Network of Excellence on
Engineering Secure Future Internet
Software Services and Systems

Network of Excellence

Deliverable D2.2

First Release of the SDE for Security-Related Tools

Project Number	:	256980
Project Title	:	NESSoS
Deliverable Type	:	Report

Deliverable Number	:	D2.2
Title of Deliverable	:	First Release of the SDE for Security-Related Tools
Nature of Deliverable	:	R
Dissemination Level	:	Public
Internal Version Number	:	1.0
Contractual Delivery Date	:	M12
Actual Delivery Date	:	31.10.2011
Contributing WPs	:	WP5-WP9
Editor(s)	:	Marianne Busch (LMU), Nora Koch (LMU)
Author(s)	:	Marianne Busch (LMU), Alexander Concha (INRIA), Bart Jacobs (KUL), Nora Koch (LMU), Holger Schmidt (UDE), Bjørnar Solhaug (SINTEF), Mathieu Turuani (INRIA)
Reviewer(s)	:	Marinella Petrocchi, Pierluigi Roberti

Abstract

Within the NESSoS project, a variety of analysis and development tools for different aspects of service security engineering will be used. We identify existing and tools under development for security engineering, evaluating and comparing them to produce a comprehensive overview (see D2.1 [19]). However, we go one step further, we ensure that our tools can be used within a common environment – a so-called tool workbench or tool integration platform. This will increase the speed of the development as well as the level of security of service-oriented software and systems.

In a first step, we identify the requirements such a tool workbench should satisfy and briefly present related work. Subsequently, we describe the functionality and technical features of the selected tool workbench, and finally, we provide an overview of the tools already integrated in the common environment.

The selected platform, the Service Development Environment (SDE) provides a suitable basis for tool integration that supports the development of service-oriented software. On this platform, tools are services that can be published and discovered; they provide arbitrary functionality and can be used as-is, or combined defining new services.

By integrating NESSoS tools into the SDE, individual tools become available to a broader user range and in a larger context easing the engineering of secure software. The SDE is used for integrating all kinds of development utilities, regardless of platform or programming language.

The SDE is based on the Eclipse IDE and enables a simple and light-weight integration of tools providing a common user interface and orchestration mechanisms, i.e., the combined use of tools for solving a problem. In the view of the SDE, tools each consist of functions, which can be invoked in the SDE using a text or graphic editor.

The tool integration into the SDE started with the following NESSoS-related tools: First integrated SDE tools that are NESSoS-related are: Avantssar-atse (CL-ATSE), Avantssar Orchestrator, CORAS Tool, EOS (Eye OCL Software), Jalapa, MagicUWE, UML4PF and VeriFast.

Keyword List

Service Development Environment, IDE, Integrated Development Environment, Tool Integration, Tool Workbench

Document History

Version	Type of Change	Author(s)
0.1	Initial description of UML4PF	Holger Schmidt (UDE)
0.2	First version of SDE description	Marianne Busch, Nora Koch (LMU)
0.3	Avantssar-atse and Avantssar orchestrator descriptions	Alexander Concha (INRIA)
0.4	Verifast description	Bart Jacobs (KUL)
0.5	Improved related work, MagicUWE description	Marianne Busch (LMU)
0.6	Improved version requirements and SDE	Nora Koch (LMU)
0.7	Improved version	Nora Koch, Marianne Busch (LMU)
0.8	Improved CL-AtSe & Orchestrator	Mathieu Turuani (INRIA)
0.9	Acronyms	Marianne Busch (LMU)
1.0	General improvements	Marianne Busch, Nora Koch (LMU)
3.0	Accepted by the internal reviewers	

Table of Contents

List of Acronyms

API Application Programming Interface

BPMN Business Process Modeling Notation

CASE Computer-Aided Software Engineering

CBK Common Body of Knowledge

CPL Common Public License

GUI Graphical User Interface

IDE Integrated Development Environment

LTL Linear Temporal Logic

NESSoS Network of Excellence on Engineering Secure Future Internet Software Services and Systems

OCL Object Constraint Language

PDE Plug-in Development Environment

RIA Rich Internet Application

SDE Service Development Environment

SOA Service-Oriented Architecture

UI User Interface

UML Unified Modeling Language

UML4PF UML for Problem Frames

URL Uniform Resource Locator

UWE UML-based Web Engineering

WSDL Web Services Description Language

List of Figures

List of Tables

1 Introduction: The aim of the SDE

Within the NESSoS project, a variety of tools is under construction which aid developers in tackling security-related problems. These tools are not only developed at different sites, but are also vastly different with regard to user interface, functionality, required computing power, execution platform and programming language. However, all of the tools contribute to the development life cycle of secure software and provide results which may serve as input to other tools. Thus, developers should be able to use these tools in combination – generating so-called tool chains – to automate as many tasks as possible.

Consequently, an integration approach is needed which allows tool developers to easily add their tools to a service-oriented workbench – also called integrated platform – because tools are integrated as services. We have decided that the Service Development Environment (SDE) [47], formerly known as Sensoria Development Environment, meets our needs best. The SDE was developed within the Sensoria project [51], a FET initiative funded by the EU from 2005 to 2010. It is currently maintained and extended by LMU within the scope of the NESSoS and the ASCENS [3] projects.

The aim is to enable users of the SDE to discover and seamlessly use all kind of tools integrated in the platform. The objective of both projects is to integrate all the tools developed and used by the project partners.

The SDE is an Integrated Development Environment (IDE), which is based on a Service-Oriented Architecture (SOA) approach, where each tool is represented as a service. Furthermore, it provides the ability to compose new tools out of existing ones, a process known as orchestration in the world of services.

A simple example of such a tool chain is depicted in Figure 1.1. We assume that we create a model using a `Specification`-Tool. This model is then passed to an `Analysis` tool, e.g., a model checker. The user of this tool chain does not have to be familiar with how to provide this model and how to start the model checker, as the tool chain is executed automatically. Creating a new service as orchestration of other existing services is possible using either a JavaScript-based approach, or a graphical workflow approach, as described in more detail in section 3.3.

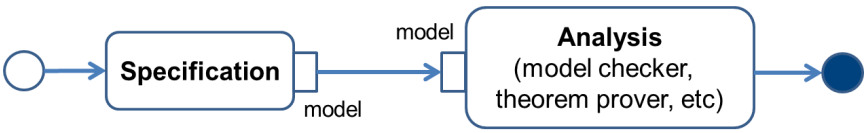

Figure 1.1: Tool chain

Another platform developed within the NESSoS project is the Common Body of Knowledge (CBK) [20, 28]. The CBK contains all relevant information on (new) methods, techniques, notations and tools providing an overview of these so-called knowledge objects. The CBK has been populated first with methods, techniques, notations and tools developed by the partners of the project and members of the NESSoS Network of Excellence. The use of both platforms – CBK and SDE – enables the combination of the wide-ranging applicability of the SDE with the structured information about tools that is stored in the CBK.

The CBK moreover contains information about all tools – not only those that are already integrated in the SDE. Furthermore, the CBK allows to compare several tools in order to find some that might be orchestrated. Researchers can easily become acquainted with a new tool, they discover in the CBK, and will be able to start immediately to work with it in the SDE. The connection between the SDE and the CBK is described in more detail in D1.2 [15].

This deliverable is structured as follows: In chapter 2 we capture the requirements for a service-oriented tool workbench and we have a look at related work. Afterwards, in chapter 3, we introduce the most important features of the SDE and describe how to use them. Chapter 4 presents tools that are already integrated in the SDE and in chapter 5 we conclude with the future steps in the development and use of the SDE platform.

2 Tool Workbenches

Developing service-oriented secure software and systems implies dealing with multiple programming languages, platforms and tools. The tasks carried out during development are ranging from modeling to implementation and from analysis to testing. The objective is to use a workbench, i.e., a platform allowing the integration of tools.

In this chapter, we discuss the requirements that such workbenches should fulfill in order to support the development of secure systems. Subsequently, we give an overview of related work (section 2.2). In the next chapter, we will present the integration platform selected and further developed within the scope of the NESSoS project: the Service Development Environment (SDE).

2.1 Requirements for a Service-Oriented Tool Workbench

The aim of the workbench is to provide a working environment for the development and analysis of secure software and systems. The power of the workbench will be given by the tools integrated in the workbench. This requires, among others, tool management and tool integration facilities. In particular, in a wide-ranging project as NESSoS several and heterogenous tools are produced and maintained by the partners, which need to be integrated into a common development platform.

In the following, we present a brief description of the most important requirements that were captured so far in lively discussions in the NESSOS meetings and with members of the ASCENS project [4]. ASCENS is another FP7 project that will use the SDE. It focuses on autonomic service-component ensembles [3]. Each requirement can be related to one of the following categories: user interface, architecture, tool management and tool integration.

User Interface. The workbench *should have a familiar user interface* so that the users need not much time for getting used to working with it. This can be achieved by extending existing workbenches for domain specific platforms, such as Eclipse. Furthermore, the workbench should *support the documentation of tools*, which means each tool and each method is described within the workbench. We already mention the diversity of tools the workbench should cover, e.g., tools for modeling, testing, analysis and transformations. For the sake of clarity, all tools should be classified in *categories*.

Architecture. The workbench shall be available under an *open source license* to allow reuse. Another focus is that it should be *easy to integrate new tools* – even existing ones that are written in arbitrary programming languages. Furthermore, the integration of commercial closed-source tools should be supported. It is important to *support a broad range of tools*, e.g., those that require interaction via a GUI or that are computational intensive and that might be partly outsourced to dedicated remote machines.

Tool Management. The *management of tools* should be supported, i.e., tools should be easy to be added and removed from the workbench. Additionally, an *automatic update mechanism* shall be provided for keeping the installed tools up-to-date.

Tool Integration. The framework should *provide assistance* for integrating new tools. On the one hand, a good documentation is always helpful and on the other hand some already integrated tools ease the process of integrating an own tool. The user should not only be able to execute single tools, but also to *orchestrate tools* using textual as well as graphical orchestration mechanisms. The resulting tool chains shall be handled equally to single tools so that the user can work on an abstraction layer without caring if a tool is made up of other tools or not. In addition, *transformations between tool outputs and inputs* should be available to convert certain formats, e.g., a text string from a file into an input string. Finally, *supporting inter-tool dependencies* is a requirement that is closely related to the update mechanism: tools can be related to each other, which means the installer has to take care of the versions of tools that are added or updated. This dependency solving makes sure that orchestrated tools can seamlessly work together.

The SDE fulfills all the aforementioned requirements and furthermore is highly extendible so that new features can be added as soon as needed, although up to now no further requirements emerged.

Table 2.1 presents an overview of the requirements and the categories identified so far. In particular, it shows how the SDE fulfills these requirements.

Category	Requirement	Feature
User Interface	Provide IDE with a familiar user interface	Eclipse-based
	Support good documentation of tools	List of installed tools and method descriptions
	Support tool categories	Tools displayed in groups
Architecture	Open source foundation	Code available under Common Public License (CPL)
	Enable easy integration of tools	Facades to wrap tool's functionality in an OSGi container
	Support broad range of tools	Generic platform for tool integration
Tool Management	Support management of tools	Add, remove and update tools
	Allow for automatic updates	Automatic actualizations
Tool Integration	Aid in development of new tools	Tutorials and existing tools
	Enable transformations between tool outputs and inputs	Support exchange of data between tools
	Support inter-tool dependencies	External dependencies
	Enable orchestration of tools	JavaScript/UML activity diagrams

Table 2.1: Requirements for a tool workbench and how SDE fulfills them.

The main objectives of the SDE – developed within the scope of the Sensoria project [50] – can be summarized as:

- to provide an Integrated Development Environment (IDE) for developing service-oriented software,
- to offer the ability to chain various tools together to perform complex operations, and
- to offer state-of-the-art research results in the form of tools to industrial users.

The SDE has been selected in the NESSoS project as the tool integration platform on which to build and focus on the integration of *security related* tools into the tool workbench. Fortunately, the SDE is generic enough to allow the integration of all tools as services, as such security related tools are part of this set and do not have to fulfill a special precondition. For further details on the SDE the reader is referred to Chapter 3.

2.2 Related Work

"Service-oriented architectures" and the term of "service" are the underlying principle of the tool workbenches fulfilling the requirements for the NESSoS project. The concept of orchestrated services is becoming increasingly popular, as can be seen from the shift to cloud computing services and the rising number of available cloud platforms [55]. Unfortunately, cloud services are not as flexible as services in a service development environment, because the input and output of services is not described generically. Instead, cloud services need to know the API of other services in detail in order to use them. Therefore it is not possible to orchestrate services from the outside, by only connecting outputs and inputs.

In the EU project PROFUNDIS, the tool integration platform PWeb [9] was developed, which provides a directory service for managing web services. This directory service makes it possible to publish or to search for a web service and it enables the orchestration of tools. Both the SeCSE Development Environment [49] and PWeb are entirely web-based, consequently it is not possible to add already existing offline tools (especially if they use a user interface), which are quite common in the field of secure software engineering.

Another tool approach is jETI [37], which continues the Electronic Tools Integration platform (ETI) using Java and web services. An Eclipse-based client is available, but the integrated tool's functionality is executed on a server and user interfaces of offline tools can only be used through a remote terminal.

Furthermore, process management systems have similar possibilities to connect services, as e.g., the commercial (partly open-source) tool ProcessMaker [46] and the open-source tool Activiti [1]. Instead of the activity diagram-based orchestration in the SDE, process management systems use BPMN [43] diagrams. BPMN 2.0 distinguishes – among others – between user tasks and script tasks (using a scripting engine) and Activiti additionally adds a Java service task and a webservice task. Java service tasks allow the developer to use Java classes (without local GUI) and to return so called "process variables". Webservice tasks can use web-based services that are described in WSDL [54]. Access to a web-based GUI is possible. These approaches are less flexible than the SDE regarding the integration of existing tools with local user interfaces.

Further service-based development environments, related to the SDE, are discussed in [38, p.298].

3 The Service Development Environment

The SDE is a tool workbench that provides an overview of available tools and their area of application and allows developers to use these tools in a homogeneous way and re-arranging tool functionality as required. The SDE is based on Eclipse - an open source platform - and is open source itself.

The SDE was developed for the integration of tools required in the development process of service-oriented services and systems, including modeling, analysis, code generation, and runtime functionality. Thus, the SDE (1) offers an extensible repository of a broad range of tools and describes their functionality and area of application in a user-friendly way, (2) allows developers to combine tool functionality, and (3) enables users to stay on a chosen level of abstraction, hiding the specific internal features, sometimes formal details, of the tool as much as possible.

Subsequently, we provide an overview of the SDE functionality in section 3.1, the features of the SDE are presented in section 3.2 and the use of the SDE is described in section 3.3.

3.1 Overview

The SDE was designed to integrate tools that support all development phases in an engineering process including phases such as modeling, analysis, code generation and runtime support. However, phases could be added or removed defining a different development process according to the specific needs of a domain, e.g., secure engineering. Despite the changes in the development process, the SDE tool workbench does not require any adjustments, but eventually it will be required to integrate domain specific tools.

The SDE provides the above mentioned support through a lightweight integration architecture [39]. The core features of this architecture are:

A SOA-based Platform: The SDE is based on a SOA, allowing easy integration of tools and querying the platform for available functionality. The tools hosted in the SDE are installed and handled as services.

A Composition Infrastructure: The SDE offers a composition infrastructure which allows developers to automate commonly used workflows as an orchestration of integrated tools.

Hidden Formal Methods: The SDE allows developers to use formal tools without requiring them to understand the underlying formal semantics.

Model-Driven Support: The SDE encourages the use of automated model transformations which translate between high-level models and formal specifications, and supports model-driven development of software and systems.

Regarding the requirements listed in the previous chapter, the SDE fulfills the requirements of all four categories, i.e., related to the user interface, architecture, tool management and tools integration (see Table 2.1).

User Interface. The SDE is based on the Eclipse development environment offering a standard, well-known and customizable user interface. Basing the tool integration platform on Eclipse increases the acceptance of the SDE within the project and regarding potential users that can download the SDE from the NESSoS website. The SDE provides a good documentation mechanism as installed tools are automatically listed. Functions can be annotated to explain the purpose and the semantics of methods and parameters. In addition, SDE supports the creation of tool categories.

Architecture. The SDE is an open source tool available under the Common Public License (CPL). The SDE enables an easy integration of tools (see chapter 4). For legacy tools, a wrapper in an OSGi container must be created. These wrapper tools have to include a XML file, which enables the integration of the tool and its functions in the development environment. The SDE is a generic and open tool workbench as it is not restricted to a specific kind of tools. It also supports the execution of code on remote computers.

Tool Management. The SDE supports adding and deleting of tools as well as listing of available tools and updating tools automatically.

Tool Integration. The SDE enables orchestration of tools via JavaScript or using a in SDE built in graphical editor. The latter provides an intuitive manner to construct tool chains. The features for allowing easy integration of transformations increases the acceptance of model-driven approaches and the use of verification techniques. The core of the SDE checks if external dependencies are satisfied. To compose functions of several tools, the output of one function has to have the same format as the input of the subsequently connected function.

Finally, the use of the well-known programming language Java encourages the use of the SDE.

3.2 Features of the SDE

The user can access the SDE functionalities by a user interface, as shown in Figure 3.1:

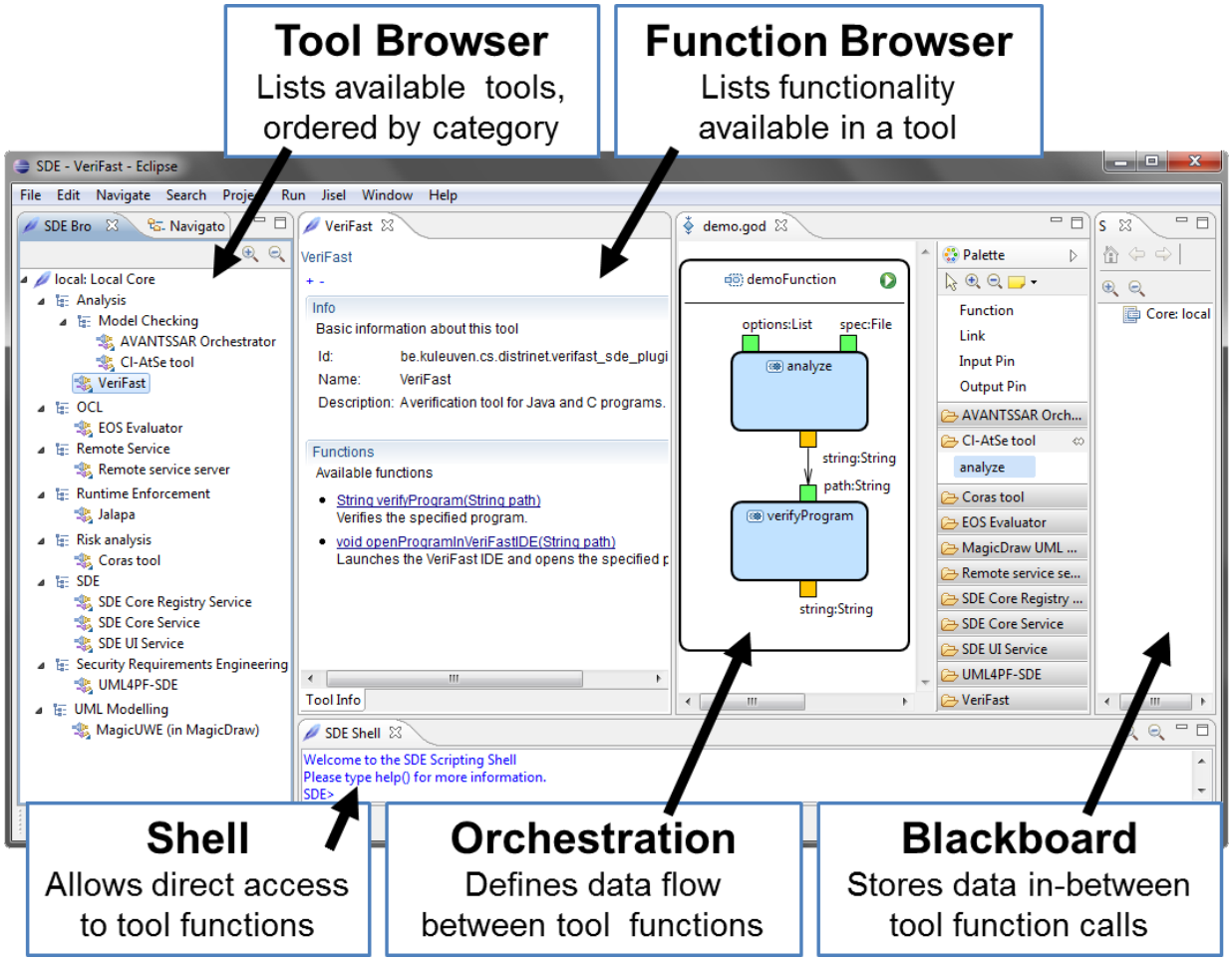

Figure 3.1: User interface of the SDE

- On the left-hand side, the **SDE Browser** is displayed. It contains a categorized listing of all tools which are currently available in this particular instance of the SDE.
- After double-clicking at one tool in the SDE Browser, the description of the tool as well as of its functions are displayed in the **Function Browser**.

- On the right-hand side, the **SDE Blackboard** is displayed. The blackboard is used to store Java object values in-between service invocations when executing a tool's functions manually from within the SDE Browser.
- At the bottom, the **SDE Shell** is displayed. It is a manual orchestrator which can be used to employ JavaScript to call tool functions.
- The **SDE Orchestrator** allows the user to create tool chains graphically by linking functions of (different) tools in order to create a new service that manages the comprised tools, functions, inputs and outputs.

Technically, the Eclipse platform [27] and its underlying, service-oriented OSGi [44] framework is used. OSGI is based on so-called bundles, which are components grouping a set of Java classes and metadata providing among other things name, description and version. An OSGi bundle may provide arbitrary services to the platform and therefore all tools are integrated as bundles which offer certain functions for invocation by the SDE platform.

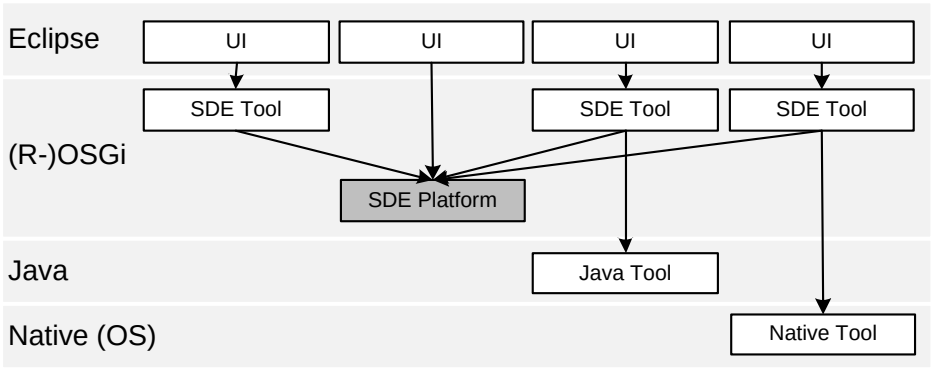

Figure 3.2: Architecture of the SDE (adapted from [39]).

This means the User Interface (UI) that the SDE provides within Eclipse comprises the (graphical) access not only to the SDE workbench, but also to all tools, as depicted in Figure 3.2. Existing Java tools and even native tools can be seamlessly integrated in the SDE by writing a wrapper, which defines how functions of each tool can be used.

As outlined above, the SDE provides an environment for using and orchestrating tools. The tools themselves are provided by the developers and in general must be installed separately; however, some are already distributed with the SDE as examples. The basic functionality of the core is threefold:

- It provides access to all the registered tools by an API (which can be seen as a basic discovery service) and by a UI. The API allows retrieving tools based on their ID or name and is intended to be used from within Java, while the UI allows graphical browsing of registered tools directly for the end user.
- It provides access to the tool functions by API and by UI as well. The API allows calling arbitrary functions on the registered tools from within Java, while the UI provides a generic interface for executing these functions, storing the results, and re-using results as input for other functions.
- It provides an orchestration mechanism using activity diagrams and JavaScript. Using such orchestrations, the API discussed in the previous two points can be accessed and tools-chains can be built in a simple way.

Besides simply using their functionality, tools can also be orchestrated with partial help from the platform. This can be achieved by different means as well. In particular, the usual scenarios are orchestration using the built-in shell or JavaScript-based tools or the graphical orchestration inside SDE or an external mechanism that is, e.g., written in Java.

Tools to be used as part of the SDE must be implemented as OSGi bundles and contain a declarative description of the entry points of their functionality but are otherwise unlimited in their implementation. In particular,

- Tools may be written in Java and may consist of an arbitrary number of libraries, other Eclipse plug-ins, or external code.
- Tools may also wrap native code, thereby providing an interface to non-Java software.
- Tools may include functionality for calling remote services, thereby providing the link to Web services.

3.3 Using the SDE

After having elaborated on the features of the SDE, this chapter provides a short tutorial about how to install the SDE (subsection 3.3.1), how to integrate existing tools into the SDE (subsection 3.3.2) and how to use its orchestration functionality as described in subsection 3.3.3. A more detailed tutorial can be found at the SDE website [48].

3.3.1 Installing the SDE

At the beginning, the Eclipse Modeling Tools Version (version 4.3 or newer) [27] has to be downloaded and unzipped. Once Eclipse has been installed, the SDE core and many tools can be installed via update sites, using the menu `Help / Install New Software...`. The link to the SDE Eclipse Update Site is `http://svn.pst.ifi.lmu.de/update/sde`.

This update site contains two features that should be installed: The core itself and the development feature which eases development of new SDE tools.

The SDE website [47] contains a step-by-step tutorial, a bug tracking system, videos demonstrating the SDE in action. Integration of tools related or developed within the scope of the NESSoS project are currently under construction and can be accessed through the CBK [28] tool descriptions (the necessary URL can be found in the section “Eclipse Update Site for SDE” [20]). To install such an integrated tool, the link has to be copied from the CBK in the `Install New Software...` dialog of Eclipse. If both the SDE and a tool is installed, the tool and its functions are listed in the SDE browser that can be accessed in the SDE perspective: `Window / Open Perspective.../ Other.../ SDE`.

3.3.2 Integrating a New Tool into the SDE

Creating new tools or adapting existing tools in the SDE requires the following three steps to be executed by a developer:

1. Creating or reusing an OSGi bundle.
2. Creating a tool class/interface and functions within this bundle, which implement the tool functionality.
3. Registering the tool with the SDE core.

Depending on the type of tool to be implemented, an OSGi bundle may already be available or needs to be created from scratch. If the tool to be integrated is an Eclipse plug-in (starting from version 3.0), it is already implemented as an OSGi bundle. If the tool is written in Java, the code might be used from within an OSGi bundle or Eclipse plug-in as well, either in source or as a .jar library. If the tool is written in a native language or provided as a Web service, the recommended way is to create a wrapping OSGi bundle which forwards all functionality.

For a simple example we assume that we want to integrate an existing tool into the SDE, which is the most common case in the NESSoS project. This means that a wrapper for that tool has to be created.

1. First, the user shall open a wizard in Eclipse from the `File / New / Project.../ Plug-In project` menu and enter the name of the SDE tool.

2. Afterwards, the plug-in contains a MANIFEST.MF file in the META-INF folder which contains the OSGi settings for this plug-in. Adding the line `eu.sensoria_ist.casetool.core,` to the section `Require-Bundle` adds the SDE core as a dependency.

3. The tool bundle and the implementing class must be registered with the SDE so that the tool can be listed in the SDE browser and is able to become a part of tool chains. Publishing a tool is done by creating an Equinox extension similar to registering other extensions in Eclipse, i.e., by registering functionality at an extension point. The user can write such an extension either by hand in XML inside the `plugin.xml` file, or add it via the Plug-in Development Environment (PDE) wizard (as depicted in Figure 3.3), which in turn writes the XML. Additionally, with the SDE package installed, Java source code and annotations can be used within an interface that comprises the functions of the tool. These annotations can be transformed to the XML file using a SDE context menu.

4. Of course the classes with their methods that are referenced as entry points in the `plugin.xml` file have to be implemented in a way that the functions wrap the original tool, i.e., the parameters are passed from the SDE to functions of the integrated tool and that communicates with the underlying tool and hands back results.

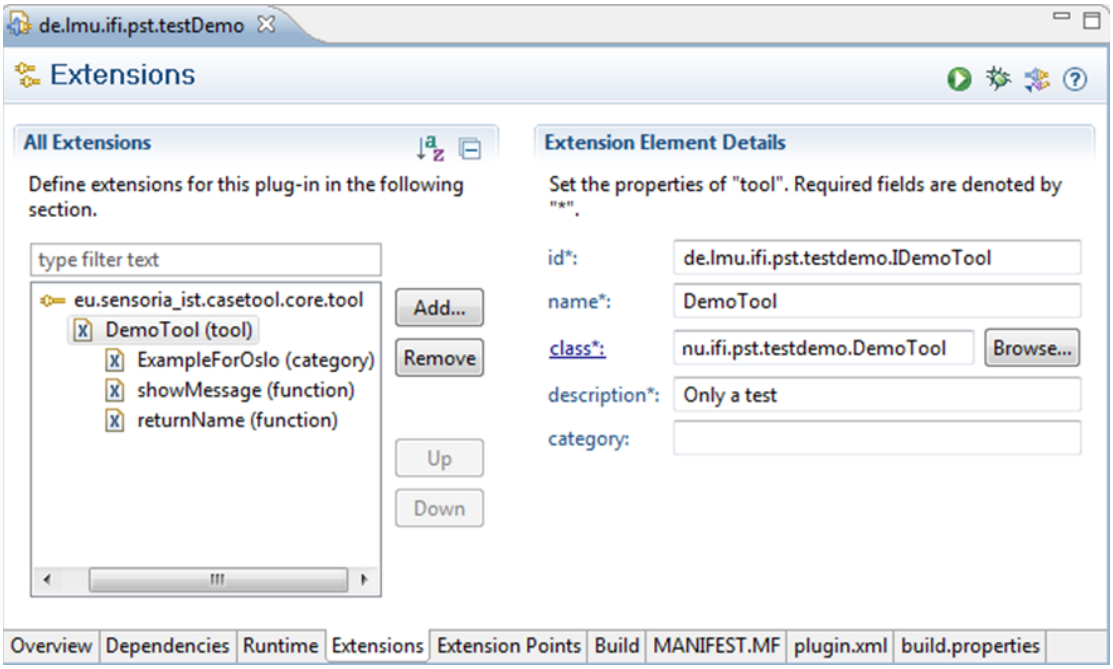

Figure 3.3: Adding functions to a SDE tool (PDE wizard)

Finishing tool integration implies that the SDE core has to be added as a dependency to the tool bundle and that the `plugin.xml` file has to be created that describes the integrated functions of the tool. The necessary packages and libraries have to be exported, e.g., as Eclipse Update Site, so users can install the new tool using the Eclipse update mechanism. In the rare case that commercial software is required, it of course has to be purchased and installed separately.

3.3.3 Creating Tool Chains

The SDE provides two different tool orchestration facilities for building tool chains: a graphical orchestration with data-driven activity diagrams and a JavaScript orchestrator. For instance, the use of the graphical orchestrator is explained in the following.

The user starts the construction of the tool chain, i.e., orchestrating tools by selecting `File / New / Other / SDE / SDE Graphical Orchestration`. Two files are created:

- The first file contains the diagram, and has the extension .god (for graphical orchestration diagram)
- The second file contains the orchestration itself, and has the extension .go (for graphical orchestration)

It is recommended to pick the same name for both files (excluding extension).

The canvas of the editor corresponds to a new tool to be created in the SDE. As each tool can contain multiple functions, a function needs to be added first, using the palette on the right-hand side, afterwards clicking onto the canvas to create a new function. It has to be named appropriately, before tool functions can be dragged into the new function from the palette on the right-hand side, which contains all invocable functions of all installed tools. Additionally, the following meta tools may be used:

- Link to model data flow from an output of a function to the input of another
- Input Pin to add an input parameter to a function
- Output Pin to add an output parameter to a function

Before the new function can be executed, the tool needs a name. For this, the user has to right-click on the canvas and select properties. In the properties view, both the name and id have to be filled in. Finally, a simple click on the green play button in the upper right corner of a function executes it. Furthermore, the SDE menu may be used to convert the resulting tool to a SDE tool.

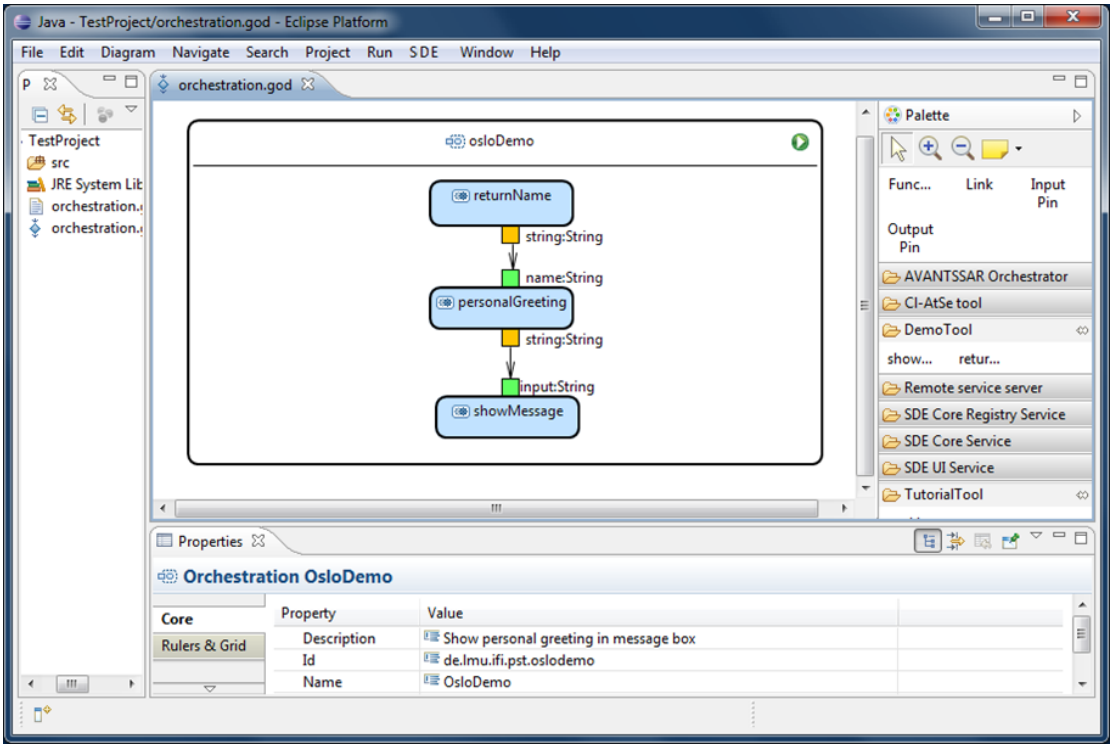

Figure 3.4: SDE's graphical orchestrator

A simple demo, which was built for the NESSoS meeting in Oslo, is depicted in Figure 3.4. `returnName` and `showMessage` are originally functions of one tool and `personalGreeting` is a function of another tool, but the user of the resulting `osloDemo` tool, does not have to care about the encapsulated tools.

4 Tool Integrations

In this Chapter we present a brief description of the tools which were integrated into the SDE in the first year of the NESSoS project (see Figure 4.1). The descriptions focus on the purpose and functionality of the tools and their integration into the SDE, including the technical requirements and some advices for their usage. For detailed usage examples the reader is referred to the CBK [20]. The technical requirements are shown in tables that are similar to those available in the CBK.

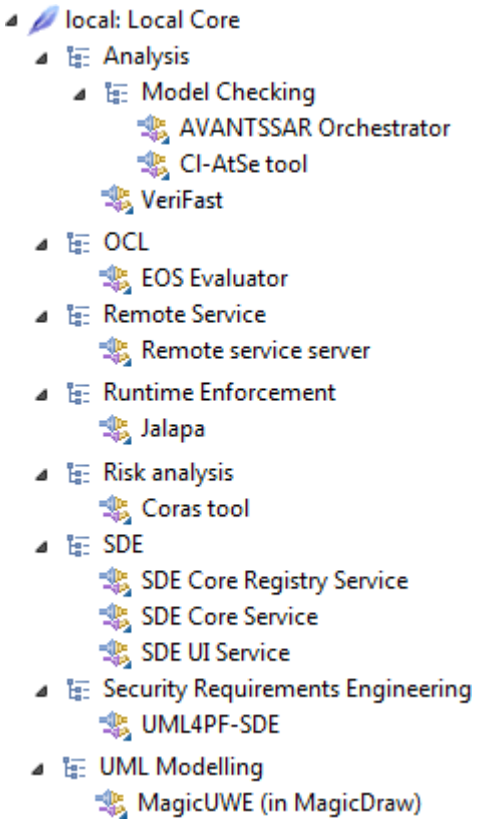

Figure 4.1: SDE browser

4.1 Avantssar-atse (CL-ATSE)

CL-AtSe [52, 2, 6] is a Constraint Logic based Attack Searcher for security protocols and services. The main idea in CL-AtSe consists in running the protocol or set of services in all possible ways by representing families of traces with positive or negative constraints on the intruder knowledge, on variable values, on sets, etc. Thus, each run of a service step consists in adding new constraints on the current intruder and environment state, reducing these constraints down to a normalized form for which satisfiability is easily decidable, and decide whether some security property has been violated up to this point. CL-AtSe does not limit the service in any way except for bounding the maximal number of times a service can be iterated, in the case such an iteration is allowed in the specification. Otherwise, the analysis might be non-terminating on secure services and only heuristics, approximations, or restrictions on the input language could lift this limitation.

The online version of the tool can be accessed from http://cassis.loria.fr/.

4.1.1 Features

Input CL-AtSe reads any specification written in ASLan language [7] by default. ASLan is a formal language for specifying trust and security properties of services, their associated policies, and their composition into service architectures. A service's behavior (or protocol role) in ASLan is defined by three elements: a set of transitions that can change facts from the current state to another, including messages to be exchanged (with applied security policies) as first order terms; some initial state — a set of facts; and a finite set of Horn clauses typically used to define an authorization logic.

The security goals to verify can be expressed in two ways: as an attack state and Linear Temporal Logic (LTL) goal. The former is represented as a pattern of a state that is considered insecure (if the tool is able to attain such global state that conforms with this pattern, an attack is reported). The latter is more general and expressive: if the tool is able to satisfy the given LTL formula it will report an attack. However, the tool has currently a limited support of LTL goals.

Output If a security property of the input specification is violated then CL-AtSe outputs a warning (UNSAFE), some details about the analysis (e.g. whether the considered model is a typed or an untyped one), the property that was violated (secrecy, for instance), statistics on the number of explored states, and, finally, an ATTACK TRACE that gives a detailed account of the attack scenario. If no attack was found then similar information is provided (except UNSAFE and the ATTACK TRACE).

Simplifications and optimizations CL-AtSe implements modules to simplify and optimize the input specification statically before analysis. The simplification module aims at reducing the search space without truly changing the structure of the problem to be analyzed. This option is activated by default

because the computations involved are quite small compared to the expected gain on the analysis time. The optimization module, on the other hand, tries to change the structure of the input specification to preprocess a significant part of the branching that should be done during analysis, so that many dead-ends can be found and removed statically. This option is not activated by default since, theoretically, in the worst case it might be equivalent to a complete analysis, especially for specifications with weak message structure (e.g. a large number of encryptions with no headers, etc.). However, when the specification is adapted the gain during analysis can be huge. It can be activated with the –opt option.

Suspending the analysis CL-AtSe is capable to stop and restart later an analysis, by writing and reading a so-called state file containing the current state of the analysis. This can be produced either anytime during a very long analysis to resume, e.g., in case of power loss, and this can be used each time an attack is found to store the state of the system without the attack, and thus, resume the analysis to find other attacks (either on the same branch with other constraints, or on other branches). The latter is used by the AVANTSSAR Orchestrator (See section 4.2, [8]) to enumerate orchestrations. CL-AtSe takes care not to produce new attacks which would admit an attack already found as a prefix. Finally, the stop-and-restart functionality is used for massive parallel computing: with the appropriate option, CL-AtSe can advance a bit in a protocol's analysis before stopping and writing one state file per branch that still need to be analyzed. This can produce as many sub-state files as needed, and each can be analyzed independently from others by CL-AtSe, e.g., on different CPUs or PCs.

4.1.2 Integration into the SDE

The technical details of the integration in the SDE are described in the following, an example is shown in Figure 4.2:

Technical Requirements	The plugin, which is a wrapper of the CL-AtSe tool, includes an embedded version for Linux (x86) and Windows for machines that do not have this tool. It can also use an existing executable via the "CLAtSeExecutable" option of the SDE interface (IClAtSeTool).
License	unspecified
Eclipse Update Site	`http://cassis.loria.fr/eclipse/`
Installation Guide	The preferred approach to install the plugin is by using the Eclipse's Update Manager.
Tool input type	A file (java.io.File) corresponding to the protocol specification expressed in the ASLan language.
Tool output type	The return type is of type java.lang.String, which contains the result of the analysis, as mentioned at page 27.

The analyze function, which analyzes the specified protocol in order to determine if a security property is violated or not, has the signature `String analyze(File spec, List<String> options)`. The specified parameter makes reference to the ASLan specification file. The list of options is used to pass the command line options to the native CL-AtSe tool. Afterwards, a string contains the result of the analysis done by CL-AtSe.

4.2 Avantssar Orchestrator

The AVANTSSAR Orchestrator [8] is a tool for automatic orchestration of Web Services taking into account their security policies. In short, it generates a service called mediator that is able to satisfy requests of a given client with the help of given community of available services. Therefore, what we call "Orchestrator" is a service (or tool) that reads the ASLan models of the client plus other services, and produces a new service called "mediator". The mediator can query services, deduce new knowledge, and create new messages to adapt the client and services interfaces. And we call "orchestration" the safe interaction between the "mediator" and the client plus other services that fulfill all goals and security constraints.

The orchestration approach relies on the analogy with a state reachability problem in cryptographic protocols analysis domain. Exploiting this idea, the Orchestrator is based on CL-AtSe (see section 4.1), a tool for solving cryptographic protocols insecurity in a presence of an active Dolev-Yao intruder.

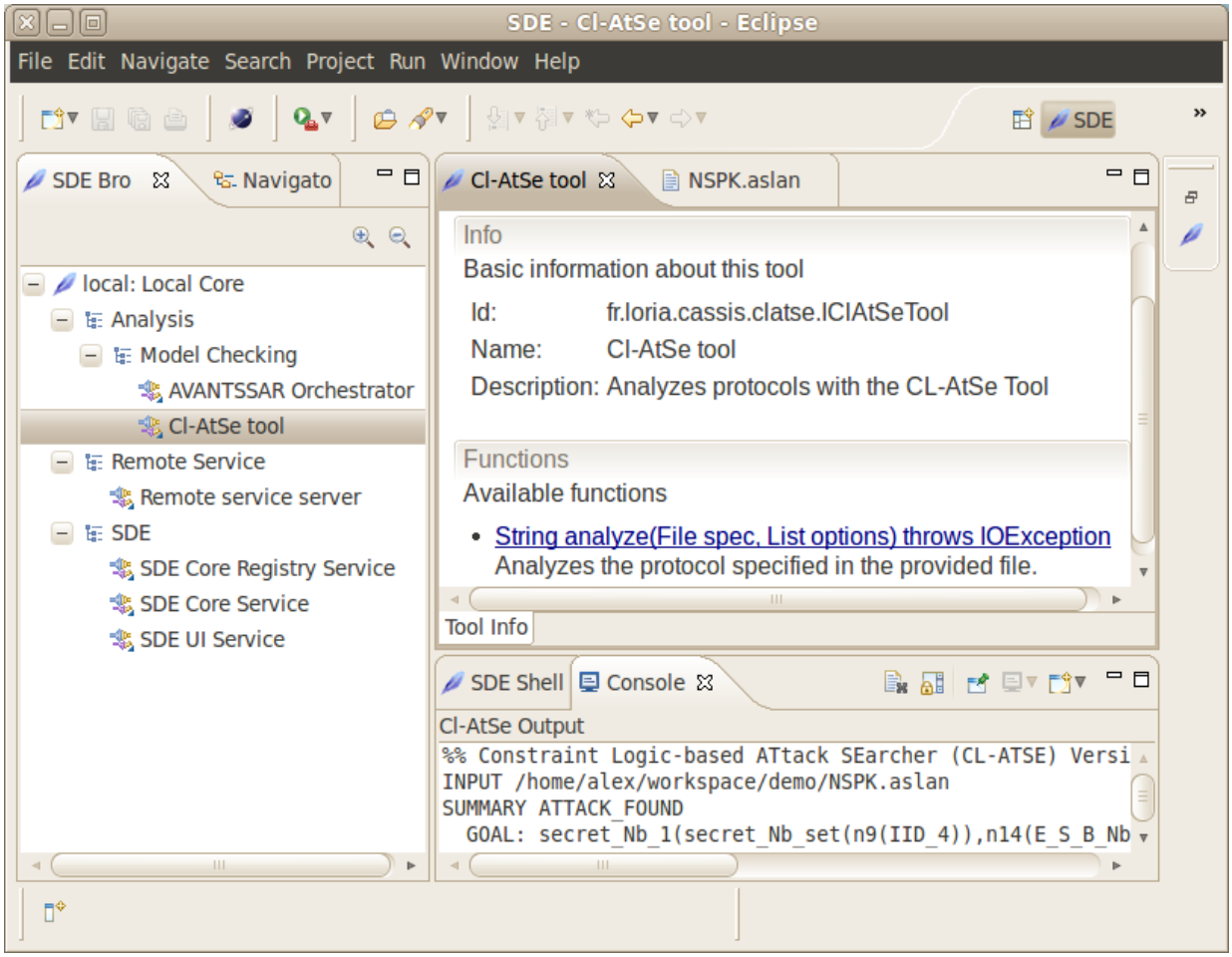

Figure 4.2: CL-AtSe in the SDE

The online version of the tool can be accessed from http://cassis.loria.fr/.

4.2.1 Features

Input The input problem is described in the ASLan language [7], extended with some keywords in order to distinguish agents like, e.g., the client, some service, etc. Given a set of available services and a client, the tool automatically generates a mediator, a composed service that is able to satisfy the given client's requests. By invoking available services with requests that satisfy their security policies which are built from collected so far messages, it enriches its knowledge with new data from the services' replies and transforms it if needed using Dolev-Yao operations in order to reply to a client's request. The available services and a client are specified in a form of transition systems in the ASLan language (see section 4.1). The mediator is defined only by its initial knowledge. Another type of input is admitted: instead of specifying a client, one may partially define a mediator by providing only the part related to the communication with the putative client.

The tool also accepts session-id, an identifier of a job, to continue solving the previously defined problem. Besides the orchestration problem, a modeler, using ASLan syntax, can add some global security properties that can be validated after solving the orchestration problem. These properties are checked by creating automatically a new specification model from the orchestrator's output. This model describes the client, the generated mediator, the services, and the security properties. Then, along all possible orchestration's outputs, one is chosen for which the analysis with CL-AtSe of this model finds no attack.

Output In the case where the specification of the client is given, an ASLan specification of the mediator that satisfies the client's requests is produced. In the case, where the partial specification of the mediator is given, a specification of the putative client is generated. Moreover, a new mediator service is issued, which extends the mediator given in the input with the necessary interactions with the available services.

The tool also outputs the session-id as required above to build other solutions of the same problem. The global security properties are preserved, thus, the output specification is ready to be analyzed for detecting vulnerabilities.

Generating different solutions Since the generated mediator can be vulnerable with regard to specified global security properties in the presence of active Dolev-Yao intruder, it will probably be rejected by the modeler. Then, the user possibly wants to generate another mediator and check it again for the vulnerabilities using, e.g., CL-AtSe.

In order to automatize the process the Orchestrator provides a facility of generating *next* possible mediator. To this end, after defining the problem input, the tool outputs some session-id which can be used later as a tool input in order to restart the searching for the new mediator from the point it found the last one. This feature is based on the ability of CL-AtSe (which is used as a module of the Orchestrator) to suspend and continue its run over the search tree representing all possible behaviors of the services in the given specification.

4.2.2 Integration into the SDE

The technical details of the integration in the SDE are described in the following:

Technical Requirements	The plugin is a wrapper of the AVANTSSAR Orchestrator tool. It requires at least the Eclipse version 3.5 and the SDE core plugins.
License	Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License (CC BY-NC-ND 3.0)
Eclipse Update Site	`http://cassis.loria.fr/eclipse/`
Installation Guide	The preferred approach to install the plugin is by using the Eclipse's Update Manager. You can see the following screencast to guide you during this process `http://cassis.loria.fr/wiki/Wiki.jsp?page=Nessos`
Tool input types	The available services and a client are specified in a form of transition systems in the ASLan language.
Tool output type	As an output, an ASLan specification of the mediator that satisfies client's requests is produced.

The orchestrate function, which automatically generates a mediator, a composed service that is able to satisfy the given client's requests, has the function signature `String orchestrate(File inputSpec)`. Input parameters are on the one hand the file containing the specification of the available services and on the other hand the client expressed in the ASLan language. The function returns a string containing a mediator which satisfies the client's requests if an orchestration is found, or an error indicating that no orchestration is found for the specified services.

4.3 CORAS Tool

CORAS [36, 22] is a model-driven approach to risk analysis that consists of three tightly integrated building blocks, namely the CORAS method, the CORAS language and the CORAS tool.

When conducting a risk analysis using CORAS, diagrams made with the CORAS risk modelling language are used intensively during the structured brainstorming sessions of the risk analysis workshops for the purpose of risk identification, risk estimation, risk evaluation and risk treatment. Because models are made on-the-fly in these sessions, appropriate tool support is decisive for the success of an analysis; to avoid interrupting the flow of discussion, it is important that the models are made efficiently and are clearly presented.

4.3.1 Features

The CORAS tool is a diagram editor supporting the modelling of risks and threats in the CORAS language. It is designed to support on-the-fly modelling and provides full support of the CORAS language. In particular, the model-driven approach uses models extensively during all phases of the risk analysis process, both to gather and document the results and as basis for specific risk analysis tasks. For this purpose, CORAS comes with several kinds of diagrams, such as threat diagrams for risk identification, risk diagrams for risk evaluation, and treatment diagrams for the identification of options for risk mitigation.

In addition to five basic kinds of CORAS diagrams that support the risk analysis process, the tool comes with support for making three other kinds of CORAS diagrams that are developed to provide specialized support: High-level CORAS supports abstraction and easily comprehensible overviews of large risk models; dependent CORAS supports documentation of assumptions and dependencies, as well as modular reasoning; legal CORAS supports documentation of legal aspects and their impact.

The CORAS tool facilitates quick modelling and editing of syntactically correct diagrams by drag-and-drop functionality and automatic prevention or automatic alert of syntactically illegal expressions. Because the CORAS tool is a diagram editor for making CORAS diagrams it does not have input and output, other than the CORAS project files that are created.

4.3.2 Integration into the SDE

The technical details of the integration in the SDE are described in the following:

Technical Requirements	Eclipse 3.5 (Galileo) EMF 2.5 GMF 2.2 Epsilon Core 0.8.9 Epsilon EMF/GMF Live Validation 0.8.9
License	Eclipse Public License 1.0 [30]
Eclipse Update Site	`http://www.nessos-project.eu/svn/WP2/integratedTools/CORAS-tool/`
Installation Guide	The preferred approach to install the plugin is by using the Eclipse's Update Manager
Tool input type	CORAS project file (*.coras_project)
Tool output type	CORAS project file (*.coras_project)

There are two available functions offered by the CORAS SDE plugin, one for creating a new CORAS project and one for opening an existing CORAS project. The former, `createProject()`, has no input parameters and no return value. After the call, the user must specify the file name. The latter, `openProject(String filePath)`, takes as input the file path for a CORAS project file and has no return value.

4.4 EOS (Eye OCL Software)

EOS (Eye OCL Software) [21, 29] is a Java component for performing efficient evaluation of OCL [41] expressions on medium-large size scenarios. Since OCL is the standard specification language chosen by model-driven security modeling languages like SecureUML for formalizing *authorization constraints* (i.e., constraints on the permissions granting access to protected resources), the EOS component will play a major role within tools supporting model-driven security software development, e.g., to check the aforementioned authorization constraints at run-time.

4.4.1 Features

The EOS component includes an OCL parser (which uses SableCC) and an OCL evaluator, the latter consisting of about 7K lines of Java code. The current version handles most of OCL, including the possibility of adding user defined operations.

With the idea of making it as “pluggable” as possible, the EOS component is not based on any particular (meta)modeling framework: its public interface provides methods to insert elements, one-by-one, into user-models and scenarios, and to input the expressions to be evaluated as strings of ASCII characters. This decision allowed us also to design the EOS’s data structure for internally storing user-models and scenarios in such a way that object properties are efficiently accessed. In addition, before evaluating a collect expression, we try to (over)estimate the size of the resulting collection and allocate memory in advance. The rest of the EOS implementation is rather straightforward: OCL iterator-expressions are executed using Java for/while loops and standard OCL operations, when possible, are executed using the appropriate Java operators. As expected, expressions are evaluated in EOS following an eager strategy: in particular, collection-expressions are fully evaluated and their resulting elements are all allocated in memory.

Some of the most relevant usages of EOS are:

- To parse OCL expressions in the context of a given user-model (typically a class diagram) returning a well-typed AST.
- To check OCL invariants, which may include user defined functions, over a given scenario (typically an object diagram)
- To evaluate OCL queries, which may include user defined functions, over a given scenario.

4.4.2 Integration into the SDE

The technical details of the integration in the SDE are described in the following:

Technical Requirements	Java 1.6 or higher.
License	unspecified
Eclipse Update Site	http://www.bm1software.com/updates/sde/
Installation Guide	The preferred approach to install the plugin is by using the Eclipse’s Update Manager.
Tool input type	EOS is a Java component: its public interface provides methods to insert elements, one-by-one, into user-models and scenarios, and to input the expressions to be evaluated as strings of ASCII characters.
Tool output type	The plugin’s return type is of type java.lang.String, which contains the value resulting from the evaluation.

More than twenty methods of EOS can be used from within the SDE. Details about the integrated methods can be found at the following Javadoc website: http://www.bm1software.com/eos/doc/.

4.5 Jalapa

Jalapa [10, 11] is an extension to the security model of Java that allows for specifying, analysing and enforcing history-based usage policies. Programmers can sandbox an untrusted piece of code with a policy, enforced at run-time through its local scope.

4.5.1 Features

Jalapa provides developer with the facilities for designing and applying their local security policies. Local policies [12] generalise both history-based global policies and local checks spread over program code. They exploit a scoping mechanism to allow the programmer to “sandbox” arbitrary program fragments. Local policies smoothly allow for safe composition of programs or services with their own security requirements, and they can drive call-by-contract composition of services [13]. In mobile code scenarios, local policies can be exploited, e.g. to model the interplay among clients, untrusted bundles and policy providers: before running an untrusted bundle, the client asks the trusted provider for a suitable policy, which will be locally enforced by the client throughout the bundle execution.

Local policies are defined through usage automata, a variant of finite state automata (FSA), where the input alphabet comprises the security-relevant events, parametrized over objects. So, policies can express any regular property on execution histories. Our policies are local, in the sense that programmers can define their scope through a “sandbox”construct.

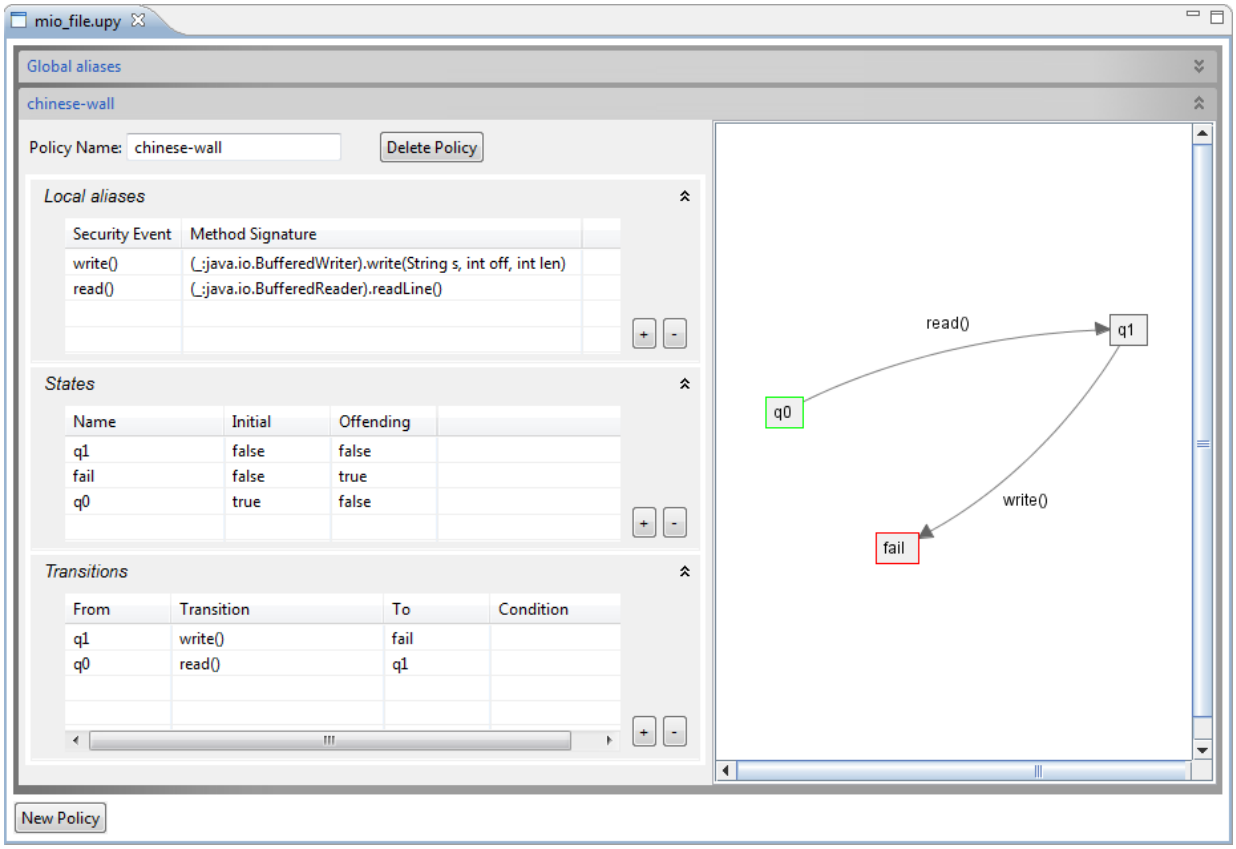

Figure 4.3: Jalapa policy editor

4.5.2 Integration into the SDE

Technical Requirements	Eclipse 3.5 (Galileo) [25]
License	Eclipse Public License 1.0 [30]
Eclipse Update Site	http://www.nessos-project.eu/svn/WP2/integratedTools/Jalapa/
Installation Guide	Detailed documentation and tutorial is available at http://jalapa.sourceforge.net/.
Tool input type	Usage Policy file (*.upy)
Tool output type	Standard Java binaries (*.class or *.jar)

The Jalapa tool consists of few components which are listed below.

- Policy Editor. A graphic component for editing security policies (see Figure 4.3).
- LocUsT Model Checker [14]. A model checker for the verification of history expressions and local policies.
- Jisel Instrumentator [11]. A bytecode instrumentator for enriching applications with security controls on local policies.

Figure 4.4 shows the main functionalities of the Jalapa tool integrated in the SDE view. The functionalities are:

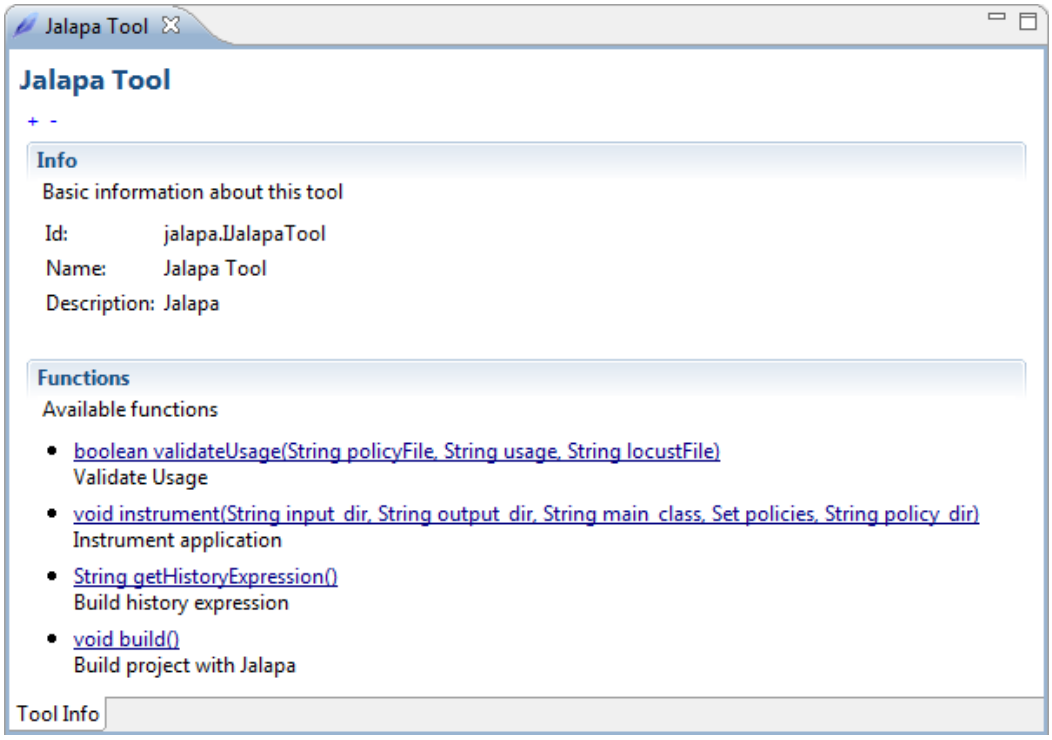

Figure 4.4: Jalapa as a SDE component

validateUsage invokes LocUsT for validating an history expression against a set of policies.

instrument inserts security checks in the project binaries with Jisel.

getHistory returns the plain text representation of the history expression of a program.

build invokes the previous functions and generates a secured build of the project.

4.6 MagicUWE

The CASE tool MagicUWE [35] was created to support the development of Web applications. It focuses on the modelling phase and uses the UML-based Web Engineering (UWE) methodology.

4.6.1 Features

UWE provides a UML [42] extension (a so called UML profile) based on stereotypes, tagged values and OCL constraints. The advantage of UWE is to stick to the standard UML, which allows using UML CASE tools such as MagicDraw. We decided to rely on the MagicDraw CASE tool [40] (v.16.8) for modeling all kinds of Web applications [18]. Nevertheless, our tool concept may be adopted for other commercial and open source UML CASE tools. The aim is to augment usability providing additional support in the use of the Web specific elements in the design, automatizing certain steps and providing shortcuts.

Within the NESSoS project, the main focus regarding UWE is on the security extension (formerly called UWEsecurity [16]) that is fully integrated in UWE [17]. With this security extension of UWE, Web developers are enabled to model security aspects as authentication, access control and secure connections efficiently. UWE aims at a sufficient abstraction which is needed at the interface between Web Engineering and Security Engineering.

Whenever UWE models are created, some tasks have to be repeated over and over, such for example how the navigation menu structure is built. Furthermore, some consistency checks and transformations

are very time consuming if executed manually. The plugin MagicUWE provides – among others – features like inserting UWE's stereotyped elements and copying stereotypes and their tags from the toolbar. Furthermore, MagicUWE supports Rich Internet Application (RIA) patterns and transformations between UWE models. MagicUWE also allows the modeler to:

- copy UWE stereotypes and their tags between a state machine and its substate machines
- specify tags facilitated by a context menu
- derive the type of a substate from the stereotypes of a superstate recursively
- inherit stereotypes for use cases stored in a package
- check features of the models

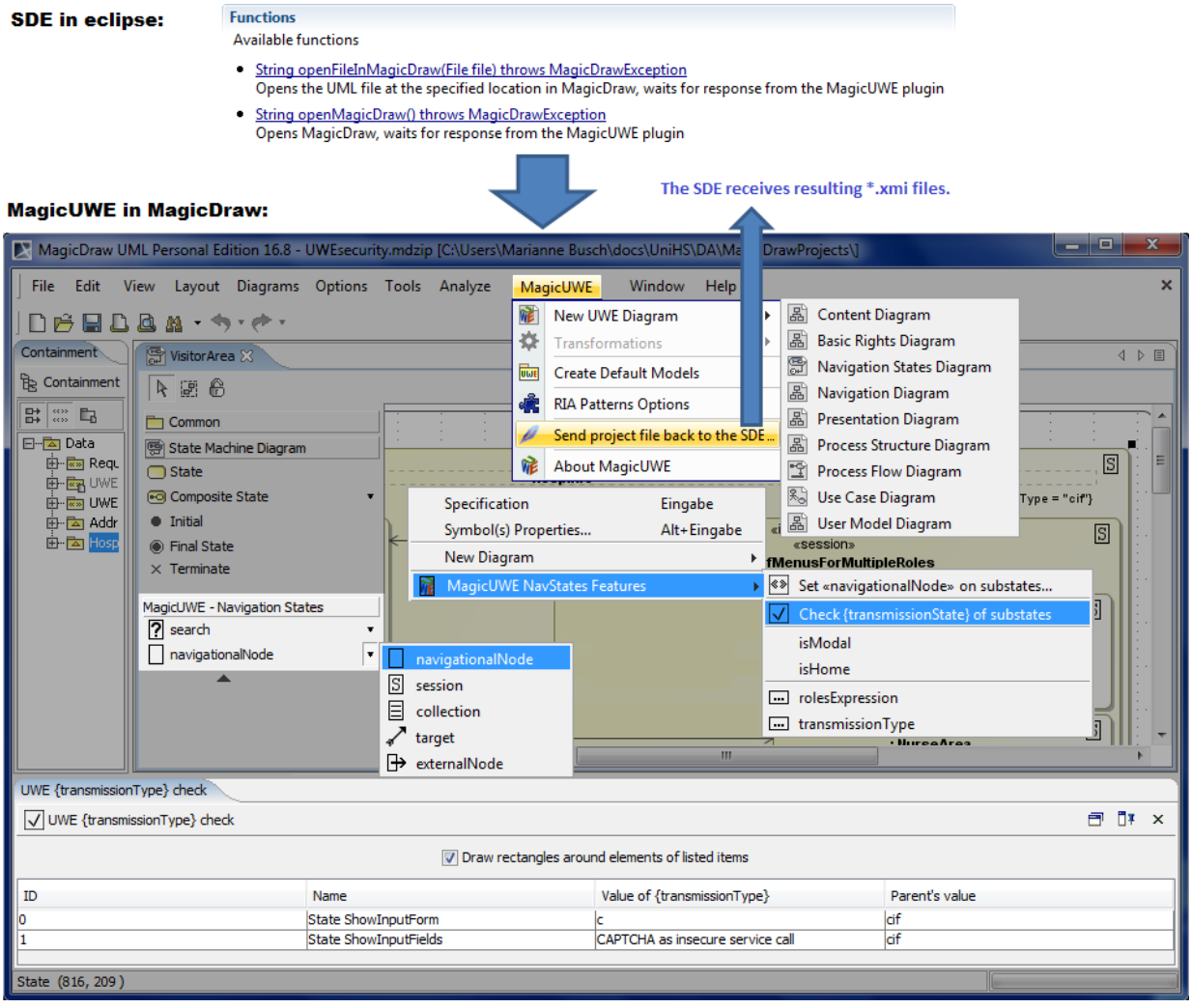

Figure 4.5: MagicUWE in MagicDraw, launched from the SDE

Figure 4.5 depicts some of these plugin features. An example for such a functionality is the check whether the transmission type of a secure connection is changed within nested states of the application's navigation model. This allows the modeler to see, if a service as, e.g., a CAPTCHA that is used within a secure area communicates over an unencrypted connection. Without MagicUWE, all substates would have to be checked for changes by hand, which can be very time-consuming for larger models.

4.6.2 Integration into the SDE

MagicUWE is a typical example for a SDE plugin that uses an external GUI. Therefore MagicDraw, which includes MagicUWE, has to be executed locally. The technical details of the integration in the SDE are described in the following:

Technical Requirements	MagicDraw v.16.8 (at least Personal) and the MagicDraw plugin MagicUWE have to be installed.
License	Common Public License Version 1.0 [24] [1]
Eclipse Update Site	http://www.nessos-project.eu/svn/WP2/integratedTools/MagicUWE/
Installation Guide	MagicUWE is available at http://uwe.pst.ifi.lmu.de/toolMagicUWE.html within an installer: `java -jar MagicUWEvXXXInstaller.jar` (or double-click on the file in many operating systems) There is also a zip version for MagicDraw Resource Manager. If you prefer to use the MagicDraw Resource Manager, you can import the zipfile. (The folder `MagicDraw/plugins/MagicUWE` has to be deleted before *re*installing MagicUWE.)
Tool input type	MagicDraw 16.8 project file, e.g. a *.mdzip file
Tool output type	*.xmi files (exported within MagicDraw using emfuml2xmi v3)

Due to the fact that MagicUWE is a plugin for MagicDraw, both tools must be installed, before using the MagicUWE wrapper plugin within the SDE. MagicDraw has to be executed in order to work with MagicUWE. After the execution within the SDE, the MagicUWE wrapper waits for the resulting project files, using client/server communication. Of course the user can skip this process at any time. In order to be compatible to other tools, the resulting project file is exported in the XMI format.

Within the MagicUWE wrapper two methods are available:

- `void openFileInMagicDraw(File file):String throws MagicDrawException` and
- `void openMagicDraw():String throws MagicDrawException`.

Both execute MagicDraw, and wait for a result path, the first one additionally allows the developer to specify a project file. The option `MAGICDRAW_EXECUTABLE_LOCATION` has to be specified within the SDE and should point to the absolute path to the MagicDraw executable in local file system, whereas the option `PORT_FOR_MAGIC_UWE` stores the port where MagicUWE will connect the SDE server in order to send the exported project file back.

4.7 UML4PF

UML4PF [53, 23, 31] is a tool to support requirements analysis and architectural design based on Michael Jackson's problem frame approach [32].

4.7.1 Features

Using UML4PF, a requirements engineer can develop different kinds of diagrams for different purposes, e.g., to represent the operational environment of the software to be developed, the functional requirements, the security requirements, and so on. UML4PF especially supports *security requirements engineering* as described in D6.2 [5, Sect. 2.2] in more detail. UML4PF replaces Jackson's original notation for representing the diagrams by defining a corresponding UML [42] profile. UML4PF consists of an *Eclipse plugin* that allows software engineers to work with the defined profile. Furthermore, the UML profile is complemented by a large number of *validation conditions* expressed in OCL [41] that make it possible to perform semantic checks on the developed diagrams.

Figure 4.7 provides an overview of the UML4PF realization. Gray boxes denote re-used components, whereas white boxes describe those components that are developed at UDE. The functionality of the tool comprises the following:

- The *UML Profile for Problem Frames* defines the relevant stereotypes for the approach.
- The *Requirements Editor* allows one to add new requirements.
- The *Model Generator* automatically generates model elements.
- The *OCL Validator* checks if the model is valid and consistent by evaluating appropriate *OCL expressions*. It also returns the location of invalid parts of the model. All in all, we have defined about 50 OCL validation conditions for the analysis phase.
- The *sdgen Editor* is used to edit sequence diagrams.
- The *Interactive ModelTransformer* serves to create software architectures through interactive model transformations.

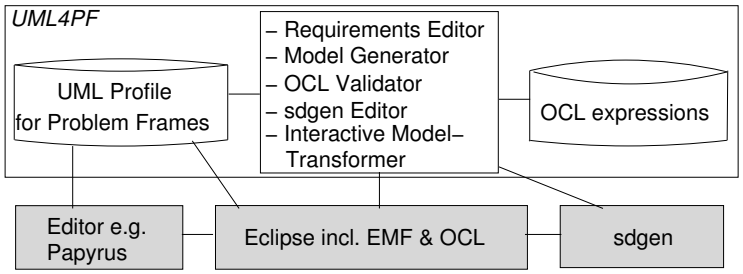

Figure 4.6: UML4PF tool realization overview

With these functionalities, UML4PF supports software engineers in developing a coherent and complete set of requirements documents. A more detailed description of UML4PF and its capabilities can be found in Deliverable D6.2 [5].

4.7.2 Integration into the SDE

The technical details of the integration in the SDE are described in the following:

Technical Requirements	Eclipse 3.5 (Galileo) [25] Eclipse Plugin Papyrus 1.12 [45] Eclipse MDT OCL SDK (OCL Console) [26]
License	Eclipse Public License 1.0 [30]
Eclipse Update Site	`http://www.nessos-project.eu/svn/WP2/integratedTools/UML4PF/`
Installation Guide	As usual in Eclipse. See `http://www.uml4pf.org/installation.html` for a detailed tutorial.
Tool input types	User input via GUI, model files (*.uml)
Tool output types	User output via GUI, model files (*.uml)

The UML4PF SDE plugin currently offers four functions:

- `loadUML4PFModel` has no input parameters and no return value. It shows a file selection dialog to the user, and then loads the selected UML4PF model file (*.uml). If the model is successfully loaded, it is made available via the blackboard.
- `storeUML4PFModel` has no input parameters and no return value. It stores a loaded model into the same file that is previously loaded.
- `generateModelElements` has no input parameters and no return value. It auto-generates required model elements using the Model Generator, e.g., it generates based on an association representing an interface with shared phenomena between domains an interface class with methods corresponding to these phenomena.

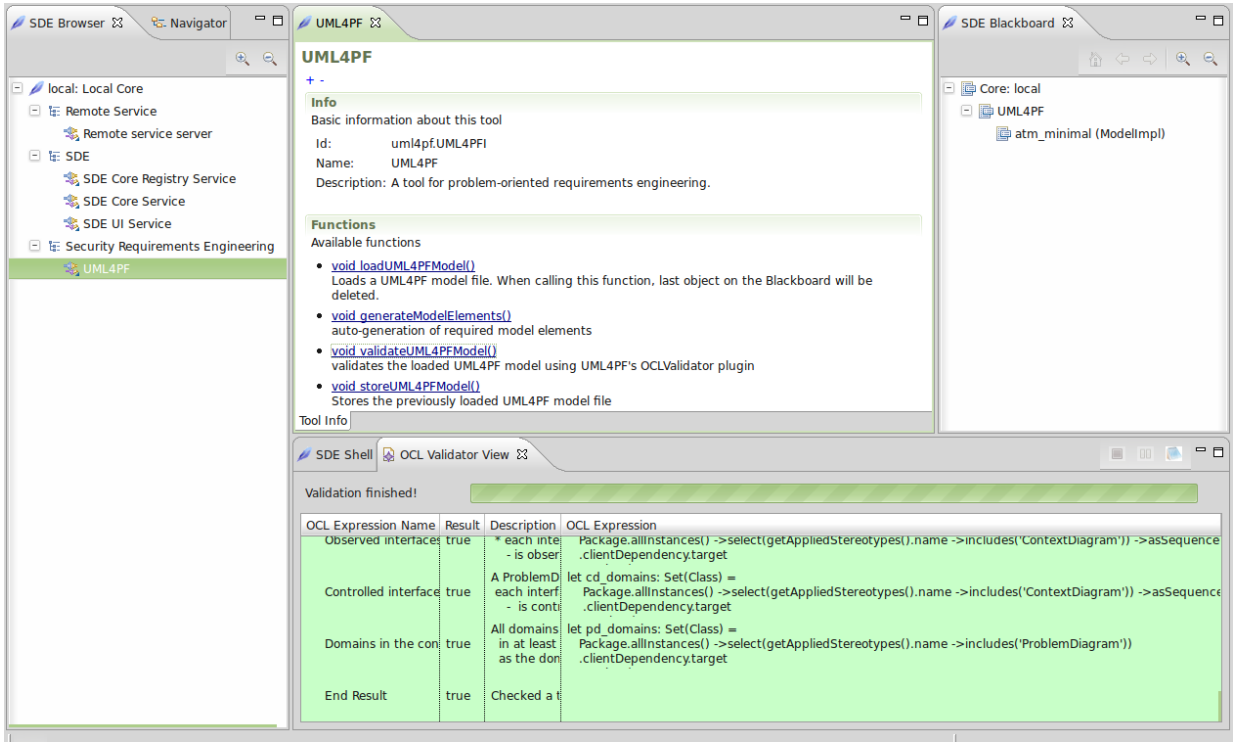

Figure 4.7: Execution of UML4PF OCL validator within SDE

- `validateUML4PFModel` has no input parameters and no return value. It validates a loaded UML4PF model using the OCL Validator, which returns true iff the loaded UML4PF model is valid, and false otherwise. The UML4PF wrapper plugin does not return any value directly, because the OCL Validator of the UML4PF plugin shows the whole validation progress to the user.

4.8 VeriFast

VeriFast is a verifier for single-threaded and multithreaded C and Java programs annotated with preconditions and postconditions written in separation logic.

4.8.1 Features

All applications contain (subtle) bugs, and finding them is costly and often very laborious. Even after spending any amount of time testing and doing code reviews, some bugs may still be present in the code. For some mission critical applications, this is simply not good enough. VeriFast [34, 33] is a verification tool for single-threaded and multithreaded C and Java programs that allows you to prove that annotated source code does not contains bugs from a number of different categories. In particular, all array index out of bounds errors, null pointer dereferences, synchronization problems, and certain other types of errors can be found using VeriFast. It can also be used to prove functional correctness of a program.

Specifically, VeriFast is a verifier for C and Java programs annotated with preconditions and postconditions expressed in separation logic. To enable rich specifications, the programmer may define inductive datatypes, primitive recursive pure functions over these datatypes, and abstract separation logic predicates. To enable verification of these rich specifications, the programmer may write lemma functions, i.e., functions that serve only as proofs that their precondition implies their postcondition. The verifier checks that lemma functions terminate and do not have side-effects. Since neither VeriFast itself nor the underlying SMT solver need to do any significant search, verification time is predictable and low.

4.8.2 Integration into the SDE

The technical details of the integration in the SDE are described in the following:

Technical Requirements	The SDE plugin currently works on Windows only.
License	unspecified
Eclipse Update Site	http://www.cs.kuleuven.be/~bartj/verifast_sde_plugin
Installation Guide	Windows only for now. Apart from that, it should just work.
Tool input type	Java or C source code files, with VeriFast annotations in specially marked comments.
Tool output type	The message “0 errors found”, or the line and column of a potential error in the input source code file.

The VeriFast SDE plugin offers two functions: `verifyProgram` and `showProgramInVeriFastIDE`.

Function `verifyProgram` takes as input the name of an annotated C or Java program. It verifies the program and returns the result of the verification: either the message “0 errors found”, which implies that the program is free from array index out of bounds errors, null pointer dereferences, invalid API method calls, race conditions on variables, certain types of deadlocks, and violations of application-specific correctness criteria specified through the annotations; or the source location of a potential error. If no program name is provided, an example program is verified.

Function `showProgramInVeriFastIDE` takes as input the name of an annotated C or Java program. It then launches the VeriFast Integrated Development Environment (IDE) and shows the program inside the IDE. The user may then instruct the IDE to verify the program, by pressing the Play button. If verification fails, the IDE shows a symbolic execution trace leading up to the error. By selecting a step of the trace, the user can inspect the symbolic representation of heap memory, the local variable store, and the assumptions at this step. If no program name is provided, an example program is shown.

5 Summary and Outlook

In this deliverable, we have examined the requirements for service-oriented tool workbenches and briefly presented related work. The requirements were used to evaluate the workbench that was chosen for the NESSoS project: the Service Development Environment (SDE).

Based on a service-oriented architecture itself, the SDE contains an increasing number of engineering tools for secure Future Internet (FI) software services and systems. The SDE not only provides tool information in a uniform way but also allows (remote) invocation of tool functionality and enables composition of tools by a textual as well as a graphical orchestration mechanism.

We believe that providing individual development tools as services and including features like self-describing services, remote invocation, and orchestration into our tooling environment greatly extends the applicability of the integrated tools. Furthermore, usability increases due to the use of tool chains that enables the use of tools without understanding the full details of the underlying technique.

The SDE, including the integrated tools, is available for download at our dedicated tooling website, http://www.nessos-project.eu/sde. The website also contains a tutorial, a bug tracker and videos demonstrating the SDE in action.

About one third of the tools that are described in the CBK is already integrated. Examples for tools that have been integrated into the SDE during the first year of the NESSoS project are: Avantssar-atse (CL-ATSE), Avantssar Orchestrator, CORAS Tool, EOS (Eye OCL Software), Jalapa, MagicUWE, UML4PF and VeriFast. The crucial point of course is to continue to enhance the integration of tools into the SDE. One of the next steps is to provide an example of tool orchestration and to investigate further ways of orchestration using several tools. Due to the fact that the SDE is a generic framework, it is able to cope with all kinds of tools. Nevertheless, the SDE will be further developed addressing security specific requirements as soon as they arise.

Bibliography

[1] Activiti. BPM Platform. http://www.activiti.org/, 2011.

[2] C. Arora and M. Turuani. Validating integrity for the Ephemerizer's protocol with CL-Atse. *Formal to Practical Security*, pages 21–32, 2009.

[3] ASCENS. Autonomic Service Component Ensembles. http://www.ascens-ist.eu/, 2011.

[4] ASCENS. D6.1: SCE Tooling – Tool Integration Requirements and Technology, 2011.

[5] Y. Asnar, F. Massacci, R. Scandariato, H. Schmidt, L. M. S. Tran, M. R. V. Alvarez, and J. A. Martín. NESSoS Deliverable D6.2 – First Version of the Security Requirements Modelling Language and an Initial Solution for the Scenarios. 2011.

[6] AVANTSSAR. Deliverable 4.2: AVANTSSAR Validation Platform v.2. http://www.avantssar.eu, 2010.

[7] AVANTSSAR. Deliverable 2.3 (update): ASLan++ specification and tutorial. http://www.avantssar.eu, 2011.

[8] AVANTSSAR project. Automated Validation of Trust and Security of Service-Oriented Architectures. http://www.avantssar.eu.

[9] M. Baldamus, J. Bengtson, G. Ferrari, and R. Raggi. Web services as a new approach to distributing and coordinating semantics-based veri cation toolkits. *Electronic Notes in Theoretical Computer Science*, 105:11 – 20, 2004. Proceedings of the First International Workshop on Web Services and Formal Methods (WSFM 2004).

[10] M. Bartoletti, G. Costa, P. Degano, F. Martinelli, and R. Zunino. Securing Java with Local Policies. *Journal of Object Technology*, 8(4):5–32, 2009.

[11] M. Bartoletti, G. Costa, and R. Zunino. Jalapa: Securing Java with Local Policies. *Electronic Notes in Theoretical Computer Science*, 253(5):145–151, 2009.

[12] M. Bartoletti, P. Degano, and G. L. Ferrari. History-based Access Control with Local Policies. In *Proceedings of Foundations of Software Science and Computation Structures (FOSSACS)*, 2005.

[13] M. Bartoletti, P. Degano, and G. L. Ferrari. Plans for Service Composition. In *Workshop on Issues in the Theory of Security (WITS)*, 2006.

[14] M. Bartoletti, R. Zunino, M. Bartoletti, and R. Zunino. Locust: a tool for checking usage policies. Technical report, University of Pisa, 2008.

[15] C. Bartolini and A. Bertolino. NESSoS Deliverable D1.2 – NESSoS Joint Virtual Research Lab. 2011.

[16] M. Busch. Integration of Security Aspects in Web Engineering. Master's thesis, Ludwig-Maximilians-Universität München, 2011. http://uwe.pst.ifi.lmu.de/publications/BuschDA.pdf.

[17] M. Busch, A. Knapp, and N. Koch. Modeling Secure Navigation in Web Information Systems. In J. Grabis and M. Kirikova, editors, *10th International Conference on Business Perspectives in Informatics Research*, LNBIP. Springer Verlag, 2011.

[18] M. Busch and N. Koch. MagicUWE — A CASE Tool Plugin for Modeling Web Applications. In *Proc. 9th Int. Conf. Web Engineering (ICWE'09)*, volume 5648 of *Lect. Notes Comp. Sci.*, pages 505–508. Springer, 2009.

[19] M. Busch and N. Koch. NESSoS Deliverable D2.1 – First release of Method and Tool Evaluation. 2011.

[20] CBK. Common Body of Knowledge. http://www.essos-cbk.org/, 2011.

[21] M. Clavel, M. Egea, and M. A. G. de Dios. ECEASST Building an Efficient Component for OCL Evaluation. *ECEASST*, 15, 2008.

[22] CORAS method. CORAS tool. http://coras.sourceforge.net/, 2010.

[23] I. Côté, D. Hatebur, M. Heisel, and H. Schmidt. UML4PF – a tool for problem-oriented requirements analysis. In *Proceedings of the International Conference on Requirements Engineering (RE)*. IEEE Computer Society, to appear in 2011.

[24] CPL. Common Public License Version 1.0. http://www.opensource.org/licenses/cpl1.0, 2011.

[25] Eclipse Foundation. Eclipse Galileo. http://www.eclipse.org/galileo/, 2011.

[26] Eclipse Foundation. Eclipse: MDT – OCL. http://www.eclipse.org/modeling/mdt/downloads/?project=ocl, 2011.

[27] Eclipse Foundation. The Eclipse Open Source Community and Java IDE. http://www.eclipse.org/, 2011.

[28] S. Eicker, M. Heisel, H. Schmidt, and W. Schwittek. NESSoS Deliverable D5.1 – Common Body of Knowledge. 2011.

[29] EOS. Eye OCL Software. http://www.bm1software.com/eos/, 2011.

[30] EPL. Eclipse Public License Version 1.0. http://www.opensource.org/licenses/eclipse-1.0, 2011.

[31] D. Hatebur and M. Heisel. Making pattern- and model-based software development more rigorous. In *Proceedings of International Conference on Formal Engineering Methods (ICFEM)*, volume LNCS 6447 of *Lecture Notes in Computer Science*, pages 253–269. Springer, 2010.

[32] M. Jackson. *Problem Frames. Analyzing and structuring software development problems*. Addison-Wesley, 2001.

[33] B. Jacobs, J. Smans, and F. Piessens. A quick tour of the VeriFast program verifier. In *APLAS*, 2010.

[34] B. Jacobs, J. Smans, and F. Piessens. Verifast. http://www.cs.kuleuven.be/~bartj/verifast/, 2011.

[35] LMU. Web Engineering Group. MagicUWE. http://uwe.pst.ifi.lmu.de/toolMagicUWE.html.

[36] M. S. Lund, B. Solhaug, and K. Stølen. *Model-Driven Risk Analysis – The CORAS Approach*. Springer, 2011.

[37] T. Margaria. Web services-based tool-integration in the ETI platform. *Software and Systems Modeling*, 4:141–156, 2005. http://eti.cs.uni-dortmund.de.

[38] P. Mayer. *MDD4SOA: Model-Driven Development for Service-Oriented Architectures*. PhD thesis, LMU München, 2010.

[39] P. Mayer and I. Ráth. *Rigorous Software Engineering for Service-Oriented Systems*, chapter The Sensoria Development Environment. Springer, 2011.

[40] No Magic Inc. Magicdraw. http://www.magicdraw.com/.

[41] OMG. OCL 2.0. http://www.omg.org/spec/OCL/2.0/.

[42] OMG. UML. http://www.uml.org/.

[43] OMG. BPMN. http://www.bpmn.org/, 2011.

[44] OSGi Alliance. OSGi Specification Release 4. http://www.osgi.org/Specifications/, 2008.

[45] Papyrus. Open Source Tool for Graphical UML2 Modeling. http://www.papyrusuml.org/, 2011.

[46] ProcessMaker. Web Services API Trigger Builder. http://www.processmaker.com/, 2011.

[47] SDE. Service Development Environment. http://www.nessos-project.eu/sde, 2011.

[48] SDE. Tutorial. http://sde.pst.ifi.lmu.de/trac/sde/wiki/Tutorial, 2011.

[49] SeCSE. Service Centric Systems Engineering. http://www.secse-project.eu/, 2007.

[50] SENSORIA. Deliverable D7.4b. http://www.pst.ifi.lmu.de/projekte/Sensoria/del_24/D7.4.b.pdf, 2006.

[51] Sensoria Project. Software Engineering for Service-Oriented Overlay Computers. http://www.sensoria-ist.eu/, 2011.

[52] M. Turuani. The cl-atse protocol analyser. *Term Rewriting and Applications*, pages 277–286, 2006.

[53] University Duisburg-Essen. UML for ProblemFrames (UML4PF). http://www.uml4pf.org/.

[54] W3C Recommendation. WSDL. http://www.w3.org/TR/wsdl20/, 2011.

[55] Wikipedia. List of cloud platforms. http://en.wikipedia.org/wiki/Category:Cloud_platforms, 2011.