Sensoria

016004
Software Engineering for Service-Oriented Overlay Computers

D7.4b: Report on the Sensoria CASE Tool

Description and Evaluation

Lead contractor for deliverable: LMU
Author(s): Philip Mayer (LMU), Hubert Baumeister (DTU)

Due date of deliverable: 31.08.2007
Actual submission date: 31.08.2007
Revision: Final
Classification: PU

Contract Start Date: September 1, 2005 Duration: 48 months
Project Coordinator: LMU
Partners: LMU, UNITN, ULEICES, UWARSAW, DTU, PISA, DSIUF, UNIBO, ISTI, FFCUL, UEDIN, ATX, TILab, FAST, BUTE, S&N, LSS-Imperial, LSS-UCL, MIP, ATXT

Integrated Project funded by the European Community under the "Information Society Technologies" Programme (2002—2006)

Executive Summary

This document contains a description and evaluation of the Sensoria CASE[1] tool (SCT), which is intended to support the development of service-oriented software by integrating the various tools developed as part of the Sensoria project as well as useful external tools.

The tools developed within Sensoria range from modelling over analysis and transformation to deployment and runtime tools. SCT is thus able to support the complete chain of writing, analysing, deploying, and running service-oriented software systems.

In this document, the current state of the CASE tool is reported. Starting with the requirements and underlying aims, we will report on the evolution of the CASE tool, motivate design decisions, describe the implementation of SCT, and finally discuss integrated tools and how they can be used in collaboration.

This deliverable shows that the Sensoria CASE tool has been successfully used to integrate various Sensoria tools, and for orchestrating them to be used in combination.

Contents

Computer Aided Software Engineering

1 Introduction

In this chapter, we will provide an overview of the aims of Sensoria CASE tool (SCT), reflect on the requirements identified in the previous deliverable, summarize the findings, and give an outline of the remaining chapters.

1.1 About the Sensoria CASE Tool

The Sensoria CASE tool is intended to support the development of service-oriented software by integrating the various tools developed as part of the Sensoria project as well as external tools to support the development, analysis, deployment, and execution of service-oriented software systems. By integrating them into the Sensoria CASE tool, these tools become available to a broader user range and in a larger context, and are thus more usable by developers.

The primary target users of the CASE tool are software developers involved in the creation of service-oriented systems. As a consequence, the aim of the CASE tool is to

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

To enable this vision, the CASE tool must feature an IDE-like user interface which integrates a variety of tools in a coherent fashion, allowing users to quickly assemble and combine the tools they need for a certain task.

Besides the primary target users, SCT will also be used by developers of tools. In order to reach a high number of integrated tools, the integration itself should be fast, simple, and non-invasive. The CASE tool therefore needs to provide tools for tool integration as well.

1.2 Initial Requirements Revisited

In the previous deliverable [1], several initial requirements were listed for the Sensoria CASE tool:

- The user should be able to **use** Sensoria tools through the CASE tool. Usage is either
  - manual, i.e. the user calls each tool separately, or
  - orchestrated, i.e. the users may use arbitrary orchestration mechanisms for calling tools in combination.
- Tools should be easy to **write**, **add** to SCT, and **remove** from SCT. In particular:
  - The CASE tool should not require a certain platform or language, i.e. integration of remote or legacy applications should be possible.
  - The CASE tool should only require lightweight extensions to add tools to the platform.
  - The user should also be able to create new tools from orchestrations.
- SCT should allow **publishing** of tools and **discovering** existing tools published by others.

In the first development phase of the Sensoria CASE tool (months 18 to 24), three distinct prototypes of SCT have been implemented to meet the aims and requirements detailed above.

Through this process, important insights were gained with regard to how the corresponding functionality can be supported in a coherent fashion. All of above requirements have been implemented in the current version 3 of SCT.

1.3 Results

The current version of SCT – version 3 – is now in active use by the Sensoria community. About one third of the Sensoria tools have been integrated into SCT and made available for tool composition, thus already offering state-of-the-art research to end users. Being built on OSGi and the Eclipse platform, SCT integrates into an industry-standard IDE and extends this IDE for developing service-oriented software based on rigorous formal methods, and through a scripting engine, for creating new tools by composition.

SCT now provides the following functionality:

- Developers may use the tools which are provided by the Sensoria partners:
    - SCT offers a rich UI for interacting with tools in a graphical manner.
    - A scripting engine is provided for composing tools by using JavaScript.
    - Tools may also provide their own UI for the end users.
- Tool writers may easily write tools for SCT**,** add tools to SCT**,** and remove tools from SCT:
    - A lightweight, descriptive approach is taken to tool registration. Tool support is provided for this step.
    - Tools may be written in any language or paradigm, as long as an adapter is provided to the OSGi platform.
    - The platform itself may run independently of Eclipse.
- Developers can use an update site to retrieve the current version of the CASE tool. As tools are available from update sites as well, developers can easily create a development environment customized to their needs.

SCT version 3 provides a stable platform for tool integrations and development of service-oriented software, adding an additional layer of tools and techniques on top of the Eclipse IDE. For the next development cycle, some additional features and requirements have been identified in section 4.3.

1.4 Structure of this Deliverable

In this document, the current state of the CASE tool is reported:

- In the next chapter, we will report on the evolution of the CASE tool.
- Chapter 3 details the implementation of the current version of the CASE tool.
- In chapter 4, we will report on integrated tools and how they can be used in collaboration.
- We conclude in chapter 5, putting the results into perspective between work which has already been completed and which still needs to be done.

2 Evolution of the Sensoria CASE Tool

As pointed out in the previous chapter, the first generation of the Sensoria CASE tool went through three distinct versions in its development. The first version was based on OSGi [5], featuring a textual user interface via a UNIX-like shell and an initial service-oriented architecture. The second version extended the OSGi version with a graphical user interfaced based on Eclipse [3], thus adding a graphical interface and the ability to interact with other Eclipse plugins. Finally, in the third version more of the Eclipse technology and frameworks (like wizards) were used while still relying on OSGi for the tool integration. The third version thus combines the best of both worlds, being based on OSGi for its core functionality and on Eclipse technology for ease of use.

2.1 Version-Independent Concepts

All three versions follow the same basic concepts in their design and implementation in order to provide a service-oriented platform for (development) tool integration. These concepts are:

- Tools are services, and provide arbitrary functionality.
- Tools can be used as-is, or combined using orchestration mechanisms.
- Tools can be published and discovered.

In the view of the Sensoria CASE tool, the Sensoria tools each consist of functions, which can be invoked in the CASE tool with or without User Interface (UI). The UI is not necessarily tied to a specific function, but can also be provided in a cross-function (tool-scoped) way. These functions are also provided through an Application Programming Interface (API), allowing tool orchestration with arbitrary (contributed) orchestration languages.

Note that SCT consists of a core (the actual SCT functionality) and potentially many Sensoria tools. In this deliverable, we will use the term SCT for the core either with or without the Sensoria tools, as appropriate in the respective context.

2.2 Version 1

Having identified the OSGi platform as an established framework for implementing service-oriented systems in Java, the first version of the CASE tool was implemented directly on top of this platform. In OSGi, services are implemented as so-called *bundles* which provide arbitrary services in the form of Java objects and methods. Thus, fundamentally, SCT tools will be implemented as OSGi bundles and therefore use the Java language. Note, however, that a bundle may well be a wrapper for external services – for example, applications written in other languages like C or for Web services running on a completely different server. Some existing tools already use such wrappers.

As OSGi in itself is only a component framework, a simple text-based user interface in the spirit of a UNIX shell was created to interact with the bundles. This shell was based on BeanShell, a Java based scripting language [2].

An evaluation of the implementation of the first version has shown the following insights:

- The use of bundles and Java-based OSGi services objects has the right abstraction layer for representing and publishing the Sensoria tools.
- The use of a shell for interaction with Sensoria tools provides, in essence, a manual orchestrator which can be of great help for testing and debugging the tool APIs.
- As not all resulting objects of tool invocations are serializable, these intermediate objects must be stored as volatile objects.

In order to move on to an integrated development environment where a complete project's life cycle can be handled from within one environment, the first version was subsequently extended to use the Eclipse platform, as detailed in the next section.

2.3 Version 2

As pointed out above, the second version of the Sensoria CASE tool was implemented on top of the Eclipse platform. Eclipse is an open source framework which is focused on creating an open development platform for building, deploying and managing software across the lifecycle. As such, it provides an ideal platform for the implementation of the Sensoria CASE tool. In particular,

- Eclipse is based on OSGi itself, and all of its components are available as bundles.
- Eclipse consequently follows the extensibility paradigm; thus, it is easy to extend with new functionality.
- There are many tools already available with an Eclipse integration which can be used in the context of developing service-oriented software.
- Eclipse already provides a highly usable IDE-like graphical user interface, which uses the Standard Widget Toolkit (SWT) [6] as its basic UI framework, but may also be used with the Java Swing platform.

The implementation of the Eclipse version of the CASE tool continued to be based on the use of bundles and OSGi services as tool implementations, as this approach comes naturally to Eclipse as well. In addition:

- A graphical user interface, based on Eclipse/SWT, was added to
  - enable users to browse tools.
  - enable users to browse volatile resources.
- BeanShell scripts, stored as files, could now be executed directly within the platform.

By building on Eclipse, the second version has greatly increased the usability of the Sensoria CASE tool.

2.4 Towards Version 3

From the experience gained from the first two implementations of SCT, additional requirements and features were identified to be implemented in the third version of the CASE tool:

- In order to use basic tool functionality without requiring the Eclipse platform, both the CASE tool core and the implemented tools should follow a strict separation between the OSGi platform and the Eclipse framework – using OSGi for the services themselves and Eclipse for the user interface.
- Tool orchestration and tool discovery are among the research topics of Sensoria, thus the CASE tool should allow arbitrary orchestration and discovery mechanisms at the core level. Therefore, SCT should not be dependent on the BeanShell.
- As the Eclipse platform is based on OSGi, a lot of bundles are loaded at startup. In order to cut down on startup time, Eclipse uses a lazy loading mechanism based on declarative service specifications. In order to benefit from this mechanism and to achieve a more user-friendly service registration, the Sensoria tools should be registered and used in a declarative way as well.

As a consequence, version 3 of the CASE tool was developed. This is the current version and will be detailed in the next chapter.

3 Design and Implementation of the Sensoria CASE Tool

This chapter gives an overview of the design and implementation of the third and current version of the Sensoria CASE tool. We will start with a description of underlying technologies, move on to the general design of the CASE tool, and finally describe the implementation.

3.1 SCT Architecture

To allow development, analysis, deployment, and execution of service-oriented software systems, the CASE tool implementation needs to deal with the following technical requirements:

- It needs to accommodate vastly different tools, which each come with their own set of APIs and data types. It is thus not possible to define a common API for the tools. In SCT, this problem is solved by using (declarative) OSGi services for each tool.
- As tools range from user-intensive graphical modelling tools to computationally intensive analysis tools, some tools might need a sophisticated graphical user interface, while others only provide a command line-like interface. Both is possible within SCT, as it provides a generic invocation UI for each tool, but at the same time also allows custom tool UIs to be integrated.
- To be able to integrate as many tools as possible, requirements for tool builders should be kept low. SCT re-uses OSGi and Eclipse technology and declarative service descriptions which are generated from Java annotations for a fast and straightforward integration process.

The basic architecture of the CASE tool is laid out in Figure 1.

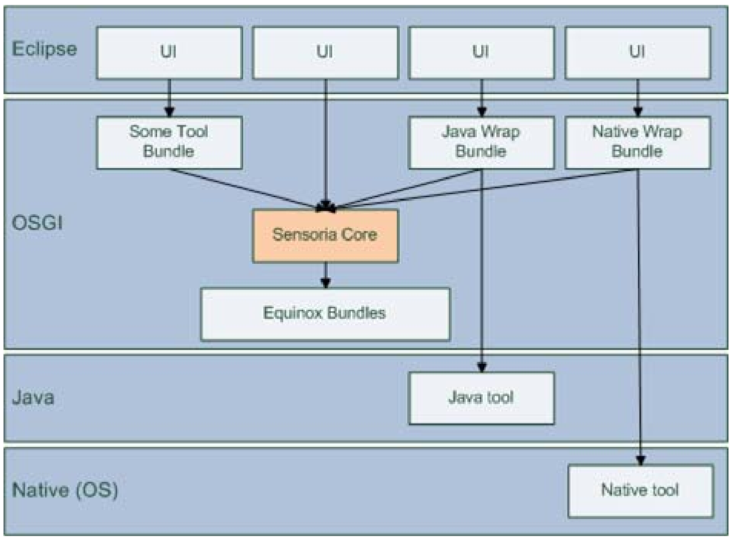


Figure 1: Sensoria CASE tool architecture

As can be seen in the figure, the CASE tool core and the Sensoria tools are based on OSGi only (or, more specifically, the Equinox implementation [4] of OSGi). This means that fundamentally, tools must be implemented in Java, although they may of course wrap native code or remote invocations as they wish. Being only based on OSGi, they can be invoked completely independently from Eclipse. If they additionally choose to provide a UI, this UI is integrated into and based on the Eclipse platform, as is the UI for the Sensoria CASE tool core itself.

As mentioned before, tools can be combined to provide new functions by an orchestration tool. Sophisticated orchestration and service discovery are not provided by the SCT core itself, but are expected to be provided by the Sensoria partners. However, basic discovery and orchestration services are available.

3.2 Basic Functionality

The basic functionality of SCT is threefold:

- It provides access to all the registered tools through an API (which can be seen as a basic discovery service) and 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 through an API and UI as well. The API allows calling arbitrary functions on the registered tools from within Java, while the user interface provides a generic wizard UI for executing these functions, storing the results (on a blackboard), and re-using results as input for other functions.
- It provides a basic orchestrator in the form of a JavaScript based shell. Using JavaScript, the API discussed in the previous two points can be accessed and tools can be orchestrated in a simple way. This implementation is based on the scripting functionality of the Java Development Kit (JDK) version 6.

For most tools, integration into the Sensoria CASE tool is only one way of providing users with access to their functionality. Thus, the Sensoria CASE tool only imposes a minimum number of requirements on tool writers. In particular, tools can be used by developers in any of the following three ways in the CASE tool environment:

- By using the generic wizard UI outlined above.
- By entering commands in a manual orchestrator like the Sensoria shell.
- By using UIs provided by the tools themselves, which are independent of the CASE tool UI.

Besides simply using their functionality, tools can also be orchestrated with partial help from the core. This can be achieved by different means as well. In particular, the following three scenarios are envisioned:

- Orchestration using the built-in shell, repeated by storing the orchestration as a script.
- Orchestration using Java, i.e. within other tools.
- Orchestration by employing special orchestration tools yet to be provided.

As outlined above, the core provides the necessary functionality to retrieve tools to enable such orchestration.

3.3 Core Implementation

The Sensoria CASE tool core has been implemented as two OSGi bundles – the core itself, which is only based on Equinox, and the UI, which is based on Eclipse. An additional development bundle provides useful functionality for developers of new tools.

Additionally, as an example, one orchestrator tool (the shell) has been implemented to enable basic orchestration with the use of JavaScript.

The following figure shows how the various components of the Sensoria CASE tool – as well as some the examples – fit together.

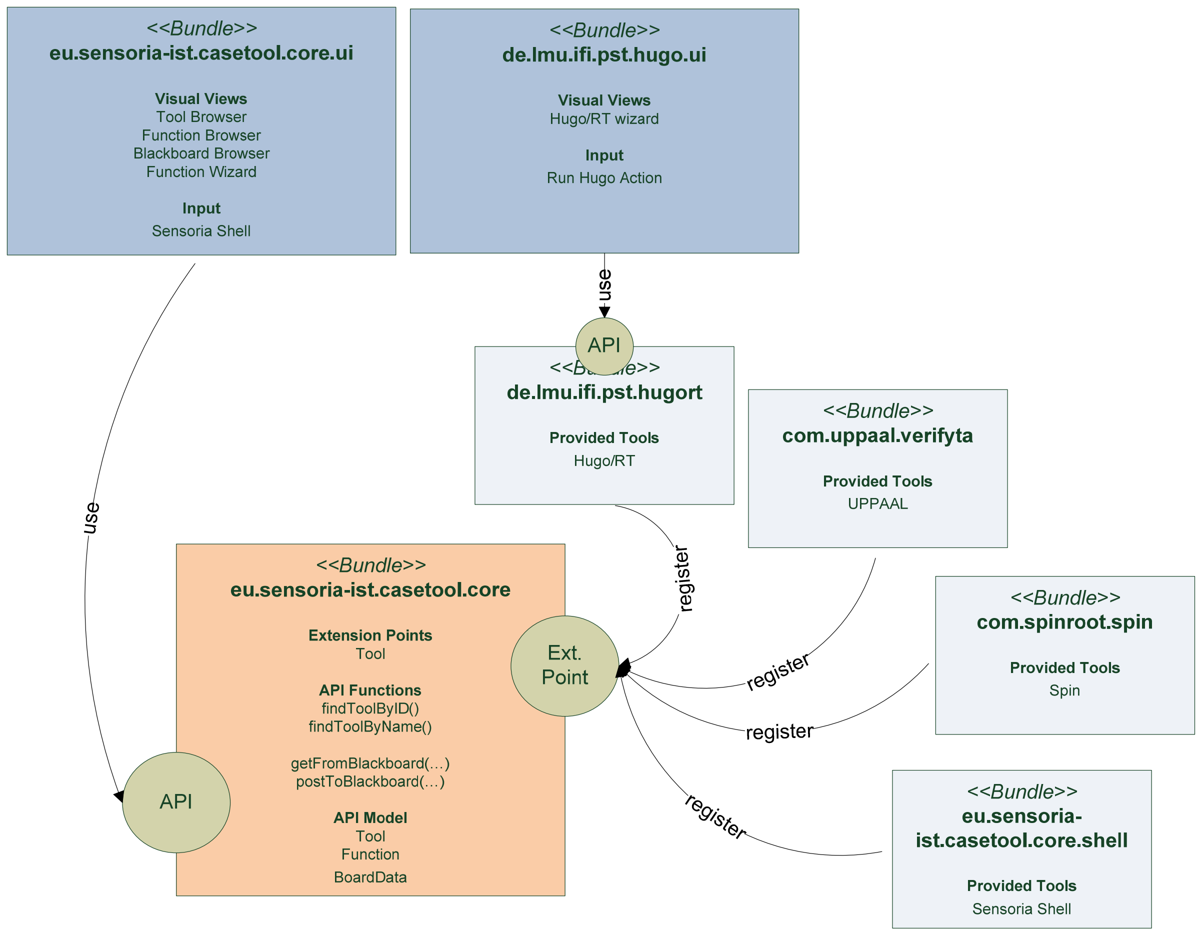


Figure 2: CASE Tool Technical Architecture

In the lower left corner, the SCT core bundle is displayed in orange. It provides two externally accessible interfaces:

- An **Equinox extension point** (on the right-hand side), which can be used for tool registration.
- A **Java API** (on the left-hand side), which can be used by orchestrators, discoverers, and UI to access the registered tools.

In the upper left corner, the Sensoria UI bundle is displayed in blue. It provides several visual views and uses the API of the core to retrieve tools and their functionality.

On the right-hand side, four example tools are displayed (Hugo/RT, UPPAAL, SPIN, and the shell); one of which has an additional UI bundle available (Hugo).

3.3.1 Core Extension Point

The extension point of the SCT core allows tools to register themselves with the core in a declarative fashion. An XML fragment is used to describe a tool along with all of its functions and the implementing class; the Equinox platform then provides this information to the CASE tool core which loads the corresponding functionality.

The next figure shows an example of a tool registration by using the Equinox extension point of the core. In the example, the SPIN bundle is registered with the Sensoria CASE tool core.

```
<extension point="eu.sensoria_ist.casetool.core.tool">
   <tool
      id="com.spinroot.spin"
      name="SPIN">
      description="The SPIN Model Checker"
      category="Analysis/Model Checker"
      class="com.spinroot.spin.RemoteSpinRunner">

      <function
         name="checkWithSPIN"
         description="Checks a Promela and LTL file according to Spin
                      properties, returns a trace in a String array."
         parameters= "java.lang.String, java.lang.String,
                      java.lang.String[]"
         returns=    "com.spinroot.spin.SPINResult"
      </function>
      <option
         name="SpinWebServiceURL"
         description="The URL to the SPIN Web Service."
         defaultValue=http://localhost:8080/axis/services/SpinService" />
   </tool>
</extension>
```

Figure 3: Registering a tool

As can be seen in the figure, the SPIN model checker is registered with the core with an ID, a name, a description, a category, and an implementing class. Furthermore, one function is registered with a name, description, the Java-typed parameters and a Java-typed return type. Additionally, the tool provides an option for the user – in this case, the bundle wraps a Web service, so the Web service URL needs to be configurable.

The above definition has been generated semi-automatically from Java annotations of the SPIN tool interface in the source code. The following figure shows the annotated Java code.

```
@SensoriaTool(
   name= "SPIN",
   description= "The SPIN Model Checker",
   category= "Analysis/Model Checker")

@SensoriaToolOptions( {
   @SensoriaToolOption(
        name="SpinWebServiceURL",
        description= "The URL to the SPIN Web Service.",
       defaultValue= "http://localhost:8080/axis/services/SpinService")
   } )
public interface ISPINService {

   public static final String ID= "com.spinroot.spin";

   @SensoriaToolFunction(
        description= "Checks a Promela and LTL file according to Spin
                      properties, returns a trace in a String array.")
   public SPINResult checkWithSPIN(
        String promelaFileContent,
        String ltlFileContent,
        String[] options) throws SpinException;
}
```

Figure 4: Tool Definition using Java Annotations

3.3.2 Core API

Besides the extension point for registering tools, the core also provides API to orchestrators, discoverers, and its own UI. The most important API functions are listed in the next figure.

```
public interface ISensoriaCore {

	public IToolStore getTools();
	public IBlackboard getBlackboard();
}

public interface IToolStore {

	public Set<Tool> getTools();
	public ToolElement getRoot();
	public Tool findToolById(String toolID);
	public Tool findToolByName(String toolName);
}

public interface ITool {

	public String getId();
	public String getDescription();
	public String[] getCategory();
	public Object getServiceInterface();
}
```

Figure 5: The Sensoria CASE Tool Core API

The functions displayed allow for

- retrieving tools, which can then be used to get the description, category, and service interface for invoking functions.
- retrieving the blackboard on which volatile resources are stored.

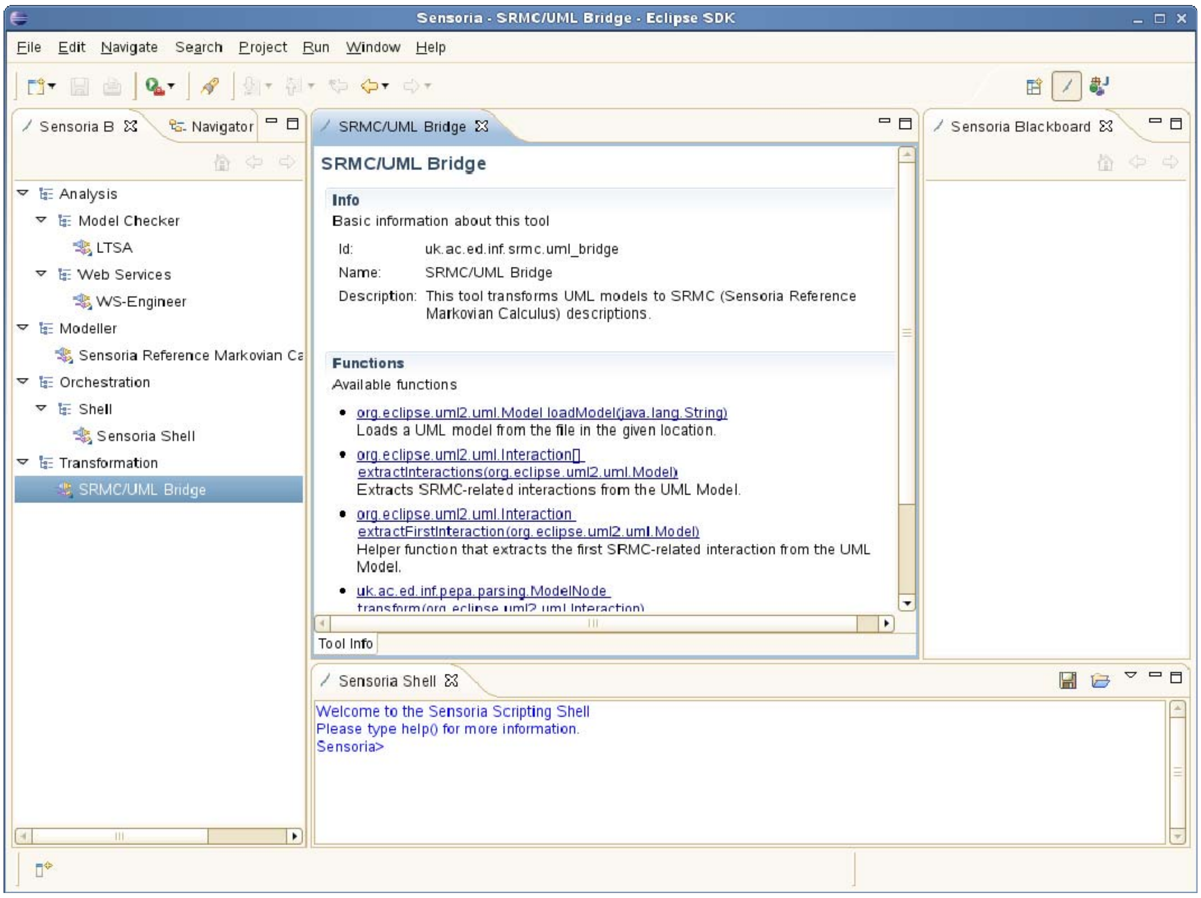
The service interface of each tool directly corresponds to the registered class from the previous section; it can thus be cast to the service interface.

3.4 User Interface

The Sensoria CASE tool provides one new perspective, two new views, and one editor to the Eclipse platform. In addition, the basic orchestrator provided with the SCT core (the Sensoria shell) provides an additional view and a launcher for executing shell scripts.

3.4.1 Sensoria Perspective

The Sensoria CASE tool perspective has the following layout.



SRMC/UML Bridge

Info
Basic information about this tool

Id:	uk.ac.ed.inf.srmc.uml_bridge
Name:	SRMC/UML Bridge
Description:	This tool transforms UML models to SRMC (Sensoria Reference Markovian Calculus) descriptions.

Functions
Available functions

- org.eclipse.uml2.uml.Model loadModel(java.lang.String)
  Loads a UML model from the file in the given location.
- org.eclipse.uml2.uml.Interaction[] extractInteractions(org.eclipse.uml2.uml.Model)
  Extracts SRMC-related interactions from the UML Model.
- org.eclipse.uml2.uml.Interaction extractFirstInteraction(org.eclipse.uml2.uml.Model)
  Helper function that extracts the first SRMC-related interaction from the UML Model.
- uk.ac.ed.inf.pepa.parsing.ModelNode transform(org.eclipse.uml2.uml.Interaction)

```
Welcome to the Sensoria Scripting Shell
Please type help() for more information.
Sensoria>
```

Figure 6: Sensoria perspective

Three views and one editor are visible:

- On the left-hand side, the **Sensoria browser** is displayed. It contains a categorized listing of all tools which are currently available in this particular instance of the CASE tool.
- On the right-hand side, the **Sensoria blackboard** is displayed. The blackboard is used to store Java object values in-between service invocations when using the manual generic UI to access tool functions.
- At the bottom, the **Sensoria shell** is displayed. As pointed out above, the shell is a basic orchestrator which can be used to employ JavaScript to call tool functions.
- The **Sensoria function browser** in the centre shows more information about a selected tool. Besides some general information about the tool in the upper section, all available functions and options of the tool are listed in the lower sections.

Tool functions may be invoked by clicking the appropriate link in the function browser (this will invoke the generic wizard UI described in section 3.4.2), by entering commands into the shell (see section 3.4.3), or by using a tool-specific UI.

3.4.2 Generic Wizard UI

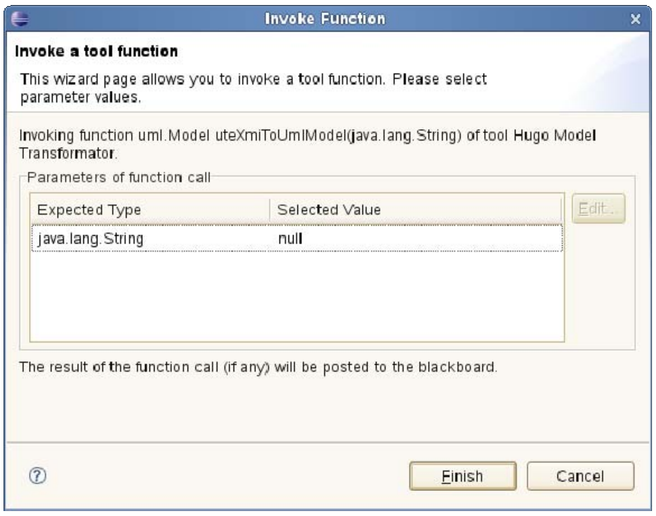
The most basic way for invoking tool functions – which is probably most useful for simple testing and debugging – is provided by the generic wizard UI. This UI allows calling arbitrary functions of arbitrary tools, providing input to those functions from files, strings, or from the blackboard, and posting of the result of the invocation on the blackboard as well.

For example, consider the following function of the Hugo/RT tool:

```
uml.Model uteXmiToUmlModel(java.lang.String)
```

This function transforms an XMI- or UTE-based UML State Machine model into a Hugo/RT UML model. The model is returned as a Java object of class `uml.Model`; the function requires the UTE or XML specification to be given as a `String`.

Invoking the generic wizard on this function yields the dialog in Figure 7.



Invoke a tool function

This wizard page allows you to invoke a tool function. Please select parameter values.

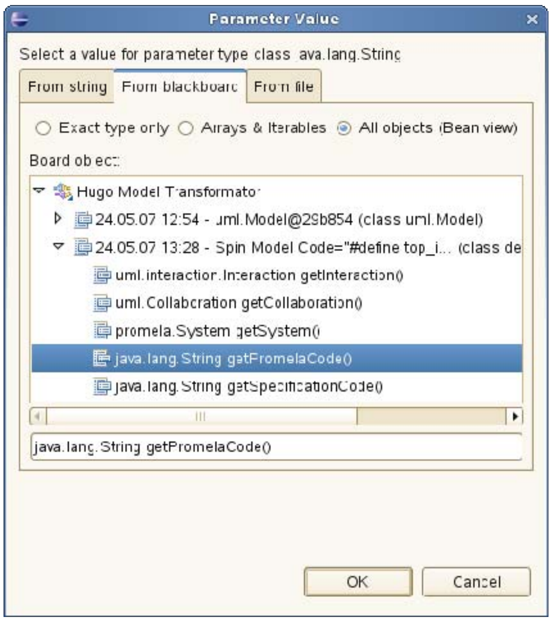
Invoking function uml.Model uteXmiToUmlModel(java.lang.String) of tool Hugo Model Transformator.

Parameters of function call

Expected Type	Selected Value
java.lang.String	null

The result of the function call (if any) will be posted to the blackboard.

Figure 7: Invoking a function



Select a value for parameter type class java.lang.String

( ) Exact type only ( ) Arrays & Iterables (●) All objects (Bean view)

Board object:

```
▽ Hugo Model Transformator
  ▷ 24.05.07 12:54 - uml.Model@29b854 (class uml.Model)
  ▽ 24.05.07 13:28 - Spin Model Code="#define top_i... (class de
      uml.interaction.Interaction getInteraction()
      uml.Collaboration getCollaboration()
      promela.System getSystem()
      java.lang.String getPromelaCode()
      java.lang.String getSpecificationCode()
```

java.lang.String getPromelaCode()

Figure 8: Selecting a Bean

As can be seen from the screenshot, one parameter must to be provided. By selecting the parameter and clicking **Edit**, another dialog pops up which allows choosing a value for the parameter (Figure 8).

In this dialog, you have three choices for selecting a parameter value:

- You can simply provide text in a text field. This is only possible for String-typed parameters.
- You can select a value from the blackboard. This is possible for all kinds of parameter types.
- You can load the input for the parameter from a file.

Figure 8 shows an example of selecting a bean accessor from an object on the blackboard.

3.4.3 Shell and the Scripting Functionality

One of the main aims of the CASE tool is enabling orchestration of tools. As tools can be discovered using the Sensoria core and their interface functions invoked, orchestration can be provided by arbitrary tools on top of this service layer. The Sensoria shell provided within the core is a manual orchestrator which provides such an orchestration mechanism as a UNIX-like shell with the additional ability to store, load, and execute scripts.

The Sensoria shell is based on the JDK 6 scripting engine and employs JavaScript as its language. It allows access to the Sensoria core from within scripts; thus any tool can be retrieved from the core and its interface functions executed. Arbitrary variables may be defined and reused as usual.

Scripts like these can also be written directly using the Eclipse text editor. The recommended file ending for such scripts is `.sscript` for **Sensoria script**. These files can be run in the same way as other executable files within Eclipse by using the launching framework (i.e. selecting *Run* > *As Sensoria Script*).

4 Tool Integrations

SCT is intended to provide an integration platform for the tools developed within Sensoria. In this chapter, we will reflect on tools which have already been integrated into SCT and how they can be used in collaboration.

The process of integrating tools into SCT has two dimensions:

- First of all, the tool must be integrated into the SCT core, i.e. an OSGi bundle and extension point definition along with an API needs to be developed. This step allows users to interact with the tool from within SCT.
- Secondly, the tool must be able to be composed, or orchestrated, with other tools. In order for this to work, the interfaces of the tools and the data types of data transmitted between various tools needs to be well-defined and understood in each tool which takes part of a certain tool chain.

In the following, we will describe the tools which have already been added to SCT or will be added in the near future. The full tool list may be found in the first deliverable.

Afterwards, we will present the tool chains created by composing these tools.

4.1 Tools for the Sensoria CASE Tool

4.1.1 Available Tools

The following tools are available within SCT.

4.1.1.1 Hugo/RT

Category: Transformation

Description: Hugo/RT is a UML model translator for model checking, theorem proving, and code generation: A UML model containing active classes with state machines, collaborations, interactions, and OCL constraints can be translated into the system languages of the real-time model checker UPPAAL, the on-the-fly model checker SPIN, the system language of the theorem prover KIV, and into Java and SystemC code.

4.1.1.2 SPIN

Category: Analysis > Model Checking

Description: Spin is a popular open-source software tool, used by thousands of people worldwide that can be used for the formal verification of distributed software systems.

4.1.1.3 UPPAAL

Category: Analysis > Model Checking

Description: Uppaal is an integrated tool environment for modelling, validation and verification of real-time systems modelled as networks of timed automata, extended with data types (bounded integers, arrays, etc.).

4.1.1.4 ArgoUML

Category: Modelling

Description: ArgoUML is an open source UML modelling tool which includes support for all standard UML 1.4 diagrams.

4.1.1.5 LTSA

Category: Analysis > Model Checking

Description: LTSA is a verification tool for concurrent systems. It mechanically checks that the specification of a concurrent system satisfies the properties required of its behaviour. In addition, LTSA supports specification animation to facilitate interactive exploration of system behaviour.

4.1.1.6 WS-Engineer

Category: Analysis > Web Services

Description: The LTSA WS-Engineer plug-in is an extension to the LTSA Eclipse Plug-in which allows service models to be described by translation of the service process descriptions, and can be used to perform model-based verification of web service compositions.

4.1.1.7 SRMC Core

Category: Modeller

Description: SRMC (Sensoria Reference Markovian Calculus) Core provides support for SRMC, an extension to PEPA. It covers steady-state analysis of the underlying Markov chain of SRMC descriptions.

4.1.1.8 SRMC/UML Bridge

Category: Transformation

Description: SRMC/UML offers facilities for meta-model transformation. It translates a subset of UML2 models (Interactions and State Machines) into an SRMC description for performance evaluation. Results are reflected back into the UML model.

4.1.2 Committed Tools

The following tools are currently under development and will be integrated soon:

4.1.2.1 VIATRA2

Category: Transformation

Description: The main objective of the VIATRA2 (VIsual Automated model TRAnsformations) framework is to provide a general-purpose support for the entire life-cycle of engineering model transformations including the specification, design, execution, validation and maintenance of transformations within and between various modelling languages and domains.

4.1.2.2 TIGER Generator + Designer

Category: Code Generation

Description: The Tiger Project (Transformation-based generation of modelling environments) is a tool environment that allows to generate an Eclipse editor plugin based on the Graphical Editing Framework (GEF) from a formal, graph-transformation based visual language specification. Moreover the Eclipse JET engine is used for generation.

4.1.2.3 TIGER EMF Transformer

Category: Transformation

Description: The Tiger EMF Transformation Project is a framework for in-place EMF transformation based on graph transformation. Transformations are visually defined by rules on object patterns typed over an

EMF core model. Defined transformation systems can be compiled to Java code building up on generated EMF classes.

4.2 Scenarios for Tool Collaboration (Tool Chains)

In the following, we will give an overview of two completed and one committed tool chain as UML activity diagrams. An activity represents the functions of a certain tool, whereas the object nodes represent actual data elements. Note that these diagrams do not contain start or end nodes, as the process may be started at any activity.

4.2.1 Model Checking UML State Machines with Hugo/RT, UPPAAL, and SPIN

In the first tool chain, a UML model (in particular, a state machine) is designed in a UML modeller, transformed to appropriate input for a model checker, and then checked for correctness.

Figure 10 shows the overall flow of data in this tool chain. In the beginning, the UML state chart is created; this may be done with the ArgoUML tool currently implemented in SCT, or any other UML tool yet to be integrated. Afterwards, the model is transformed to the UTE format (an internal format of Hugo/RT) or to input for the UPPAAL or SPIN model checkers. Finally, after one of the model checkers has been invoked, the result may be viewed (which is the trace from the model checking process).

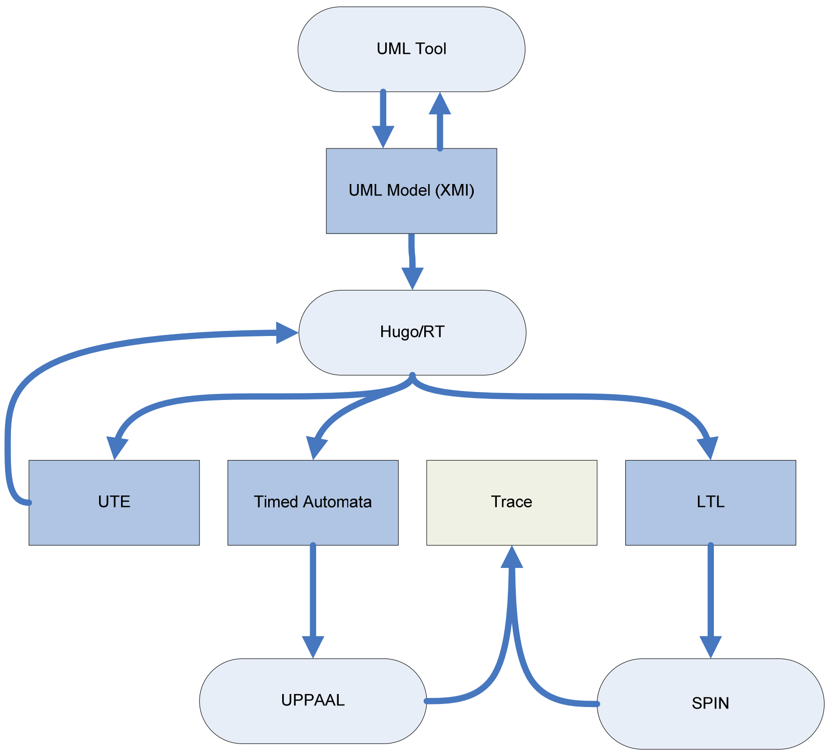


Figure 10: Model Checking with Hugo/RT, UPPAAL, and SPIN

4.2.2 Checking the Specification of BPEL Processes with WS-Engineer and LTSA

In this scenario, an executable service composition is modelled in UML (more specifically, as an activity diagram). This composition is then transformed using the TIGER Transformer or the VIATRA2 transformation tool to BPEL. The BPEL composition is read by WS-Engineer and transformed into Finite State Processes (FSP).

LTSA then checks that the specification of the system satisfies the properties required of its behaviour and generates the final output for the user. Generated traces may be transformed back to UML (for example, to message sequence charts).

Figure 11 shows a graphical overview of this process.

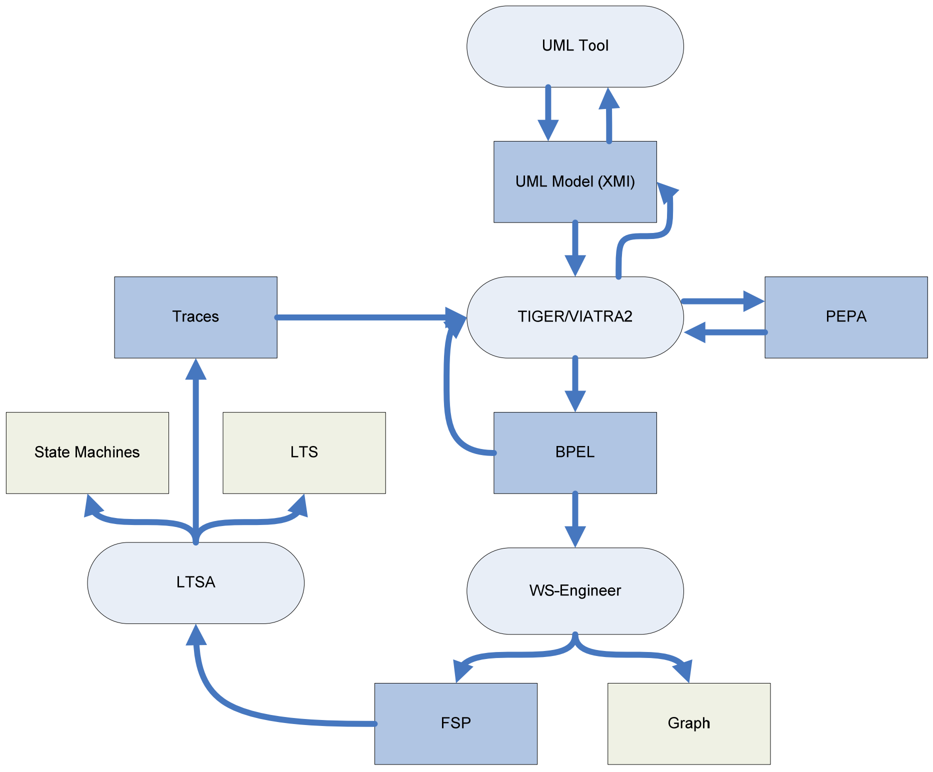


Figure 11: Checking specifications of BPEL processes

4.2.3 Throughput Analysis with SRMC/PEPA

In this tool chain, a throughput analysis is performed on a UML model. The results are given as back-annotations to UML or as a graph.

Figure 12 shows the overall flow of data in this tool chain. After being designed in a UML modeller, a UML model is read by the SRMC tool and SRMC-related interactions are extracted from the model. By using continuous-time Markov chains and steady-state probability distribution over the given state space, a throughput analysis is performed which may then be back-annotated to UML or rendered as a graph for the developer.

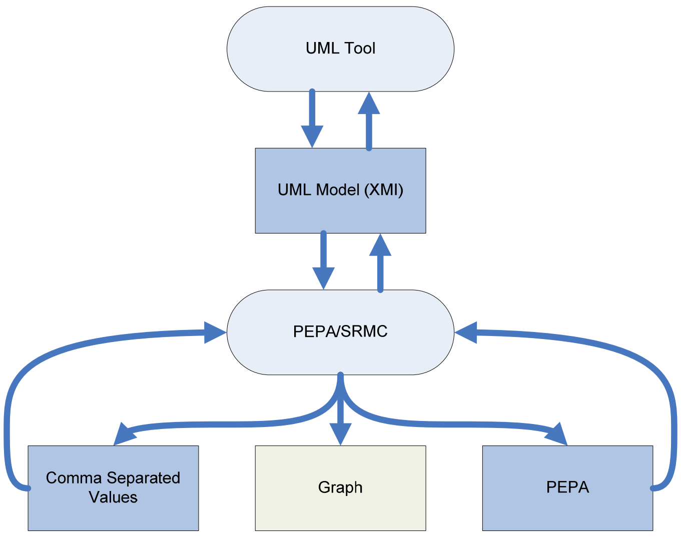


Figure 12: Throughput analysis with SRMC/PEPA

4.3 Evaluation and Requirements

With the tools listed in the previous sections, about one third of the Sensoria tools have been integrated into SCT and made available for tool composition.

This pragmatic evaluation of the CASE tool by the tool integrators themselves has shown tool integration to be usable and straightforward; the integrations were completed very quickly. Still, some requirements were identified for the next generation of the CASE tool which will be implemented from M24 to M36:

- As a tool bundle can only be implemented in Java, several bundles exist which wrap existing functionality written in C or other native languages. Currently, it is the responsibility of each bundle to either require the wrapped application to be installed on the system, or to carry it within the bundle and install it as required. We plan to add functionality to the CASE tool core to render this process easier for tool writers.
- Updates and newly available tools for SCT should be available to users as soon and as comfortable as possible. However, as different tool implementers are involved which are distributed all over Europe, the updating mechanism is inherently distributed. We will look into ways of keeping this distributed nature of tool update sites, yet offer a comfortable way for users to discover updates or new tools.
- Orchestrations of Sensoria tools – whether written using the JavaScript shell or another orchestration language yet-to-be-included in the CASE tool – should be usable as tools themselves. Thus, such orchestrations would need to be bundled, distributed, and used appropriately. We plan to provide this functionality in the next version of the CASE tool.

5 Summary and Outlook

In months 18 to 24 of the Sensoria project, the Sensoria CASE tool has been developed and integrated with nearly a dozen tools, thus laying the cornerstone for an integrated development environment for writing, analyzing, deploying and running service-oriented software systems.

Through three versions, we have identified, implemented, and evaluated various ways for realizing the CASE tool. We have reached a stable version which is highly usable and implements all of the requirements listed in our first deliverable (as detailed in the last two chapters).

Moving forward, the next steps in the development of the CASE tool are as follows:

- The additional requirements listed in the previous chapter will be analyzed, and possible approaches to solve these problems identified.
- More tools will be integrated into SCT as soon as their development process reaches a stable phase. The goal is to integrate and cross-link as many tools as possible.
- Demonstrator instances of the CASE tool will be used for evaluation of the Sensoria and SCT approach by end users, i.e. developers of service-oriented systems, using the case studies (WP8).

The results from these activities will serve as input to the implementation of the final version of the CASE tool, which will be released for month 36.

Bibliography

[1] Baumeister, H. et al. Sensoria Deliverable 7.4a - CASE tool: Initial Requirements

[2] BeanShell. http://www.beanshell.org/

[3] Eclipse Platform. http://www.eclipse.org/

[4] Equinox. http://www.eclipse.org/equinox/

[5] OSGI Platform Specification. http://www2.osgi.org/Download/Release4V41

[6] SWT. http://www.eclipse.org/swt/

Figures