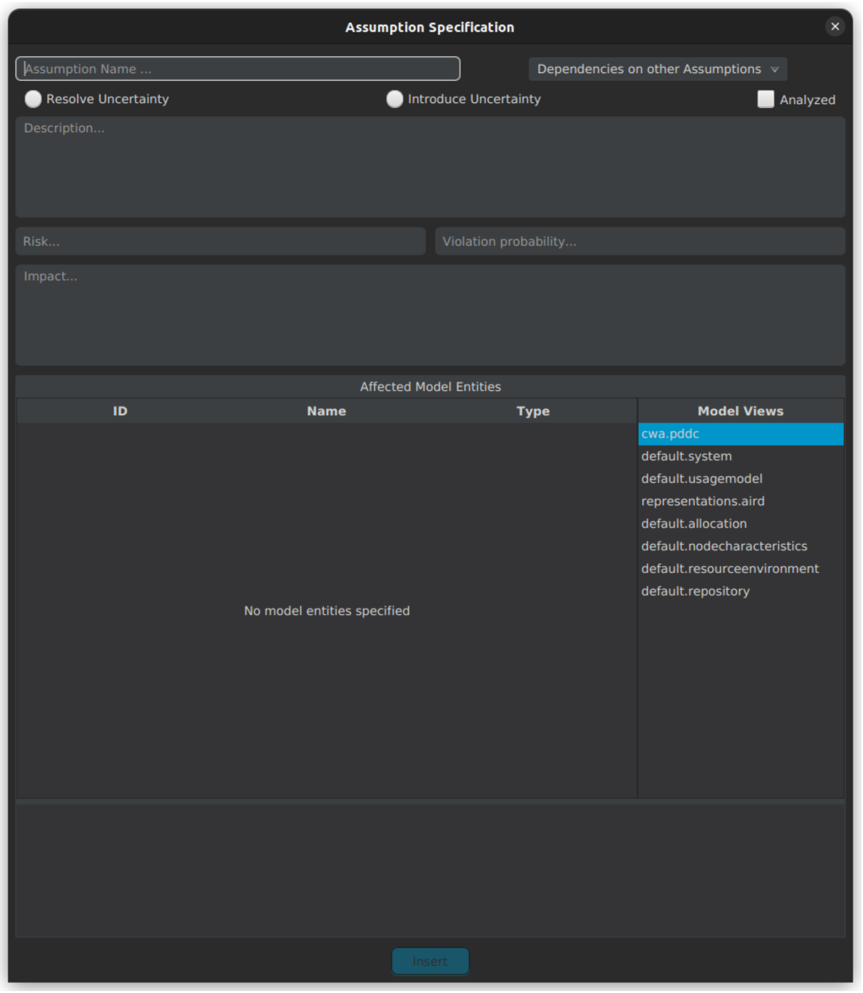


Figure 4: The assumption specification screen

Within this screen, the user can specify the assumption properties described below:

☑ Mandatory properties are:

- `Name`: The name of the assumption.
- `Type`: The type of the assumption, i.e., either `Resolve Uncertainty` or `Introduce Uncertainty` (can be specified via the two radio buttons).
- `Description`: A textual description of the assumption.

☐ Optional properties are:

- `Dependencies`: Currently, Assumption Analyzer allows dependencies between individual assumptions to be specified via the drop-down in the upper right corner. Selecting one of the other assumptions represents a dependency from the currently specified assumption to the selected one.
- `Analyzed`: A property that can be manually toggled (if desired) but is more commonly set by a particular security analysis during analysis execution.
- `Risk`: The risk associated with the assumption.

  ❗ Note that this property only accepts numerical values (e.g., 50, 3.14, 50E-1).
- `Violation Probability`: The probability of violation for the assumption.

  ❗ Note that this property only accepts numerical values (e.g., 50, 3.14, 50E-1).
- `Impact`: A textual description of the assumption's impact.
- `Affected Model Entities`: The PCM entities affected by the assumption can be specified by choosing one of the available model views from the list presented below the `Model Views` header. This populates the tree view shown at the bottom of the screen, which can then be expanded to browse through all the entities contained within the selected model view. By right-clicking an entity, one is presented with the `Add to Affected Model Entities` option. Upon clicking this option, the entity is added to the table of affected entities prominently shown in the center of the screen.

  ℹ Affected entities can be removed from the table by right-clicking the entity that should be removed and selecting `Remove Model Entity`.

  ❗ Note that entities without a valid ID (visually grayed out) cannot be added.

The optional properties that should be specified obviously depend on the concrete security analysis chosen during step item 1b.

3. Once the mandatory properties have been specified, the blue `Insert` button (shown below) at the bottom of the screen lights up and, upon a click, adds the newly specified assumption to the current configuration.

4. As soon as at least one assumption is specified (along with the analysis URI and model folder), the blue `Perform Analysis` button (shown below) is enabled.

   ℹ **Note** that the `Perform Analysis` button is only activated once a valid model path, analysis URI, and at least one assumption have been specified.

5. A click on the `Perform Analysis` button then initiates the analysis on the previously specified (remote) security analysis microservice. Once the analysis is finished, the result is shown in the lower section of the screen.

   If previous results for the configuration are present, one can switch between them by clicking on the respective entries below the `Existing Outputs` header.

3.2 Using Abunai as the Security Analysis

3.2.1 Starting the Analysis Microservice

Before Abunai can be used within Assumption Analyzer, its microservice has to be started. This can be achieved in one of two ways:

1. Clone the extended Abunai confidentiality analysis repository [5] and follow the installation steps in the included `README`. Afterwards, allow Eclipse to resolve the maven dependencies of `dev.abunai.impact.analysis.tests` by right-clicking on the project and then selecting `Maven > Update Project`

   Note: Make sure that the Maven plugin for Eclipse is installed.

   The microservice can then be started by executing `RestConnector.java`, located in the `rest`-package of `dev.abunai.impact.analysis.tests`.

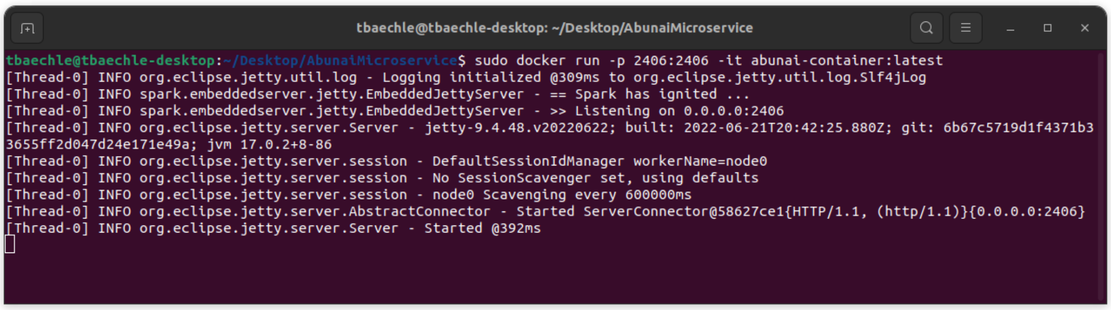
2. Use the dedicated Abunai Docker container and launch it via:

   ```
   docker run -p 2406:2406 -it abunai-container:latest
   ```

   Refer to Section 3.2.2 for instructions on how to obtain or build the Abunai container.

Once the microservice is successfully started, the console should show an output similar to the one shown in Figure 5.

Note that by default, the Analysis microservice is started on port `2406`.



```
tbaechle@tbaechle-desktop:~/Desktop/AbunaiMicroservice$ sudo docker run -p 2406:2406 -it abunai-container:latest
[Thread-0] INFO org.eclipse.jetty.util.log - Logging initialized @309ms to org.eclipse.jetty.util.log.Slf4jLog
[Thread-0] INFO spark.embeddedserver.jetty.EmbeddedJettyServer - == Spark has ignited ...
[Thread-0] INFO spark.embeddedserver.jetty.EmbeddedJettyServer - >> Listening on 0.0.0.0:2406
[Thread-0] INFO org.eclipse.jetty.server.Server - jetty-9.4.48.v20220622; built: 2022-06-21T20:42:25.880Z; git: 6b67c5719d1f4371b3
3655ff2d047d24e171e49a; jvm 17.0.2+8-86
[Thread-0] INFO org.eclipse.jetty.server.session - DefaultSessionIdManager workerName=node0
[Thread-0] INFO org.eclipse.jetty.server.session - No SessionScavenger set, using defaults
[Thread-0] INFO org.eclipse.jetty.server.session - node0 Scavenging every 600000ms
[Thread-0] INFO org.eclipse.jetty.server.AbstractConnector - Started ServerConnector@58627ce1{HTTP/1.1, (http/1.1)}{0.0.0.0:2406}
[Thread-0] INFO org.eclipse.jetty.server.Server - Started @392ms
```

Figure 5: Initial console output of the started microservice

3.2.2 The Abunai Container

The image of the Abunai Docker container is available as a `tar` archive [1]. This image can be imported into Docker with the following command:

```
sudo docker import <PathToContainerImage>/AbunaiContainerImage
```

Alternatively, it is possible to build the container from scratch. To do so, make sure to clone the repository as described in Section 3.2.1. As the Abunai microservice relies on both external and inter-project dependencies, exporting the analysis as a single `jar` file can be troublesome. Therefore, we provide `AbunaiDependencyCopier.java`, a simple Java utility program for copying all required dependencies into a specified target directory. As input, it expects two command-line parameters:

1. The absolute path to the target directory.
2. The classpath string specifying Abunai's dependencies.

After the microservice has been started at least once from within Eclipse, it is possible to extract the classpath string by selecting `Run > Run Configurations > Show Command Line`. The string listed after `-classpath` is the classpath string that can be used as input for `AbunaiDependencyCopier.java`.

Once all dependencies are copied to the desired directory, one can use the `Dockerfile` located in the `resources/abunai` directory of the Assumption Analyzer repository [3] to build the container. This `Dockerfile` is also depicted in Listing 1.

Note that *`<MyAbunaiDependencyDirectory>`* in line 5 has to be replaced with the name of the target directory used with `AbunaiDependencyCopier.java` (assuming that the Dockerfile is placed in the same directory).

```
1 # Use a base image with a Java runtime.
2 FROM openjdk:17
3
4 # Copy compiled class files and resources into container.
5 COPY <MyAbunaiDependencyDirectory> AbunaiDependencies
6
7 # Set the main class and the runtime arguments.
8 ENTRYPOINT ["java", "-Dfile.encoding=UTF-8", "-classpath", "
    AbunaiDependencies/Palladio-Addons-DataFlowConfidentiality-Analysis/
    bundles/org.palladiosimulator.dataflow.confidentiality.analysis/bin:
    AbunaiDependencies/UncertaintyImpactAnalysis/bundles/dev.abunai.impact.
    analysis/bin:AbunaiDependencies/UncertaintyImpactAnalysis/tests/dev.
    abunai.impact.analysis.tests/target/classes:AbunaiDependencies/
    UncertaintyImpactAnalysis/tests/dev.abunai.impact.analysis.testmodels/bin
    :AbunaiDependencies/external_dependency_jars/*", "-XX:+
    ShowCodeDetailsInExceptionMessages", "rest.AbunaiConnector"]
```

Listing 1: The Dockerfile for creating the Abunai container

To build the container, one can then use the following docker command:

```
sudo docker build -t abunai-container:latest <PathToDirContainingDockerfile>
```

3.2.3 Assumption Specification

When specifying assumptions for Abunai, there are two things to consider:

1. An assumption's `Description` specifies the constraints used in the confidentiality analysis. For this, as of now, the description has to follow a structure similar to:

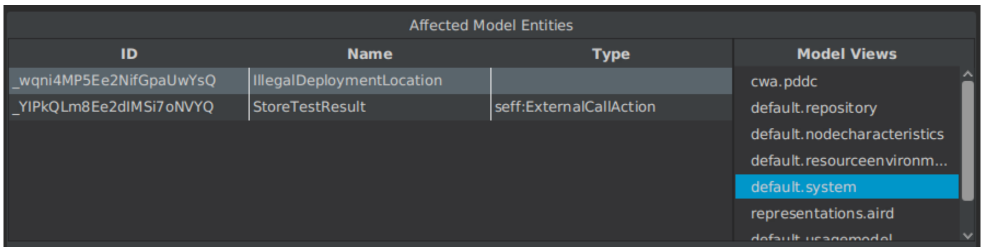
   ```
   Node Constraints:  Constraint1, Constraint2, ..., ConstraintN
   Data Constraints:  Constraint1, Constraint2, ..., ConstraintN
   ```

   Where the order in which the constraint types (node or data) are specified is not relevant. An example of how such a specification could look like is shown below:

   ```
   Data Constraints:  ConfidentialDataNotExpected, ValidationFailed
   Node Constraints:  Laboratory
   ```

   In the future, `AbunaiAdapter.java` could also be extended to use natural language processing to automatically extract the corresponding constraints without requiring a fixed input format.

2. At least one assumption should specify model entities as part of the `Affected Model Entities` property (cf. Figure 6). Otherwise, Abunai cannot determine any uncertainty sources for the analysis.



Affected Model Entities

ID	Name	Type
_wqni4MP5Ee2NifGpaUwYsQ	IllegalDeploymentLocation	
_YIPkQLm8Ee2dIMSi7oNVYQ	StoreTestResult	seff:ExternalCallAction

Figure 6: An assumption specifying two affected model entities

3.3 Additional Functionality and Hints

3.3.1 Renaming an analysis result

A previous analysis result can be renamed by double-clicking its entry below the `Existing Outputs` header. The desired name can then be entered.

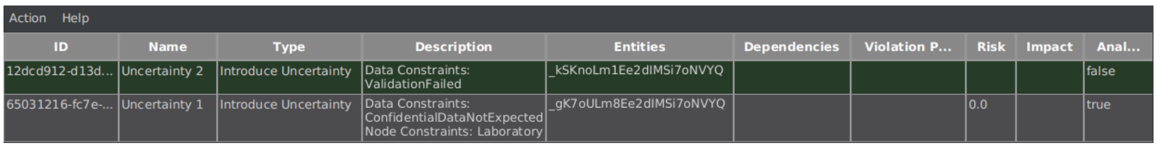
Note that the new name has to be unique within the current configuration. Otherwise, the name is reset to its previous value.

3.3.2 Editing an existing assumption

An assumption can be edited by either right-clicking and selecting `Edit Assumption` or by double-clicking its corresponding row within the table on Assumption Analyzer's main screen. Both ways bring up the assumption specification screen (cf. Figure 4), where all properties of the assumption can be edited.

3.3.3 Marking an assumption as manually analyzed

As a visual aid to the user, assumptions can be marked as manually analyzed, indicating that they have already been taken into account. This is achieved by right-clicking an assumption and selecting `Toggle Manually Analyzed`. Consequently, the assumption will now be highlighted in green, as seen in Figure 7.



ID	Name	Type	Description	Entities	Dependencies	Violation P...	Risk	Impact	Anal...
12dcd912-d13d...	Uncertainty 2	Introduce Uncertainty	Data Constraints: ValidationFailed	_kSKnoLm1Ee2dIMSi7oNVYQ					false
65031216-fc7e-...	Uncertainty 1	Introduce Uncertainty	Data Constraints: ConfidentialDataNotExpected Node Constraints: Laboratory	_gK7oULm8Ee2dIMSi7oNVYQ			0.0		true

Figure 7: Within a set of assumptions, one is marked as manually analyzed

4 Selected Design Challenges

4.1 Integration of Analyses

One of the main challenges during the design of Assumption Analyzer was determining how the application could integrate with various different security analyses. From a theoretical viewpoint, the simple solution of providing a general front-end and integrating it directly into the analyses would have been possible. However, this approach would result in a distinct Assumption Analyzer instance for every analysis. When using multiple analyses, this would mean that one has to switch between different applications, which defeats the main purpose of Assumption Analyzer, previously outlined in Section 1. Maintainability is another problem with this approach, seeing that any changes to Assumption Analyzer would have to be manually added to all security analyses.

Another alternative, we considered, was to equip each analysis with a dedicated command-line interface (CLI). Upon specification of the path to an analysis executable, Assumption Analyzer would then be capable of starting the executable and issuing requests via the CLI. While this approach is in line with the goal of allowing smooth integration with a wide variety of analyses, it also has some drawbacks. Not only can it be complex to integrate a fully functional CLI into an existing security analysis, but it also poses a number of limitations. For instance, every analysis would need to be located somewhere on the same machine on which Assumption Analyzer is run. Additionally, this location would not be allowed to change after it has been registered with Assumption Analyzer. Otherwise, it is impossible to (re-)locate the analysis, and the user would need to enter the respective path anew.

To avoid these drawbacks, we opted for a third alternative, which involves extending each analysis with a dedicated REST API, thus enabling the analyses to act as independent microservices. With the wide availability of REST frameworks, this approach is less complex than incorporating a custom CLI. Furthermore, it is far more flexible, as an analysis is now addressed by a URI instead of an error-prone absolute path in the file system. Lastly, this approach comes with some of the usual benefits of a microservice architecture. For instance, with Assumption Analyzer communicating with the analyses solely through HTTP, analyses can be deployed on other (potentially remote) machines. Especially in the case of resource-intensive analyses or large models, this can make life easier, as the analysis microservice can be hosted on a separate server or workstation.

4.2 Storing User Data and Detecting Unsaved Changes

Software whose functionality heavily revolves around user input inherently faces the challenges of how to save this data and how to detect unsaved changes. For saving structured data, file formats such as XML or JSON are commonly employed. While Assumption Analyzer started out using XML files for storing its configuration (i.e., assumptions, PCM path, etc.), it quickly transitioned to JSON due to the high availability and quality of available JSON object-mappers. Additionally, with the `SecurityCheck`-API (cf. Section 2) already relying on JSON serialization for transmitting data to the individual security analyses, a suitable object mapper was already available within the project.

For detecting unsaved changes (e.g., in case the user issues a close request via the X-button of the window), the application started out by using a simple boolean flag that was set in case any property of the application configuration had changed since the last save operation. While this approach was simple, it quickly became apparent that it was not ideal when it came to evolving and maintaining the application. This is due to the fact that at every location in the code where the application configuration is altered, the flag needs to be set. With the application changing, the number of such locations can obviously change, thus introducing errors once the flag is not set accordingly. To avoid this problem, we identified two potential alternatives:

1. Read the last saved configuration from the save file and compare its contents to the serialization of the current configuration. If they are not identical, the current configuration contains unsaved changes.

2. Always keep two copies of the application configuration in memory. One that can be altered by the user by issuing inputs to the application, and one that stores the last state that was persisted in the save file. Unsaved changes are present once these two copies do not contain identical contents.

After considering both alternatives for Assumption Analyzer, we opted for the second alternative. While the first alternative provides an easy way to detect unsaved changes in the form of comparing the serialization strings, it is obviously prone to subtle errors. If, for instance, the order in which the object mapper serializes individual fields changes after an update of the corresponding framework, this approach would incorrectly report unsaved changes when a comparison is made to an older save file.

In this regard, the second alternative is certainly more robust. However, it also comes with some drawbacks. First of all, the total memory demand of the application is increased as the application configuration is kept in memory twice. Also, the required mechanism for allowing a configuration to be cloned is more complex compared to just comparing the serialization results as in the first alternative. In addition, this mechanism needs to be adjusted when the underlying data type representing a configuration is changed (e.g., a new field also needs to be copied during a clone operation).

References

[1] Tim Bächle. *Abunai Microservice Docker Container Image.* 2023. URL: `https://gitlab.kit.edu/kit/kastel/sdq/stud/praktika/sose2023/timnorbertbaechle/-/blob/9fec5af463b8ea14aa859e1f4a6c90129f779261/src/main/resources/abunai/AbunaiContainerImage` (visited on 09/10/2023).

[2] Tim Bächle. *AnalysisParamater and AnalysisOutput Data Types.* 2023. URL: `https://github.com/TDot305/UncertaintyImpactAnalysis/blob/419ef1fdda8ca5e424dca8069b1cb5a3abacbf47/tests/dev.abunai.impact.analysis.tests/src/rest/general/RestConnector.java` (visited on 09/17/2023).

[3] Tim Bächle. *Assumption Analyzer.* 2023. URL: `https://gitlab.kit.edu/kit/kastel/sdq/stud/praktika/sose2023/timnorbertbaechle` (visited on 09/09/2023).

[4] Sebastian Hahner. *Architecture-Based Uncertainty-Aware Confidentiality Analysis.* 2023. URL: `https://github.com/abunai-dev` (visited on 08/23/2023).

[5] Sebastian Hahner and Tim Bächle. *Architecture-Based Uncertainty-Aware Confidentiality Analysis.* 2023. URL: `https://github.com/TDot305/UncertaintyImpactAnalysis` (visited on 08/31/2023).