SECONOMICS

D8.2 - Complete design of prototype

A. Schmitz, T. Schnitzler (Fraunhofer ISST)
J. Jürjens (Fraunhofer ISST & TU Dortmund)
D. Ríos (URJC)
J. Williams (ABDN)

Document Information

Document Number	D8.2
Document Title	Complete design of prototype
Version	1.0
Status	Final
Work Package	WP 8
Deliverable Type	Report
Contractual Date of Delivery	31.07.2013
Actual Date of Delivery	31.07.2013
Responsible Unit	Fraunhofer ISST
Contributors	ISST, TUD, URJC, ABDN, UNITN, SNOK
Keyword List	Tool Platform Support
Dissemination level	PU

SEVENTH FRAMEWORK PROGRAMME

SECONOMICS Consortium

SECONOMICS “Socio-Economics meets Security” (Contract No. 285223) is a Collaborative project within the 7th Framework Programme, theme SEC-2011.6.4-1 SEC-2011.7.5-2 ICT. The consortium members are:

1	UNIVERSITÀ DEGLI STUDI DI TRENTO	Università Degli Studi di Trento (UNITN), 38100 Trento, Italy www.unitn.it	Project Manager: prof. Fabio MASSACCI Fabio.Massacci@unitn.it
2	DEEPBLUE	DEEP BLUE Srl (DBL) 00193 Roma, Italy www.dblue.it	Contact: Alessandra TEDESSCHI Alessandra.tedeschi@dblue.it
3	Fraunhofer ISST	Fraunhofer-Gesellschaft zur Förderung der angewandten Forschung e.V., Hansastr. 27c, 80686 Munich, Germany http://www.fraunhofer.de/	Contact: Prof. Jan Jürjens jan.juerjens@isst.fraunhofer.de
4	Universidad Rey Juan Carlos	UNIVERSIDAD REY JUAN CARLOS, Calle TulipanS/N, 28933, Mostoles (Madrid), Spain	Contact: Prof. David Rios Insua david.rios@urjc.es
5	UNIVERSITY OF ABERDEEN	THE UNIVERSITY COURT OF THE UNIVERSITY OF ABERDEEN, a Scottish charity (No. SC013683) whose principal administrative office is at King’s College Regent Walk, AB24 3FX, Aberdeen, United Kingdom http://www.abdn.ac.uk/	Contact: Prof. Julian Williams julian.williams@abdn.ac.uk
6	TMB Transports Metropolitans de Barcelona	FERROCARRIL METROPOLITA DE BARCELONA SA, Carrer 60 Zona Franca, 21-23, 08040, Barcelona, Spain http://www.tmb.cat/ca/home	Contact: Michael Pellot mpellot@tmb.cat
7	Atos	ATOS ORIGIN SOCIEDAD ANONIMA ESPANOLA, Calle Albarracin, 25, 28037, Madrid, Spain http://es.atos.net/es-es/	Contact: Alicia García alicia.garcia@atosresearch.eu
8	SECURENOK	SECURE-NOK AS, Professor Olav Hanssensvei, 7A, 4021, Stavanger , Norway Postadress: P.O. Box 8034, 4068, Stavanger, Norway http://www.securenok.com/	Contact: Siv Houmb sivhoumb@securenok.com
9	SOÚ Institute of Sociology AS CR	INSTITUTE OF SOCIOLOGY OF THE ACADEMY OF SCIENCES OF THE CZECH REPUBLIC PUBLIC RESEARCH INSTITUTION, Jilska 1, 11000, Praha 1, Czech Republic http://www.soc.cas.cz/	Contact: Dr Zdenka Mansfeldová zdenka.mansfeldova@soc.cas.cz
10	nationalgrid THE POWER OF ACTION	NATIONAL GRID ELECTRICITY TRANSMISSION PLC, The Strand, 1-3, WC2N 5EH, London, United Kingdom	Contact: Dr Ruprai Raminder Raminder.Ruprai@uk.ngrid.com
11	ANADOLU ÜNİVERSİTESİ	ANADOLU UNIVERSITY, SCHOOL OF CIVIL AVIATION Iki Eylul Kampusu, 26470, Eskisehir, Turkey	Contact: Nalan Ergun nergun@anadolu.edu.tr

Document change record

Version	Date	Status	Author (Unit)	Description
0.1	2013-05-23	Draft	Andreas Schmitz (ISST)	Table of Contents
0.2	2013-05-23	Draft	Jan Jürjens (ISST)	Verify Table of Contents
0.3	2013-06-20	Draft	Andreas Schmitz (ISST)	Full draft of deliverable
0.4	2013-06-20	Draft	Theodor Schnitzler (ISST)	Assistance
0.5	2013-07-02	Draft	Julian Williams (ABDN)	Contribution of Model
0.6	2013-07-15	Finalizing	Jan Jürjens (ISST)	Review
0.7	2013-07-18	Finalizing	Javier Cano	Review
0.8	2013-07-18	Finalizing	Woohyun Shim (UNITN)	Review and contribution for WP1 model
0.9	2013-07-18	Finalizing	Elisa Chiarani, Martina de Gramatica (UNITN)	Quality check
0.10	2013-07-26	Finalizing	Theodor Schnitzler (ISST)	Assistance
1.0	2013-07-29	Final	Andreas Schmitz (ISST)	Finalized

Index

Executive summary

This document is the second deliverable scheduled for WP8. After having evaluated in D8.1 all tools available by the different partners we have come out with a first prototype for the design of the tool framework and its visualization required by our partners. Of all the tools introduced in D8.1, we will explain which of them will be incorporated into the tool as components and frameworks, and how they will be used. This includes the description of the WP5 implementation.

In this deliverable, we will describe in detail the whole design of the prototype and its implementation.

The architecture we have designed defines a component-wise structure which mainly consists of eight parts:

- The Analyses Plugin Provider, which loads the available analyses
- The Selector, which enables the user to select one
- The Workflow View, which structures the parameter into different steps
- The Generic Parameter, Interface which requests the parameter from the user for the selected Model
- A visual designer “Adversarial Influence Diagram Editor” (AID), which shows the underlying Models for ARA Models
- The Matlab Execution, which runs the Matlab analysis implementation
- The Result View, which plots the result given back by the analysis
- The Matlab implementations, which provides the SECONOMICS-models and analyses

First, the whole structure of the tool will be presented, in order to give an overview of the prototype and then, each component will be explained in detail. The internal functionality, including the underlying conceptual ideas, will also be discussed, showing the deeper structure of code in detail.

We have made use of the Matlab Connection, the Eclipse Framework, the Graphiti Framework, an XML Reader Component and of JFreeChart as the Plotter component.

1 Introduction

This document is the second deliverable scheduled for WP8, and together with D8.3 it is due for M18. This deliverable illustrates the complete design of the prototype, its detailed implementation and the tools, components and frameworks forming part of it. This includes a short description of the implementations of WP5. Deliverable D8.3 will show the implementation of models of work package 6.

1.1 Document Overview

After this general introduction, we will present the realization of the framework in Chapter 2. It will illustrate the structure of the framework and then, it will provide a brief introduction of all the designed components. Following, the processes for using the tool and for integrating new models into it will be shown. In Chapter 3 we will provide an in-depth insight into the different components and the underlying ideas concerning their implementation. As we will see, the most relevant components are the Analyses Plugin Provider for loading the integrated analyses; the Selector which will enable the user to select an analysis; the Generic Parameter Interface, to input parameters for analyses; the Matlab Connection for running the models; and the Result View which will show the calculated result.
Finally, in Chapter 4 we will provide a summary of this deliverable and will define the next steps.

1.2 Short description of the design of the Tool

We have implemented a prototype of the tool. Several parts of it are unfinished from many perspectives and still under development. We will provide a proof of concept for this approach. We have implemented a series of testing models to check the consistency of the prototype. We have structured the tool into several components:

- The Analyses Plugin Provider:
  It automatically searches for available analyses and loads their configuration into memory. It provides this type of information to most of the components, like the selector and executor. It is also possible to load custom analyses from a different path later on from the user interface.
- The Selector:
  The Selector enables the user to select an Analysis by double-clicking on it and creating, in this way, a new instance of it.
- The Workflow View:
  This view helps to display the parameters of one specific analysis and to structure them into different steps. With the aid of such workflows, the user is guided through the specification of the analysis in an intuitive way by filling in the parameters. In addition, some advice is given throughout the different steps.
- The Generic Parameter Interface:
  This is an interface whose functionality is to request the necessary parameters from the user for the current analysis instance. It has a generic and can be also configured in the analysis configuration file for each integrated analysis.
- The visual designer “Adversarial Influence Diagram” (AID) Editor:
  This is a designer to provide the user a tool in order to visualize the structure of the models of WP5. Additionally, the data can be entered directly into these models instead of using the generic interface.
- The Matlab Execution:
  This component runs the Matlab analysis implementation, either compiled or for debugging purposes as a Matlab live session. This module is essential to run analyses using Matlab as the underlying math engine.
- The Result View:
  This View plots the results obtained from the Matlab execution. It can be also configured within the model configuration files. There are different kinds of possible plots as, e.g., bar charts, pie charts and others.
- Matlab Models and Analyses:
  Most Models and Analyses provided by this project are provided as Matlab implementations.

The user would typically start the tool, after which all the models currently included are loaded. The user can then either run one of these preloaded models or load an external one. In any case, a new instance is created and the user can enter all the required parameters step by step through the workflow process and, subsequently, run the analyses. The analysis implemented in Matlab is run in background with the aid of the Matlab runtime environment, returning the desired results upon completion. These results are plotted using predefined graph types, which can be selected by the user.

2 Full design of the tool framework

This chapter can be regarded as a step up with respect to the abstract general concept of the tool published in D8.1 [1]. Now, this concept has been finished almost to the smallest detail and it has been conceptually implemented in a prototype in order to check the consistency of the concept. In section 2.1, we explain a slightly adjustment in the design perspective compared to the proposal. This will improve the understandability of the components and their interaction. In Section 2.2, we will describe the tools, components and frameworks used in our toolbox, most of which were evaluated in D8.1. Following, an overview is given in section 2.3, as well as an intuitive idea of how each component was conceived, we explain the program process with user interaction in Section 2.4 and the model integration process in Section 2.5.

2.1 Adjustment of design perspective

In the first proposal described in D8.1 [1], we have suggested to structure the tool framework only into three parts: The Security Problem Structurer, the Security Problem Modeler and the Security Problem Solver.

As described in D8.1 [1], we had several different models belonging to different work packages. Each model had the aforementioned three parts, including the model editor, the parameter interface and the Matlab execution, as seen in Figure 1. We had planned to structure the implementation in this way, because the Structurer and the Modeler were part of the Framework, which was implemented in Java, whereas the Analyses were implemented in Matlab.

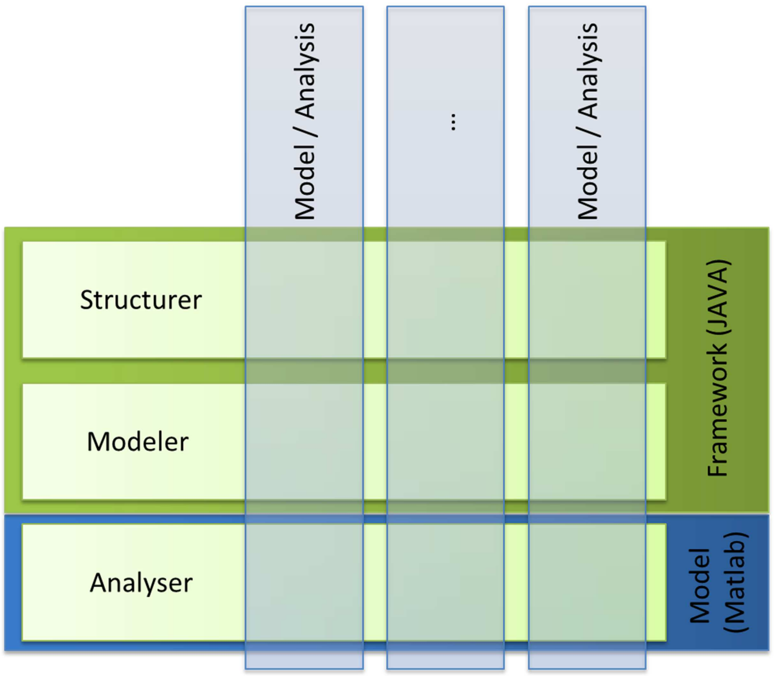


Figure 1 The horizontal and vertical structure of the tool

But while implementing the tool and developing the models, we acknowledged that the models were loosely connected. For each model much specific knowledge about the formulas involved was needed. For this reason, these models could not be implemented at once and, therefore, they were separated. Although each model has the three constitutive parts, some of them can be reused among different models. Summarizing, the primitive structure had several overlapping parts while others were more disjoint in terms of their representation. Therefore, the original structure presented in D8.1 was not easily understandable regarding this segmentation.

In consequence, we decided to split the design of the tool into different components instead of the three parts initially planned. Some of the components will be used by several models, while others are model specific. Special designers and generic parameter interfaces are also included as components. With this component structure, the tool design can be explained in a much easier way. In what follows, we will show this design in detail.

2.2 Used Tools and Frameworks

The framework will contain the analytic models developed by project partners in order to give an integrated view and a good user handling. This framework is based on the Eclipse Framework which is released by the Eclipse Foundation [9]. It defines several utility tool boxes which support the implementation process. For instance, it provides the plugin interface and also Graphiti [8], which is a tool box for visualizing structural diagrams. We have used it for implementing the Adversarial Influence Diagram (AID) Editor. Eclipse and most of its tool boxes are open source and free of charge. Besides, we have chosen Matlab as the mathematical computation engine. It is very sophisticated software, on complex numerical calculations comprising an optimized powerful engine. Although Matlab is not proprietary, the implementations can be compiled and used license free within Java applications. All files will be stored as open and standard XML files. We have used standard Java implementations to parse them. In order to visualize the results arising from Matlab calculations, we have used the open source JFreeChart [10] component, which provides elegant two-dimensional diagrams of various types.

2.3 Design of the tool framework

In this section, we describe the current design of the integrated framework. We first discuss the global structure of the tool, providing also a brief description of all planned components. Then, we describe the definition of an analysis model. A more detailed description of the different components is provided in Chapter 3.

2.3.1 Overview of the design

The tool is divided into two separated parts: (1) the integrated tool framework that is implemented in Java; (2) its conforming models from the modeling work packages which are implemented in Matlab. The tool framework is separated into self-contained components which can be implemented separately, and which will be explained in detail in Chapter 3. Figure 2 shows the different components and their interaction. Following, we provide a brief description of each component.

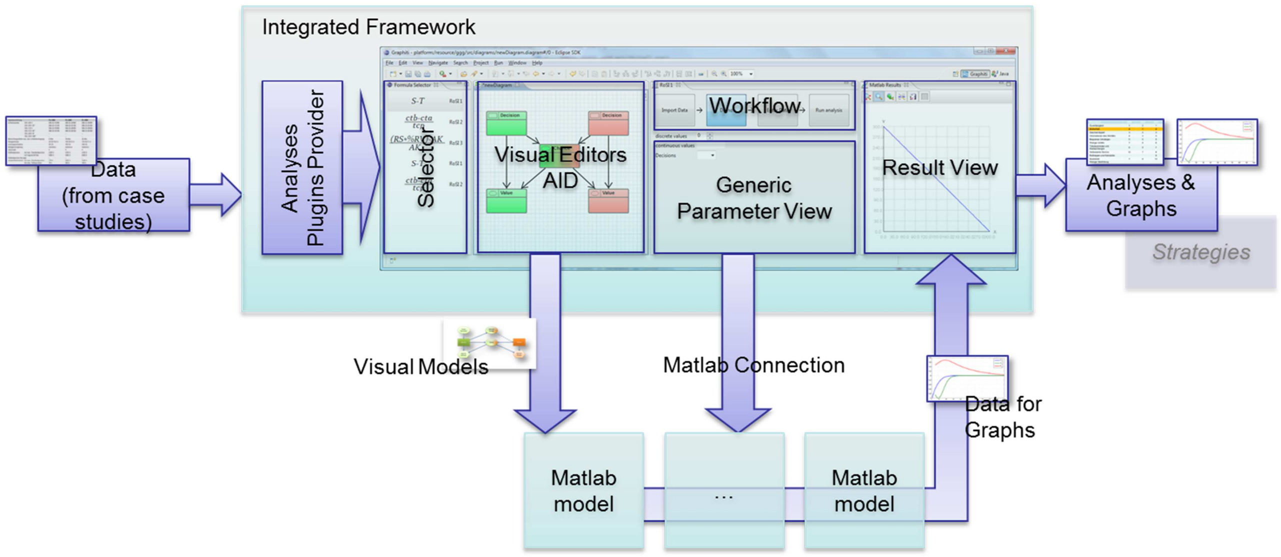


Figure 2 The complete design of the integrated tool framework

Analyses Plugins Provider

The first component is the Analyses Plugins Provider, which loads the Matlab analyses together with the associated configuration files, stores the information about the analyses and passes them to other components. It reads all data corresponding to a model out of the configuration file and other associated files. The structure of these model configuration files is discussed in Section 3.10. The configuration file contains general information about the model as, e.g. its name, the icon file name and a short description of the model. It also has information about the workflow and their parameters. Each parameter associated to a workflow has a name and a type, which can be, for example, numeric, character, etc. The file also contains the Matlab connection details as, e.g. the model file name and the name of the function which should be called. The last information to be loaded is about the different diagram views, among which the user can choose how to show the results of the analysis. All these data is stored in the Analyses Plugin Provider and can be retrieved by other components at any time.

Selector View

The next component is the Selector View. It is an Eclipse view window and is derived from the Java class *ViewPart*. It gathers the information about the analyses from the Analysis Plugin Provider and presents a list of all models currently included in the tool. They are visualized as an icon representing the model together with the model name. The icon can display a formula, a symbol, a pictogram or any associated bitmap. Both the icon and the name are provided by the creator of the model.

Workflow View

The next part is the Workflow View. It is also a visual view window, and it displays the workflow steps of the model analysis. These steps guide the user throughout the data generation for the parameters of the selected model. We can think of different pieces of available information on different layers within the company, or just of an install-wizard-like guidance. The different workflow layers are described in the configuration file which is associated with the analysis.

Generic Parameter Interface

Another component is the Generic Parameter Interface, which is used by default, if no specific parameter interface is implemented. This interface can be configured in the model analysis configuration file and it is fully flexible. In more consolidated versions of the tool it shall be possible to have several different types of parameters like, e.g., text strings, numbers and enumeration values, but also complete Excel table files. Although this parameter editor is generic and works for every model, we plan to implement specific editors for the different models to guide the user.

Matlab Connection

The next component is the Matlab Connection, which executes the Matlab analysis models. It is designed to support several types of parameters and several return values, including numbers, arrays of numbers for plots, and graphics among others. It is also possible to run compiled Matlab models, as well as remote control live Matlab sessions. Although only compiled models will be used in the finished product, running and debugging the models directly in Matlab aid in the development and testing of the models.
The connection details as, e.g., the name of the compiled Matlab file, the function name and the parameter association are obtained from the Analysis Plugin Loader. The Matlab execution starts with the aid of the Matlab Runtime Environment.

Result View

The next component is the Result View, which embeds at least one diagram plotter to visualize the results of the Matlab execution. The necessary data are gathered from the Matlab Connection component. An enhancement of this result view is currently under consideration, and will be developed, if necessary, to display a whole report connected with different diagrams. The received data can be displayed in several predefined charts. The user can switch among these charts defined in a configuration file, containing different chart configurations of various types as, for example, bar charts, pie charts, etc.

Visual Editors and AID

The last component discussed here are the Visual Editors. We shall build visual editors (also called designers) if they are helpful for our partners from the model building work packages. As an illustrative example, a relevant designer is the Adversarial Influence Diagram (AID) Editor. Influence Diagrams are a good and intuitive representation of situations of systems with one or multiple actors. They are often used in game theory but also in several other domains. This designer enables the user to design such Influence Diagrams and also fill in the parameters directly in the diagram.

2.3.2 The structure of an analysis model

The framework has a directory where the models can be dropped in. The tool then finds all Matlab models located in all subdirectories of the “Models” directory. Therefore, the physical allocation of models can be structured in arbitrary directories, sorting them by any suitable criteria. The relevant files for a model are (1) the configuration file which contains general information like its name as well as specific configuration, and (2) the diagram configuration file, which contains the presentable diagram types and their visualization configurations. Besides, a model has an associated icon which is displayed in connection with its name in the Selector View. This icon can be any image, like a formula, a logo or a small pictogram. The last file, which is the most important one, is the compiled Matlab file, which contains all the necessary logic for the model.

2.4 User process diagram and Description

Now we show how the different components of the tool framework are cooperating together and interacting with the user. This description proceeds step by step throughout the whole process, which is depicted in Figure 3.

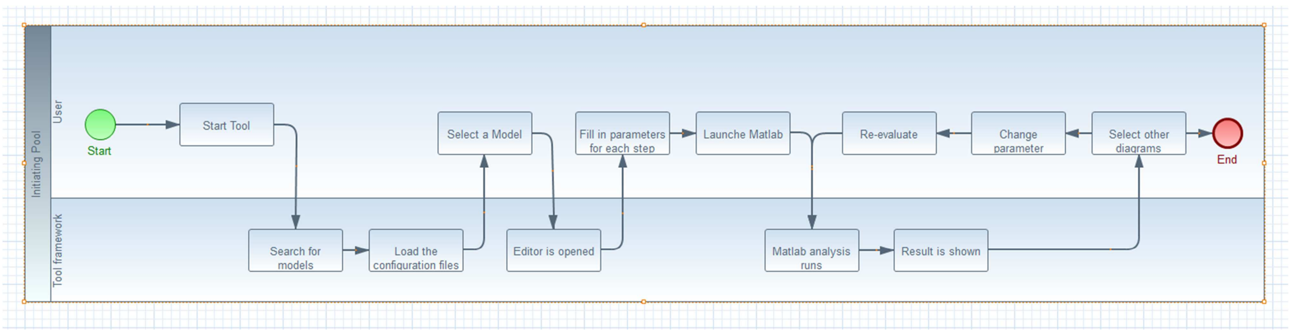


Figure 3 Tool and User interaction process diagram

Start Tool

The tool box is started by launching the main executable, which is currently called “eclipse.exe”, although we might rename it while switching to the Eclipse Rich Platform Client.

Search for Models and load them

After starting the toolbox all Matlab models will be loaded by the “Analysis Plugin Provider” component. In the tool directory there is a “Models” subdirectory where all Matlab models are stored in. The tool then finds all models located in all subdirectories so they can be ordered in any physical structure of subdirectories. The configuration file is parsed and all associated files, like the icon and Matlab file, are loaded. The retrieved information is stored in this component and can be then accessed by other components.

Select a Model

After the loading is finished, something which is done in a neglible time, the main tool window is displayed. Figure 4 shows the main window of the tool framework with all possible visible components named. On the left side the formula selector window is opened. It displays all models retrieved from the Analysis Plugin Provider, represented by their icons and names. While this example case uses as icons, other graphical icons can be used, like the use case or the situation displayed as an influence diagram.

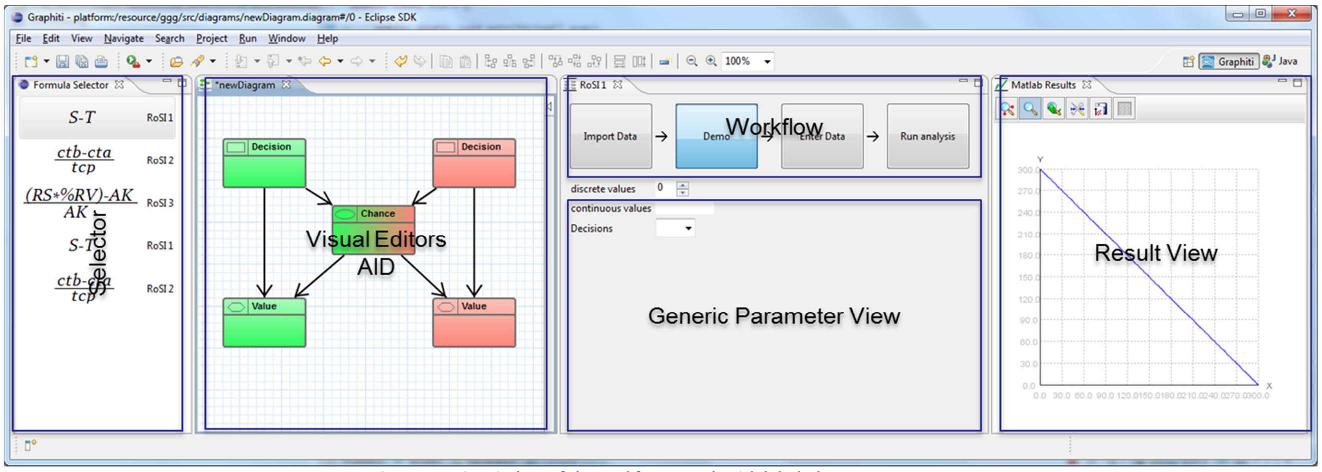


Figure 4 Main window of the tool framework with labeled components

Fill in parameters for each step

After selecting an analysis model in the selector, a new analysis file is created and opened in the "Workflow and Parameter View". This view displays the workflow of the associated model and shows the parameter editor below. The workflow is loaded in the Analysis Plugin Provider.

The Workflow is followed step by step. On each step, a Parameter View is shown. In most cases it will be the Generic Parameter View which is customized according to the model in the model configuration file. But for a given model specific Parameter Views can be also created. In any case, all necessary parameters have to be filled in to get to the next step.

Launch Matlab

The last step in the workflow is the "Run analysis" step. Here the Matlab analysis file will be launched with the aid of the Matlab Connection component. This is accomplished by the compiled Matlab code and the Matlab Runtime Environment. After the evaluation of the analysis, the results are returned to the toolbox.

Result is shown

The returned results are shown in the Result View. It contains a diagram rendering engine to display the desired plots. The user can perform usual customizing tasks as, e.g., zooming in, changing axis and recoloring, if allowed.

Select other diagrams

There can be more than one diagram associated with a specific group of results. All these diagrams are also described in the model configuration file, and can be created by selecting them in the tool. Then the diagram is shown and it can be configured with a special editor for it. Afterwards the configuration can be saved to a diagram configuration file. This file can be linked to the analysis configuration file. Then the configured, diagram will be available for future analyses of the same kind. It can also be linked to other analyses.

Change parameter and reevaluate

After a calculation is finished and the results are reviewed, the user might vary some parameters slightly and reevaluate the model.

Save and load current progress

To aid with this continuous reevaluation and reviewing process, it is possible to save the parameters as well as the last results obtained into the analysis file at any time. In this way, instead of starting a new analysis, we can load the previously saved analysis.

2.5 Model integration process

We show now how new analyses can be integrated within the framework. This section is a user guide through the whole integration process which is shown in Figure 5.

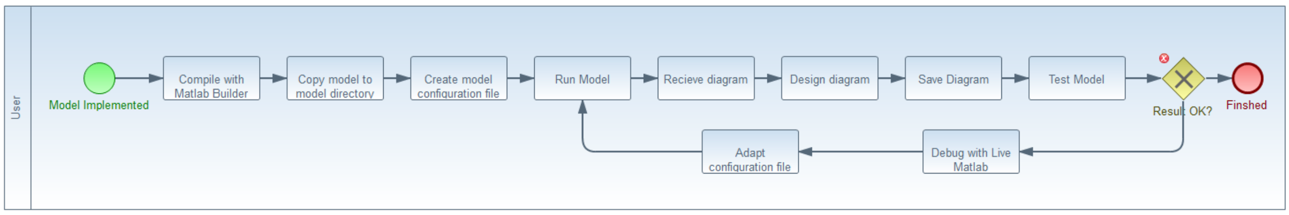


Figure 5 New model integration process

Model created

In the modeling work packages the main work is done and the formulas describing the models are written and specified.

Compiled with Matlab Builder JA

Then, the formulas describing the models are implemented in Matlab, being subsequently compiled and exported with the aid of the Matlab Builder JA [7] into Java compatible files. The specific way in which the execution of these models works can be consulted in Section 3.4.

Matlab model directory

The Matlab model file is copied into the model directory of the tool framework. See Section 2.7 for a detailed description of the directory structure.

Model configuration file

The next step is to create a model configuration file. For that purpose, we need an image as icon for the model. It can be the model formula, pictogram or any other object which can be associated with the model. The structure of the model configuration file is described in detail in Section 3.10.

Diagrams configuration file

Following, we create some default diagrams for the tool without any specific configuration. We run the model in the tool, obtaining raw diagram as result, which can be customized according to our wishes with the help of the built-in editor. Afterwards, we can export the diagram configuration to a file and link it to the model configuration file.

Testing

If something does not work as expected along the previous steps, we can use the Matlab Connection live link. It runs an installed Matlab instance and uses it as an execution platform. We can display what and how is being executed, observing the results and debugging the code within Matlab.

2.6 Parameter workflow

Within the tool framework, a workflow is implemented for the parameter interface. It enables the user to sort the different parameters and organize them together into different workflow steps. In a sense, it is comparable to well-known installation wizards. To complete the parameter form, the user starts with the first workflow step, e.g. entering statistical system data to calculate the system configuration, and will be guided step by step. Although the workflow presents steps to be processed in a sequential way, many parameters should be optional, so that the proposed order is not a strict one. Only those fields which are really necessary to go ahead with the next step in the ordered list should be a mandatory requirement.

The advantage of the workflow is that the user is guided throughout the parameters, with help to each parameter. The parameters are well structured into groups (workflow steps). Often, the information concerning the parameters is only available on different organizational layers of the company. Then, an employee involved in the project can fill in his part and pass the next workflow step to the next employee in charge.

This workflow is built generically and can be reused for almost all models. It can be configured within the configuration file.

2.7 Directory overview

In this section, we explain the file and directory structure of the tool. As aforementioned, the tool is based on the Eclipse Framework. It has a given directory structure which has been extended in this work. This structure may change if we eventually move to Eclipse Rich Client Platform (RCP). Table 1 shows the relevant directories within the Eclipse Framework.

Directory	Content
Eclipse/Plugins/	All plugins of Eclipse are lying here. The plugins of our tool framework are placed here, too. It is named eu.seconomics.tool
Eclipse/Workspace/	The user specific data are placed here.
Eclipse/Models/	We added a special directory for the models.

Table 1 Directory structure of the tool

2.8 Overview of code structure

The source code of the tool will be distributed together with the tool. It can be found in the directory *src*. The tool is internally separated into different components with their own subdirectories. Table 2 illustrates the path names for each component.

Component	Directory
Selector	src/eu/seconomics/tool/selector
Analyses Plugin Loader	src/eu/seconomics/tool/analyses
Internal XML Engine	src/eu/seconomics/tool/xml
Generic Parameter Interface	src/eu/seconomics/tool/parameter
Matlab Execution	src/eu/seconomics/tool/execution
Result View	src/eu/seconomics/tool/results
Adversarial Influence Diagram Editor (AID)	src/eu/seconomics/tool/aid
Model for AID Editor	src/eu/seconomics/tool/aid/model

Table 2 Code structure components and their associated directory

For the SECONOMICS tool framework the Model View Controller principle is used. It is a usual software construction pattern intended to separate internal logical parts from user interface parts. Following this pattern, each component with a user interface has a class *Control* which contains the logical structure behind the interface, and a class *View* which displays the view the user interacts with.

The *Selector* directory contains two classes of the program. The class *SelectorView* contains the user interface displaying the table of analyses that can be performed. The class *SelectorControl* contains the logical structure behind the view and reacts upon user interaction.

In the *Analyses* directory there are several classes which are relevant for the organization and integration of the different analysis models. The class *ReadConfigFile* imports analysis configurations from the file system. The classes *Analysis*, *Parameter* and *WorkflowStep* model the actual internal analyses. An analysis consists of several workflow steps, each of which contains, in turn, several parameters. Finally, the class *EditorViewObject* represents the gateway to our Generic Parameter Interface.

The XML directory contains two classes, the *ReadXMLFile* class and the *WriteXMLFile* class, to deal with the input and output of XML based files, which are used at several instances of our tool.

The *Parameter* directory includes all classes that are necessary for the Generic Parameter Interface. The class *ParameterEditor* represents the view the user interacts with. There is also a corresponding control class, named *ParameterEditorControl*, and a class which organizes the data processed within the editor, named *ParameterEditorInput*. The class *CallEditor* is an execution handler for triggering a start of the editor.

The *Execution* directory contains the class *MatlabExecution* in which the operations for performing an analysis in Matlab are organized.

In the *Results* component several classes that manage the handling of the results obtained from the analysis execution in Matlab are organized. The class *ResultView* is used for displaying the results to the user. The class *ResultControl* processes the raw data that Matlab returns to our tool, and builds real Java data types we can operate with. The classes *BuildJFreeData* and *ChartBuilder* make use of our graphical engine JFreeChart to create the visualization of the results. Several changes can be performed on the layout of the results diagrams, and this is accomplished by the means of the *DiagramConfiguration* class.

Finally, the *AID* directory contains certain subdirectories which are necessary for the organization of the Adversarial Influence Diagram editor and the underlying meta-model. In the *Model* directory the abstract meta-model underlying an editor instance is contained. The other parts of the AID directory contain classes which are required for the realization of specific diagram instances, as well as for all the features that our AID editor may provide.

3 Design of the Components

After the presentation of the design of the whole framework and the general interaction between the components in Chapter 2, we will explain now in detail the design of the different components. Figure 1 shows the general framework structure and those components that are working cooperatively within it. Each of these components is described in detail in the following sections, starting with the more general components that handle the background connections, moving on to the more specific sections and user interface parts.

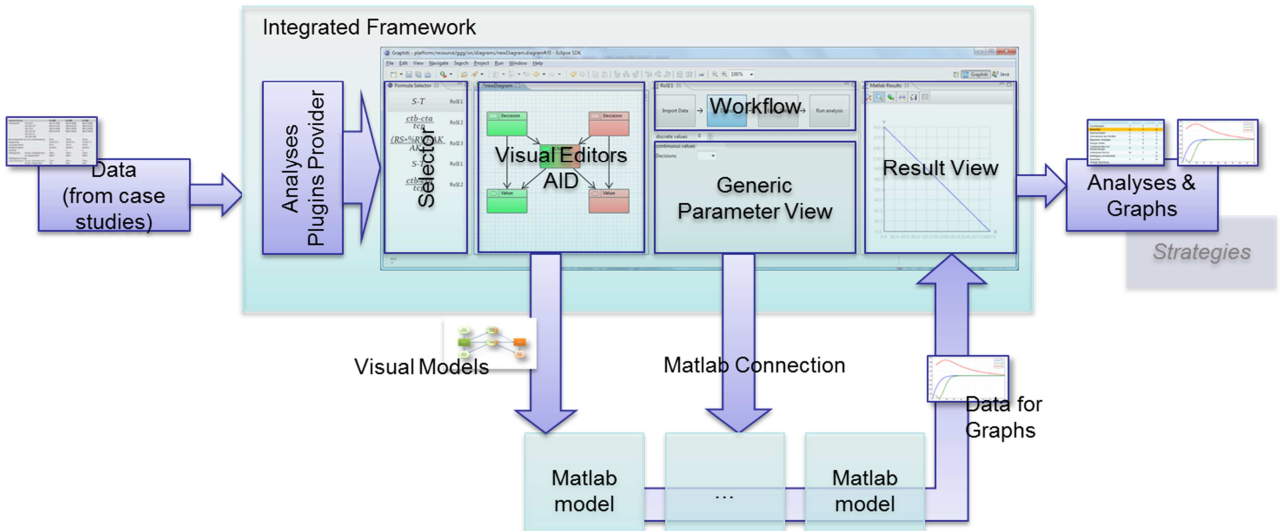


Figure 6 Structure of the whole tool framework

3.1 Analyses Plugin Provider (APP)

The Analyses Plugin Provider is one of the most critical components of our tool. It is responsible for the background loading and providing analyses that should be performed by our tool.

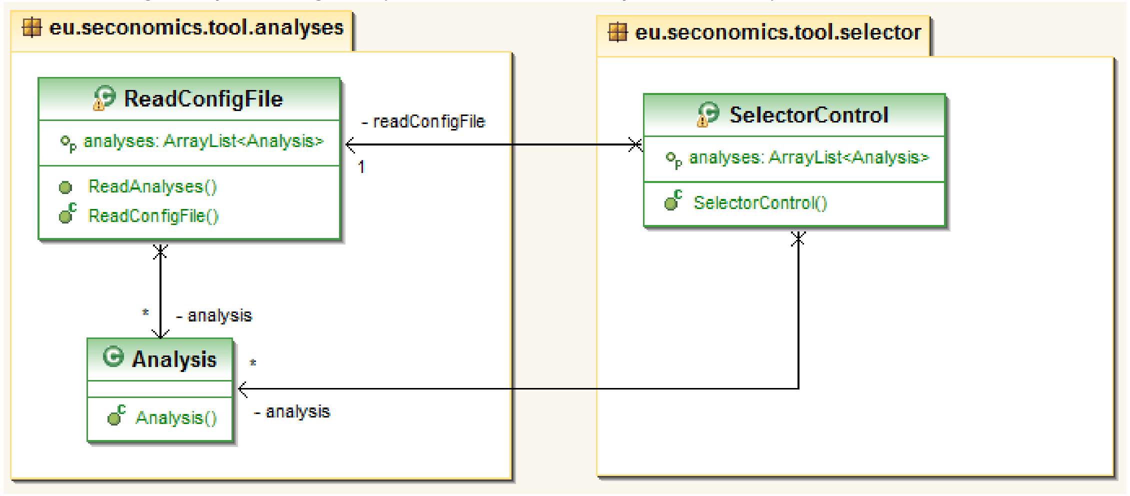


Figure 7 UML class diagram of the Analyses Plugin Provider

As the first step, the APP loads all Analysis Configuration Files in the target file system which, in most cases, will be a specific directory. Therefore, the file system is searched for our XML based configuration files, which have the file extension **.SAC*, standing for Seconomics Analysis Configuration. For each loading process (most likely only once when the tool is started), we create an instance of the internal Java Class *ReadConfigFile* from the class *SelectorControl* in the *Selector* package. Therein, a private method call for *ReadAnalyses()* is processed. By the means of the JDOM package from the Apache open source project [11], the XML structure of the file is parsed to a Java object style. After that process, the data stored in the XML elements of the file has been converted into Java Strings or the numeric Integer values, and a list of Workflow Steps and Parameters is also converted to the required format. With these previous requisites, we can initialize a newly created Analysis object, which can be provided to the internal list of analyses. Public access to this list for other classes of our tool, especially for the Analysis Selector, is provided by the public method *getAnalyses()*. For a better understanding, the structure described in this section is sketched in an UML class diagram in Figure 7.

3.2 Selector

The Selector is the starting interface of our tool that the user starts with. It offers the possibility of selecting among different formulas and models, which correspond to the analyses that can be performed. The Selector UI is based on a Table Viewer of Eclipse's Standard Widget Toolkit, the Java SWT library. An analysis is visualized by a name and an image, displayed as a table.
According to the *Model View Controller* principle, the Selector consists of the two Java classes *SelectorControl* and *SelectorView*, whereas the *SelectorControl* class represents the interface to the model, especially to the analyses provided by APP. All the analyses displayed in the Selector are initialized by APP, as it has already been described in section 3.1. While the loading process is triggered automatically when the tool is started, it can also be run manually from a dialog, where an analysis file or a directory has to be chosen. This is convenient as also external analyses are supposed to be performed.
Additionally, the SelectorControl can also be seen as a central unit of our tool, because the coordination needed for opening a result view after an analysis has been performed also takes place inside this class, specifically by the method call *openResultView()*.

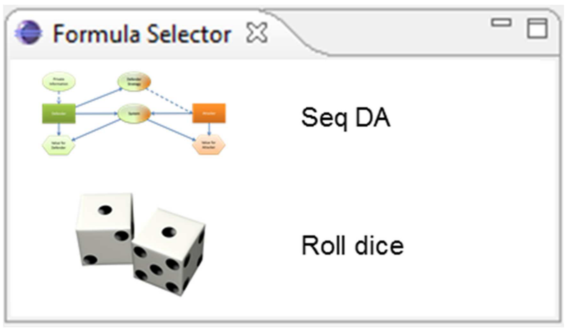


Figure 8 Selector View enables the user to create a new instance of an analysis

All the actions performed within the Selector are realized with Java *Actions*. These Actions can be seen as commands, which are executed when the user performs a specific action. The action *loadPath* at the initialization of the Selector triggers the APP to import analyses from a certain file path. A double click on an analysis triggers the Action *openParameterEditor* which results in a new view part, where the parameters for an analysis can now be entered. Finally, the Action *showResults* is triggered after an analysis has been performed successfully, and, as a consequence, the method call *openResultView()* is performed. An example Selector with two demo analyses is shown in Figure 8.

3.3 Adversarial Influence Diagram Editor (AID)

The Adversarial Influence Diagram (AID) Editor is an editor, based on Eclipse's Graphiti Framework [8]. The Graphiti Framework is used to easily build graphical editors. The AID models adversarial influence diagrams to simulate sequential Defend-Attack-models with two intelligent adversaries involved. The main idea of this editor is inspired by the GeNIe editor developed at Pittsburgh University [4], but our editor is more specialized and we have extended its functionality so that we can cover up all of our requirements some of which are not fulfilled by GeNIe.
Diagrams created with AID mainly consist of three types of node elements: *Chance*, *Decision* and *Value*. Chance elements represent statistic variables with different states and their corresponding probabilities. The sum of the values of all state probabilities in one chance node must be equal to 1. A decision node represents a set of possible choices an actor can make. In a value node, the outcome of the system, which is dependent on the selected choices and the outcome of the chance nodes, is computed. Besides, the settings (state / choice) corresponding to an optimized outcome can be computed. This can either be done as a total outcome for all participating actors, or by maximizing the expected utility for each actor

individually. This is also one of the main reasons why we cannot use GeNIe editor, as it can only compute the optimized total outcome for the whole system, respectively for all involved actors together.

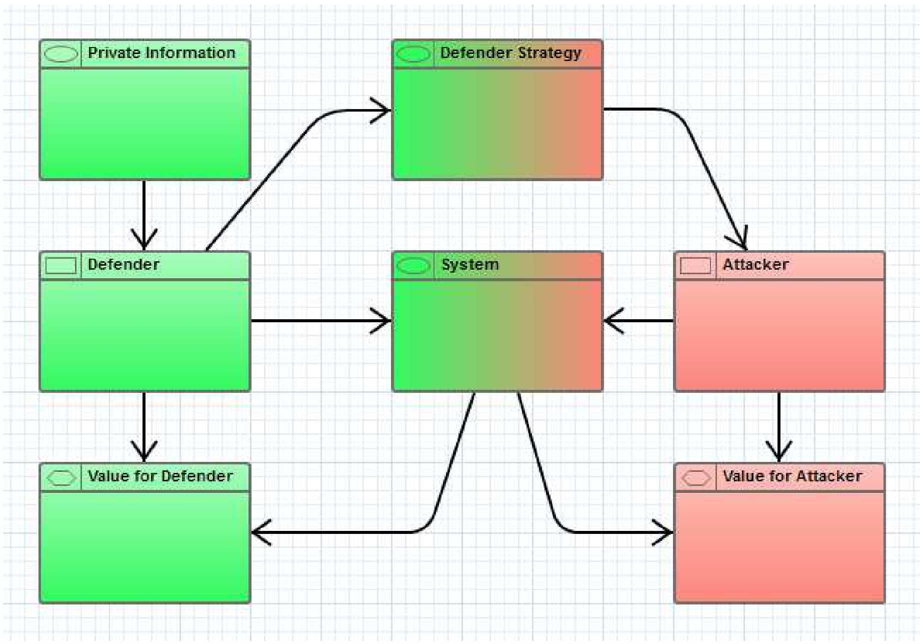


Figure 9 An example Defense-Attack Model which demonstrates the prototype of the designer

To model the accountability areas for the different participating actors, it is possible to assign each node to an actor; this would mainly be attacker or defender. When a node is relevant for both actors, this can also be modeled. The visualization of these associations is handled by different colors. Defender nodes are usually colored green, an attacker node is colored red. If a node is part of both accountability areas, this is visualized by a color gradient.
The second aspects of the influence diagram are the references between the nodes. These references are used to model the actual influences. There can be references between chances and decisions in both directions. Additionally there can be references from chances and decisions to value nodes to model which chances' states and which decisions' choices influence the outcomes in which values. An example diagram modeled with AID is shown in Figure 9.

The different node types are indicated by corresponding icons in the top left corner of the node. Chance nodes are visualized with an ellipsis, decisions with a rectangle and value nodes are represented by a hexagon. The diagram in Figure 9 models a situation which can be described as follows: The defender's decision is based on his private information and defines his strategy. The attacker's decision depends, to a certain extent, on the Defender's strategy. The state of the whole system is affected by both decisions, although there is a random component here, accounting for the fat that there is uncertainty about the outcome of an eventual attack performed by the Attacker. Finally, the value for each actor is computed from the system state and also from the corresponding actor's decision.
The implementation of AID has been accomplished as follows: The editor contains an underlying instance of our meta-model generated with the Eclipse Modeling Framework. In this meta-model, all elements a specific model may consist of are modeled as Java classes. In order to create the meta-models, we have used Eclipse's *Ecore Diagram Builder*. It is used to create meta-models. An overview of the whole meta-model is given in Figure 10.

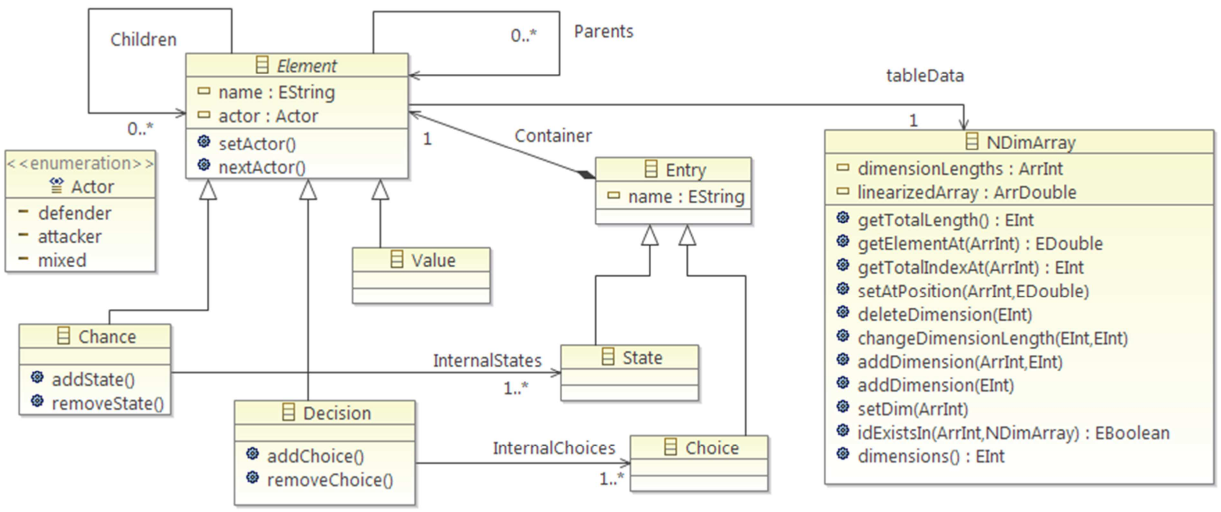


Figure 10 Meta-Model for the AID Editor created with Ecore

The central component of our meta-model is the abstract class *Element*, which must have a name of type String, and an *Actor*, which has to be always one of the three types *Defender*, *Attacker* or *Mixed*, where the latter represents represents that such Element is part of the accountability areas of both participating actors, *Defender* and *Attacker*.
The *Element* class has three subclasses, namely *Chance*, *Decision* and *Value*, as it has been already described in this section.
The internal states of a *Chance* element are implemented through the class *State*. In a similar manner, the internal choices of a *Decision* element are implemented by the class *Choice*. Both classes, *State* and *Choice* are subclasses of the *Entry* class which, in turn, is contained within the abstract super class *Element*. The connections between the different elements in an actual model are represented by the associations *Parents* and *Children* in the meta-model. Both associations realize internal lists of referenced items in the actual element. The notation *0…** at the association means that the number of referenced elements is arbitrary.
When a new reference is created in a specific model, the two referenced Elements are added to the corresponding *Parents* and *Children* lists. This is done by the means of the *addChoice()* and *addState()* method of the specific element.
Finally, the class *NDimArray* is used to store the internal values of an Element for each combination of States and Choices of all parent elements. The functionality of this class will be described in detail in Section 3.9.
The functionality of the editor itself is accomplished by means of the implementation of the so-called *features*. This is done with respect to the fundamental concept of the Graphiti Framework we made use of when we built the editor. Each of these features is realized as a particular Java class. For example, there are classes for creating the elements in a diagram, *CreateChanceFeature*, *CreateDecisionFeature* and *CreateValueFeature*. All functionality that the editor shall provide has to be accomplished through the implementation of additional features. All features have to be registered within the *DiagramFeatureProvider* class, which can thus be seen as the controller for the whole functionality of the editor.

3.4 Matlab Connection

In several cases, we have to perform highly demanding computational analyses. Therefore we shall make use of Matlab because of its power and excellent performance as a mathematical engine. As our tool is developed within a Java environment, we have to provide a connection between Java and Matlab. This is done with the aid of the Matlab Builder JA package. With this tool, functions implemented in Matlab code can be deployed as methods of Java classes into automatically generated Java libraries and be exported subsequently as JAR files. The whole process is visualized schematically in Figure 11. In this way, we can integrate and call Matlab functions from Java. The simple call of Matlab functions from Java is offered by native Matlab Java libraries, but we also offer the functionality to embed Matlab functions at Java runtime. This is done by the means of the Java reflection library. All we need to integrate a Matlab function at Java runtime is the name of the analysis name, a class name and the name of the Matlab func-function, so no extra data than that required for deploying a function from Matlab is actually needed.

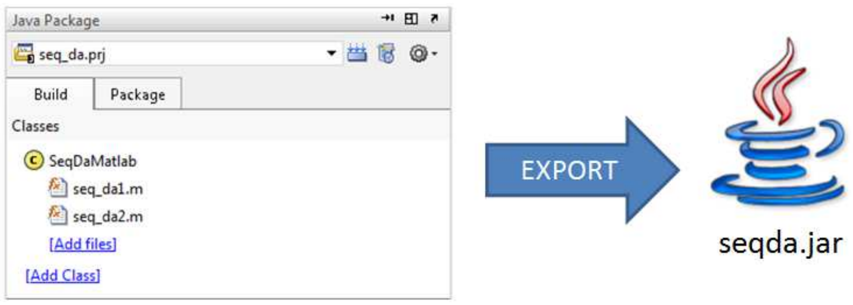


Figure 11 With the Matlab Builder JA a Java library is created

If desired, the native graphical engine of Matlab can be used for graphical visualizations, but this must be implemented in the Matlab code. For further information, see [5].

The implementation of the Matlab Connection is accomplished in one single Java class, called *MatlabExecution*. The method *performAnalysis()* is executed, once all necessary data for an analysis have been entered and the command for an execution is given by the user. The corresponding Matlab data (class, directory, etc.) are then loaded, based on the information given in the configuration file (see Section 3.10) for the analysis. To this aim, we make use of Java Reflection, which contains several libraries and classes for the integration of additional classes at Java Runtime.
For the actual Matlab function call, we need to convert all input data for the analysis to one single Java Object Array. The function is called with that array as an input afterwards, so that an Object Array will be returned. This has to be converted back to ordinary Java data types.
As already mentioned before, it should also be possible to integrate a user's own Matlab functions and run them within our tool. However, these functions do not work properly right from the beginning, especially when they are newly created. Therefore, we can start a Matlab Live Session from our tool and can execute the function in Matlab's debug mode. If an error is found, the function code can be debugged and it must then be deployed and tested again. It is essential that an official Matlab distribution is installed on the system to make use of this functionality. Detailed information on controlling a Matlab live session can be found in [5].

3.5 Result View / Graph / Visualization

There are several possibilities to perform graphical visualizations of analysis results. As mentioned in Section 3.5, we can use the native graphical engine of Matlab, but this has to be implemented within the Matlab function. In that case, graphical results would be shown in an external Matlab window. Keeping in mind that we want to develop an integrated toolbox, we have also provided a graphical visualization feature within our tool.

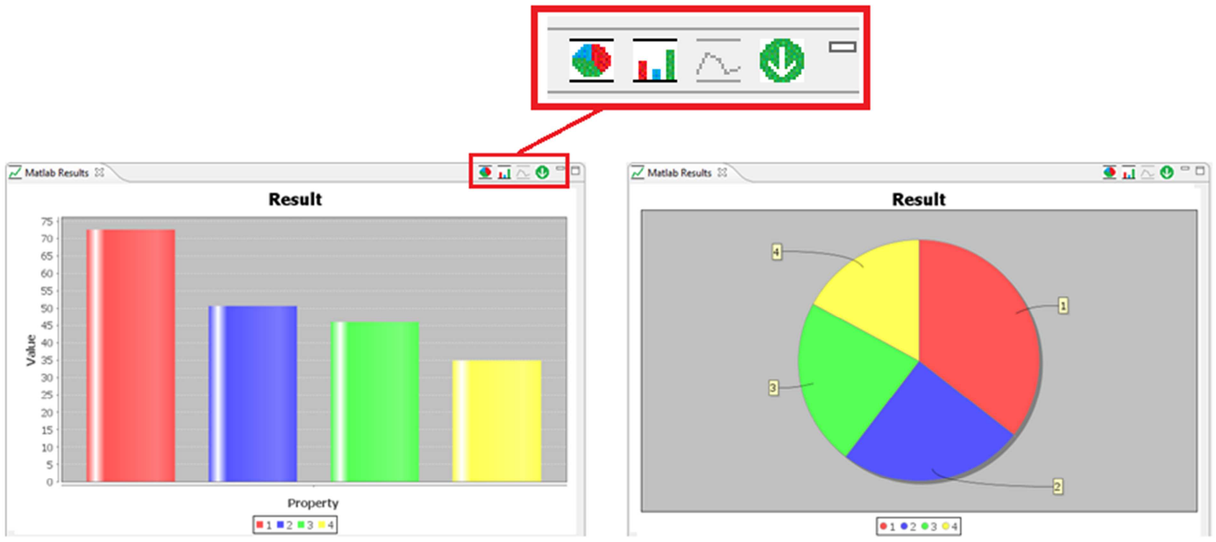


Figure 12 Result View implemented with aid of JFreeChart

Therefore we have made use of JFreeChart [10], which is a wide range API for visualizing graphical results of almost any type. As we will mostly handle results in terms of statistical background, we will have to use, for instance, pie and/or bar charts to visualize probability distributions. It should be also possible to display density functions by means of two dimensional line charts. An important remark about the limitations of JFreeChart should be done here. At this level it is not possible to visualize three dimensional graphs with this framework. This can only be done by using a two dimensional plot with the aid of different color or brightness levels, such as a grayscale, for instance. But for us this is not really relevant for the purposes of this project, as all the applications we provide should work well without the need of using three dimensional graph plots.
A result visualized with JFreeChart is completely integrated into the toolbox; it can be handled like any other module of the tool, and so for example docked to any position the user desires.

Figure 12 shows two types of graphical results of a sequential Defend-Attack analysis obtained with JFreeChart. The result corresponds to a discrete probability distribution, displayed both as bar and a pie chart. In the top right corner of the result view, here highlighted in the red box, the different available chart types for the result are shown. As the result displayed is a discrete probability distribution, bar chart and pie chart can be used. The third option – line chart – cannot be used in this case, so it is shown in grey, indicating that it is not available for this type of results. The green button on the right saves the result view settings. This saving feature is useful, as it is possible to change several of these settings. Therefore JFreeChart offers a menu where various options can be set. The number of settings the user can modify can be, dependent of the actual chart type, quite large. So the saving feature is really necessary when a completely individual chart layout has been created. The configuration is saved in an XML based file structure. For each analysis, there is a file that contains all created diagram configurations. Each configuration contains all parameters for all chart settings. When there is more than one configuration saved for one analysis, the user is asked to choose one of these configurations when a result should be shown after an analysis has been performed. If there are not any configurations for the current analysis saved, the chart is displayed with default settings, as the example result in Figure 12.

3.6 BPMN

Business Process Model and Notation (BPMN) [12] is a graphical notation for the visualization of business processes as flows, basically by means of the *Task*, *Association* and *Gateway* elements. Different accountabilities in the process can be modeled with different lanes, several lanes add up to a pool. At Fraunhofer ISST we developed an extension for this notation to be able to perform risk analyses [6]. Therefore, several new elements have been added to the notation, such as *Asset*, *Risk*, and *Mitigation*.
Assets represent valuable goods that are placed in specific parts of the business process and might need to be protected. Risk elements represent risks that can occur on specific tasks in the process. Typically, the occurrence of a risk may cause damage and, of course, involves the corresponding costs. By means of

mitigations, we represent the possibilities of preventing the business process from specific risks, or at least to decrease the impact of such risks.
Especially mitigations can have certain characteristics and, in some cases, also restrictions. As mentioned before, mitigations can decrease the eventual damage that would be caused in case of occurrence of the risk. But, on the other hand, mitigations could also have negative influence on the occurrence of several risks. Imagine a certain mitigation that completely removes a given specific security vulnerability. The implementation of this mitigation might also force an attacker to focus his attacks on other weaknesses of the system. In the same way, mitigations can also influence each other's effectiveness to certain risks. In the worst case, several mitigations can even mutually exclude each other mutually.
In large processes this can become very confusing, so we have developed a feature to analyze the application of several combinations of mitigations to find the optimized configuration with a maximum outcome.

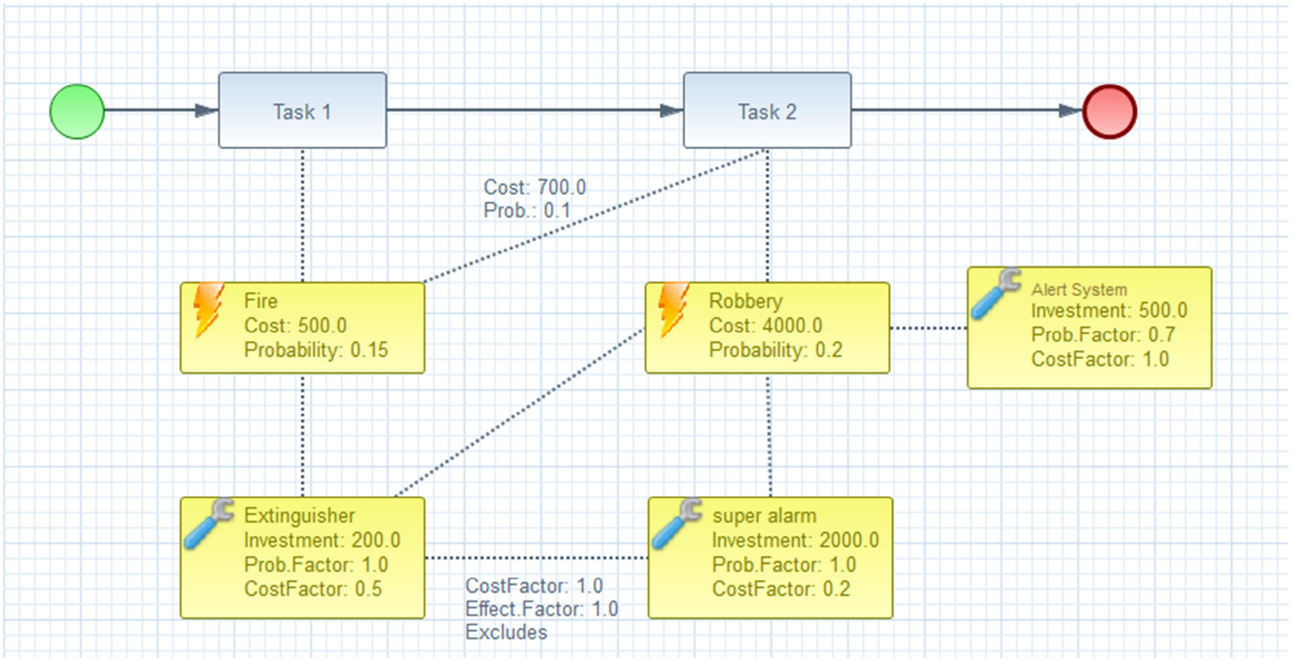


Figure 13 Example risk annotation to BPMN

An Example for an extended BPMN model is visualized in Figure 13. The simple business process consists of two sequential tasks *Task 1* and *Task 2*. There are also two risks, namely *Fire* and *Robbery*. *Fire* can impact both tasks, whereas *Robbery* can only impact *Task 2*. The parameters inside the risk elements represent default values the risk has for all tasks. Parameters on a reference (as from *Fire* to *Task 2*) overwrite the default values but only for the given reference. Finally, there are three mitigations to attenuate the risk impact. As for the risks, the values inside the element represent default values which could be overwritten at each reference between risk and mitigation. The parameters on a reference between two mitigations determine the impact among the mitigations. In the example, there is a dependency between the mitigations *Extinguisher* and *super alarm*. There is no impact on the Cost Factor or Effectiveness Factor, but a mutual exclusion instead. This means that only one of these mitigations can be implemented at the same time.
We have created an analysis prototype for this risk annotation. This analysis finds an optimal cost efficient combination of mitigations. This is done by testing all combinations and calculating the expected savings. To this aim, the analysis takes the reduced expected damage and the investment cost into account. The result is shown graphically by coloring in green the mitigations to be implemented and the deselected ones in gray.

We have implemented a prototype of a designer for these annotations. This designer is based on an available Eclipse plugin, which implements a designer for BPMN and is called BPMN2 [ref]. This editor is extended with a plugin and creates a prototype for a graphical representation to design business processes and annotate them with risks and mitigations. The result of the analysis is displayed within this designer.

3.7 Workflow / Abstraction

As the analyses can be very demanding from a computationally point of view, we have introduced an abstract workflow scheme, so that the different steps of the analysis can be kept apart from each other. So, each analysis may be decomposed into several steps, with each step representing an abstract self-contained part of the analysis. This scheme could either be used to represent different accountabilities, when several parties are involved in performing an analysis. It could also be possible to model the steps as time based, e.g. when pre-computations are necessary before the actual analysis can be performed. So there are a lot of possible applications that this feature could be useful for.

3.8 Parameter Editor

Each workflow step of an analysis is shown as a page in the parameter editor and consists of a set of parameters which have to be entered in that specific step. It is not always necessary to complete the steps in the given order, but it might be possible that a parameter in a step needs results from computations that are performed in previous steps. In this case, parameters must be flagged as required; in any other case, parameters should be set optional wherever possible.
For each parameter in the step, there is the name and a field where the value can be entered. The type of field to be displayed depends on the type of the parameter. For instance, for a String parameter it would be a common text field, but if the parameter type is a logical value, then there would be a Combo box (or radio buttons) in which all options will be listed.

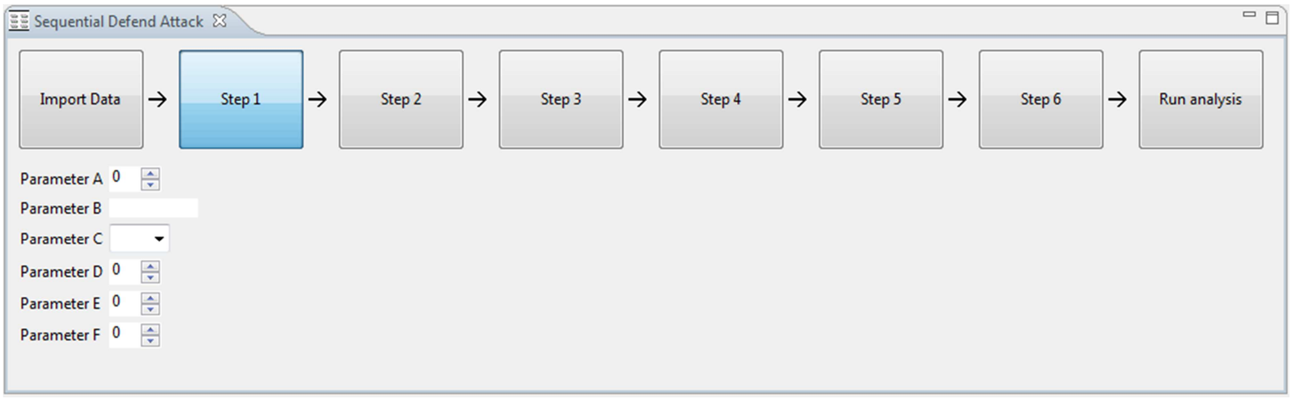


Figure 14 An Example of the Worflow and Generic Parameter Interface

Figure 14 shows an example Parameter Editor with a workflow for an analysis consisting of six steps. The first step is already in progress and needs six parameters to be entered. This view part is implemented to be completely dynamic, so that any number of workflow steps and parameters is possible. The only things that are fixed are the elements *Import Data* where pre-defined data for the analysis can be implemented from an external file and *Run analysis* which starts of the computation within the Matlab engine. The six parameters of the selected workflow steps are of different data types. This can be recognized in the field of the value. For example, A is a numerical value, B a String and C a logical value where all possible values can be selected from the box.

The data entered in the steps or the corresponding parameters can be stored using Eclipse's native saving functionality. This generates an XML based file containing information about the analysis and the values for all parameters of each workflow step. The file can be then used for a later reload, when the same analysis should be performed again without the need of starting it again from scratch.

3.9 N-Dimensional Array

An N-Dimensional Array (NDA) is an array-based data structure that can contain an arbitrary number of dimensions. We make use of this data structure within Influence Diagrams. By means of NDA the dependencies among the nodes in an Influence Diagram are modeled. Each node N within our diagrams

contains an individual NDA. Every predecessor $P_i(N)$ is represented by a dimension $D_i(N)$ in the NDA, whereas the size of each dimension depends on the number of inner states of the corresponding node $P_i(N)$. The initial dimension $D_0(N)$ of the NDA represents the node N itself.
This is necessary as we need to consider every possible combination of states of all preceding nodes to be able to set new values in the node of the NDA or to compute the outcome for any of these combinations.
To handle changes in the Influence Diagram, particularly adding or deleting references between nodes but also adding or deleting inner states in a node, we need several functionalities for the NDA. We can add, resize and delete dimensions for an optimized usage of data storage. A dimension is added to the NDA of node N, when there is a new reference from another node P to N. An NDA dimension $D_i(N)$ is deleted, when the reference between predecessor $P_i(N)$ and N does no longer exist. An NDA dimension $D_i(N)$ is resized, when the number of inner states of the corresponding predecessor $P_i(N)$ is changed.
NDA offers an API, where we can access every data by calling its coordinates in a multidimensional grid. The data itself is stored internal in one big array. From entered coordinates we can compute the actual position of the data in the internal array and vice versa. The internal array consists of the number of NDA dimensions, the length for each dimension and finally the data. This is shown in Figure 15.

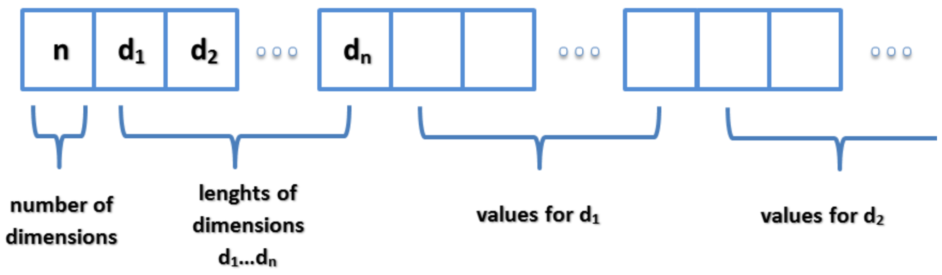


Figure 15 Transomation of the N-D Array in a lineare one-dimensional one.

3.10 Analysis configuration file

Analysis configuration files contain information about the analyses to be processed within the tool. In these files, the context of one or more analyses is defined and several properties have to be set. The file scheme is organized as an XML structure, whereas one file can contain one or more analyses. For clarity reasons, not all analyses should be composed in a single file, but rather several analyses handling the same or similar issues could be summarized in one file.
At the beginning of each analysis section, general information has to be defined. That is, for instance, the name of the analysis, the figure shown in the Selector, and some declarations for the corresponding Matlab function call – library name, class name and the actual Matlab function name. This general information is followed by the list of steps an analysis consists of. For each step, a name and a description must be set, as well as the parameters the step consists of. A parameter contains a name, a description and the type of the value to be entered.

```
<?xml version="1.0" encoding="UTF-8"?>
<analyses>
<analysis>
        <name>SeqDa</name>
        <icon>seqda.png</icon>
        <matlab>seqda</matlab>
        <class>Madrid</class>
        <call>sequential_defend_attack</call>
        <step>
                <name>Simulation</name>
                <desc>Enter the Simulation parameters here</desc>
                <param>
                        <name>runs</name>
                        <type>int</type>
                        <desc>Min 0, Max 3</desc>
                </param>
        </step>
        <view config="seqda">
                <type>bar</type>
                <type>pie</type>
        </view>
</analysis>
</analyses>
```

Figure 16 Example analysis configuration file to demonstrate the structure of it

Finally, for each analysis it has to be declared what type the output result is supposed to be. That can be different diagram types, such as pie chart, bar chart or line chart, as already described in section 3.5. At this point, it could also be possible to define that the result should be shown outside the integrated toolbox, using the native graphical engine of Matlab instead, as already mentioned in section 3.4.
The members of WP5 have already implemented an exemplarily sequential defend attack analysis. Therefore we have created an analysis configuration file, which can be found in the following, to get an idea, how such a file actually looks like. This analysis already works within the prototype of our tool framework. See Figure 16 to see an example for this file.

3.11 Diagram configuration file

As already mentioned in section 3.5, we use XML files also for saving user defined diagram configurations. As all the features of our graphical engine's settings menu should be mapped to our configuration file, the structure of the file is very similar to the structure of the menu, see Figure 17. An illustrative diagram configuration file can be found in the following code. This example does not show all possible features; rather it is just presented to get a general overview of the structure. The scope of the features also varies, depending on the diagram type. Pie charts, which are presented in the following example, are among those types with the minimum number of possibilities regarding the configuration settings; in this case, the parameters mostly represent only basic features.

```
<configurations>
<diagram type="pie", analysis="dice", configname="custom">
        <title show="true", text="Distribution", color="123">
                <style font="Tahoma", size="12", bold="false", italic="false"/>
        </title>
        <area color="123">
                <frame style="1", color="123"/>
        </area>
        <misc antialias="true", bgcolor="123"/>
</diagram>
</configurations>
```

Figure 17 Example diagram configuration file

For the implementation aspect, all the options can be accessed by Java methods. Entering new parameters for such a method results in a change of the corresponding option. When a configuration should be saved, the current values of all the options are memorized, converted to Strings and then exported with the help of the XML engine.

When one configuration is chosen, the chart configuration file of the analysis is scanned for the correct position. Therefore, the parameter *configname* is used. The corresponding parameters that have been saved are then imported to Java and applied to the chart settings.

4 Integration of Analyses

In this chapter we will describe how we integrate the different analysis models into our tool framework. First, we have a working example of the Economic and Policy models delivered by WP6 which will be described in Section **Errore. L'origine riferimento non è stata trovata.**. More background information on these models, especially regarding calibration approach, can be found in D8.3 [2]. In Section ▫ we will also show some challenges in implementing the ARA models of WP5 in Matlab.

4.1 Implementation for WP1

In the study of airport security, UNITN and DBL investigate a method that can determine the best alternative strategy for security instrument allocations. In particular, we propose models to be used to assess airport security directed at preventing terrorist attacks. This study provides a case study for various alternatives for specific security proposals. More specifically, we perform an analysis of implementing current and newly proposed security policies, exploring issues of technological cost and performance. As a result, we want to answer the following questions:

1. Given a current airport security policy, how does changing the current security policy alter the cost and/or effectiveness of the airport security? For example, is employing full body scanner or implementing a new training program cost-effective?
2. What are the tradeoffs between alternative security policies?

In each airport, all passengers/items are subject to several checks. They need to pass through various check-points where they are inspected by security personnel and devices (e.g., X-ray, metal detectors and hand search). For passengers' bags, for example, inspection occurs by passing them through a fixed X-ray scanner. Inspectors examine the scanned image for finding any signs of threat. After the inspection, the security personnel/device determines whether the passenger/item contains a threat or no threat. Each security personnel/device therefore produces an alarm or a clear (i.e., no alarm). In some cases, security personnel perform additional screening which consists of a two-stage inspection process. For suspicious passengers/items, for instance, security personnel conduct a hand-search which is a time and resourcing consuming process involving.

While careful and prudent settings of security instruments might be able to improve the level of detection of threats, errors in detection of threats cannot be circumvented perfectly. The goal of a passenger/item screening policy is therefore to reduce the risks from terrorist attacks. Any newly proposed security policy should be compared with a current security policy. A cost-benefit approach has been widely used to compare and access current and alternative security policies for passenger/item screening.

To conduct a cost-benefit analysis, we need to identify costs and benefits from current and alternative policies. Both costs and benefits have direct and indirect aspects. The direct benefits of the policy are linked with avoiding the damage of a terrorist attack, both to the airport and to the society in general. These benefits come from deterrent effect of screening policies and the potential to identify and remove a dangerous item before it is used. The indirect benefits of a policy are very difficult, if not impossible, to estimate; for instance X-ray scanning might allow inspectors to better identify unlawful materials such as drugs and smuggled goods. On the other hand, the direct costs commonly include the equipment and personnel related costs associated with a policy, while indirect costs being treated most importantly are the costs caused by delays and congestion. It should be noted that there is high uncertainty and difficulty in measuring parameters of the model for passenger screening. Nevertheless, cost-benefit analysis is a very helpful framework for providing a guideline of this decision context.
In order to conduct cost-benefit analysis, we need to investigate factors consisting of the basis of alternative security policies for screening. Before describing these alternative security policies, we present a simplified version of 'base security policy' for comparison.

- Base security policy for screening processes: The current policy mandates the scanning of 100 per cent of passengers/baggage via deployed screening machines. The costs for this security policy include amortized fixed equipment costs, annual operations and maintenance costs for the scanning equipment and salaries for the operators and inspectors.

To assess the viability of alternative policies, we consider the following scenarios.

- Alternative security policy 1-1 – 100 per cent inspection of passengers/baggage using current screening machines and   per cent additional inspection with a new security measure (e.g., 3D body scanner): This policy requires the scanning of 100 per cent of passenger/baggage with current technology.   per cent need to have additional inspection by a device with a new security technology. New devices and a team of operators and inspectors for these devices are required. In addition to the costs mentioned above, there are additional costs for purchase, operations, maintenance, inspection and delay.
- Alternative security policy 1-2 – 100 per cent inspection of passenger/baggage using current scanning machines with an additional training program: Due to improved training for inspectors, we assume that greater accuracy and a faster inspection rate is possible. While this case improves on 'current security policy' by reducing the false positive rate, it incurs the costs for the training program.
- Alternative security policy 1-3 – 100 per cent inspection of passengers/baggage using a current security technology and an additional inspection with a new security measure. An additional training program is implemented to operators and inspectors.

In addition, the following alternative security policies with centralized control will also be studied.

- Policy 2-1 (base security policy with centralized control) – 100 per cent inspection using current technology with a centralized control system: This policy requires the scanning of 100 per cent of arriving passenger/baggage, but with a remote monitoring system. While this policy is similar to the base security policy, there are some differences: inspectors are located in the monitoring center and the number of inspectors might be less than the number in 'base security policy'; monitoring devices in the center are required; additional infrastructure might needed; and errors in detection of threats and delay would be increased. Therefore, in the control center, the costs include fixed costs for monitoring devices, salaries for inspectors and fixed costs for infrastructure. At an airport, the costs consist of amortized fixed equipment costs, annual operations and maintenance costs for the scanning equipment and salaries for the operators.
- Policy 2-2 (alternative security policy 1-1 with centralized control) – 100 per cent inspection using current technology and 5 per cent additional inspection with new technology with a centralized control system: Similarly with policy 2-1, monitoring devices and inspectors for the current and new technology are located in the control center.
- Policy 2-3 (alternative security policy 1-2 with centralized control) – Centralized control with 100 per cent inspection using current technology with additional training program.
- Policy 2-4 (alternative security policy 1-3 with centralized control) – Centralized control with 100 per cent inspection of passengers/baggage using a current security technology and an additional inspection with a new security measure and an additional training program.

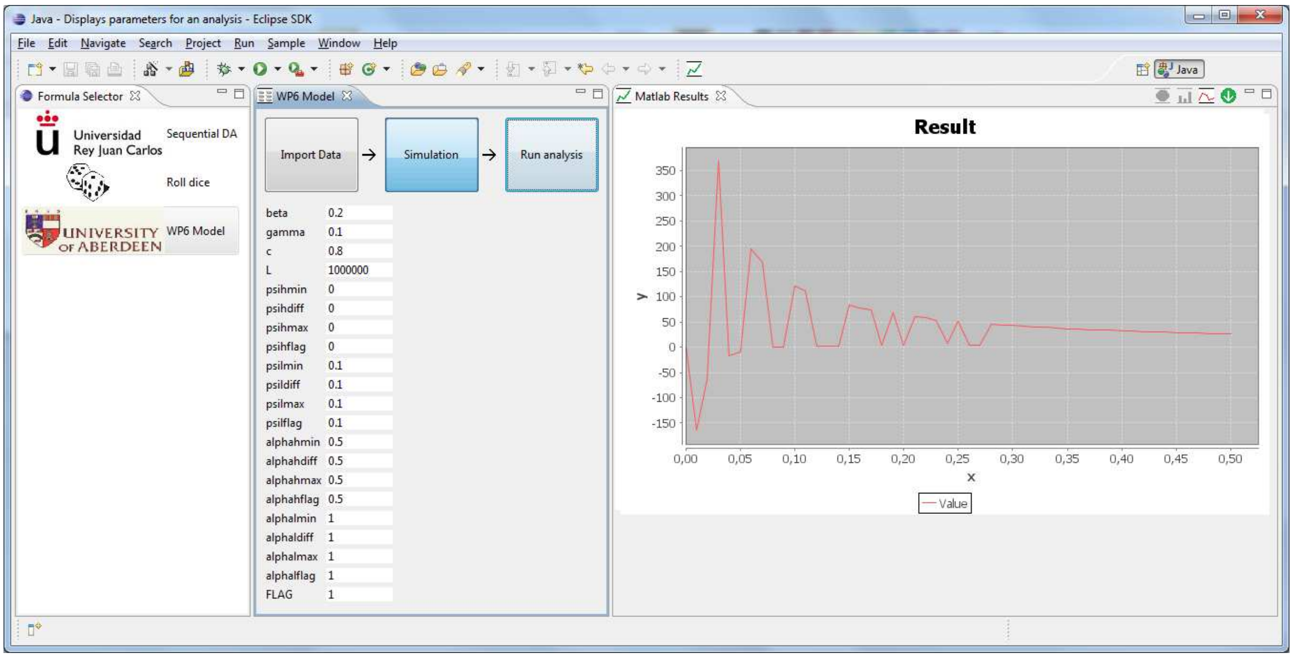
4.2 Implementation for WP5

While we were implementing the prototype for our tool framework, we also started implementing a first approach for the WP5 models including the metro case study. In this development process we have faced several challenges. By now, we have not yet found a reasonable solution for runtime and memory challenges. Using a powerful commercial mathematic engine reduces memory requirements and runtime duration effectively, but for a really feasible solution, we have to make further improvements.

The model simulates Defender-Attacker scenarios, as described in D5.1 of WP5. One of the basic ideas of these models is the use of distributions to take into account the uncertainty of the knowledge about the attacker preferences and utilities. To deal with probability distributions in practice, one needs to sample from these distributions. This means to use either big sampling matrices or to use many for-loops. Because one of Matlab's special features is the handling of intensive numerical computations, there is a big runtime advantage when using big matrices instead of loops. But on the other hand these matrices require a lot of memory, so that we ran out of memory in our tests. That is a trade-off situation. Besides mixing these two strategies another thing we are working on is to limit the complexity by reducing the search space properly. A final strategy has not been found yet; this is still a working topic.

4.3 Implementation for WP6

As a proof of concept for our tool framework, we have started with the integration of the Economic and Policy models from WP6 contributed by Aberdeen. To this aim, we have to deploy the Matlab functions that perform the model analyses to a Java package that we can access from the tool framework. This is done with the Matlab Builder, a tool that comes along with Matlab. We have to declare a Java class, which in this case has been named *Models*, and we have added the Matlab function that performs the actual economic model analysis, called *TestFunctionWrapper.m*, to this class. Later in Java, this Matlab function can be accessed as a method in the Java class. After the actual building process, we have a JAR library that contains all necessary information for the analysis execution. Furthermore, there are several class files containing the source code, but these files are encrypted and cannot be used directly. Besides, Java-doc standard documentation has been generated within the build process.



beta	0.2
gamma	0.1
c	0.8
L	1000000
psihmin	0
psihdiff	0
psihmax	0
psihflag	0
psilmin	0.1
psildiff	0.1
psilmax	0.1
psilflag	0.1
alphahmin	0.5
alphahdiff	0.5
alphahmax	0.5
alphahflag	0.5
alphalmin	1
alphaldiff	1
alphalmax	1
alphalflag	1
FLAG	1

Figure 18 Overview of tool framework with WP6 model integrated: Difference in investment in security for a firm operating with a regulated investment margin, as defensive effectiveness (abscissa values) increases. See Deliverable D8.3 section 4 for a worked summary of this type of model.

For performing the analyses through the Matlab functions, the deployed JAR library is the only element which is relevant. To be able to access the analyses from Java, we have to create an Analysis configuration and declare some parameters within it. We set the package name, the class name (Models), the name for the actual method call (TestingFunctionWrapper), and the parameters the method is called with. This information is imported when our tool starts. Figure 18 shows an execution of the WP6 model implemented in Matlab started from our tool.

The analysis can be started when the *Run Analysis* button is hit. This triggers an invocation of the operation with the use of Java Reflection. A class loader imports the class specified by its name in the configuration file from the package which is also denoted in that file. Then we create an object of the method that actually executes the Matlab function and perform the execution by calling the *invoke* method on that object.
Internally, this requires three parameters in total; the first one is the class object the Matlab function call method is part of, the second one is an object array for the input parameters of the Matlab function and finally the third one is also an object array that will contain the return values of the Matlab function after its execution. The actual call in the source code of our tool is performed as seen in Figure 19:

```
m.invoke(classObj, in, out);
```

Figure 19 Java code for execution of Matlab function

Therein,

- *m* is the method that starts the Matlab execution and that gets invoked by this piece of code,
- *invoke* is the command for the actual call or – namely - invocation,
- *classObj* is a variable representing the class *Models* which the method *m* is invoked on,
- *in* is the array with input parameters,
- *out* is the array for the return values.

In order to get feasible results for the analysis with the policy model, it is necessary to perform a several number of runs in Matlab. The organization of this has to be done inside Matlab, as we want to avoid shifting results from Matlab to Java and back again several times with regard to possible impreciseness. So for the underlying policy model, there is a wrapping function in Matlab, and we only call this one to perform the execution. The whole scheme of how we integrated the Economic and Policy models of WP6 into our tool is visualized in Figure 20.

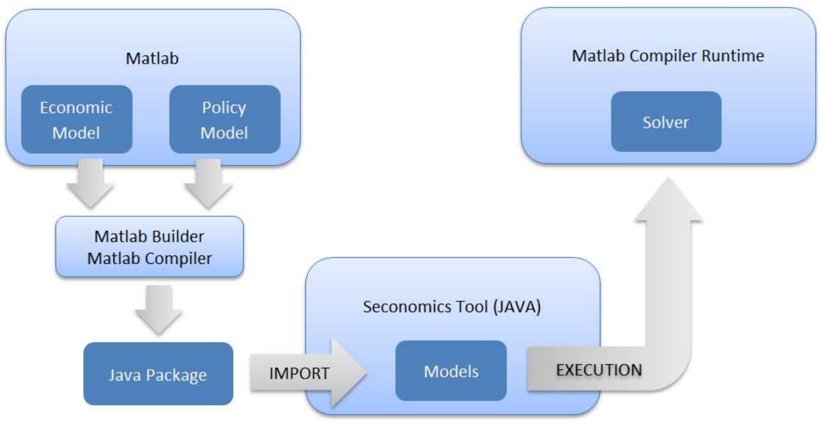


Figure 20 Scheme of model integration

The execution of a Matlab driven analysis takes is processed in the Matlab Compiler Runtime. Thus it is not necessary to own a full Matlab distribution. The Runtime can be downloaded freely from the MathWorks website. So any Matlab function that has been deployed to a Java package or as a standalone application by the Matlab Builder can be run on any machine that has a Matlab Compiler Runtime installed.
In the Runtime it is also possible to show graphical output in form of common Matlab plots, but the functionality has been constricted. Besides the fact, that this would be a separate window and we want to develop an integrated tool, this functionality limitation is also one of the reasons why we do not use the Matlab Compiler Runtime’s native plot window, but have employed JFreeChart as our graphical engine instead.

4.4 Implementation for WP8

Each case study requirement (D1.3, D2.3 & D3.3) demonstrate some need for human understandable representations of their system, e.g. of business processes, to better understand the mechanisms that generate security risks. This subsection outlines an implementation of modeling risk factors in complex systems. Specific modeling of the business process is sometimes useful in motivating the risk generating process (the outcome of the intersection of vulnerabilities, relative asset value and attack behaviour).

While developing a system, especially an IT system, different artifacts are created to support the system development and implementation. These artifacts include different kinds of UML diagrams, e.g. deployment, component or activity diagrams, as well as business process diagrams. They structure the system in a human understandable way and present different perspectives on the same system. Normally they provide a high level view onto the system, although very technical low level representations exist. They show things like the structure, the human interaction or the communications with other systems. With special extensions, e.g. UMLsec (See [21]), on these notations also security requirements can be annotated.

As violations on IT systems become more and more frequent, even accidently, all Systems have to pass a security risk analysis. In this analysis the security requirements will be defined and depending on these requirements the system will be searched for vulnerabilities and controls. Modern risk analysis standards like the NIST risk analysis defined in NIST 800-30 recommend using all available data, including specification diagrams. But how shall these artifacts be used? In most examples these diagrams are just a source for brainstorming. WP8 wants to research how these diagrams can be exploited more deeply. How can the structure be exploited? For this idea WP8 wants to develop a computer aided method based on NIST 800-30. This tool helps to refine the system definition and to find vulnerabilities and possible controls. These vulnerabilities might be found by using a vulnerability database like the NIST national vulnerability database. This database collects a great amount of vulnerabilities for many IT related components, especially software. It rates them with an index derived from a CVSS score that defines the impact and frequency of each threat.

The tool uses the common specification language and the related notation with additionally annotated vulnerabilities and risks. After annotating as many details as possible, the system shall support a cost evaluation with the return on security investment approach and recommend a control set for implementation. An example of this improved risk analysis might be applied to case study WP 2 on critical national infrastructure (see D8.3) for discussion.

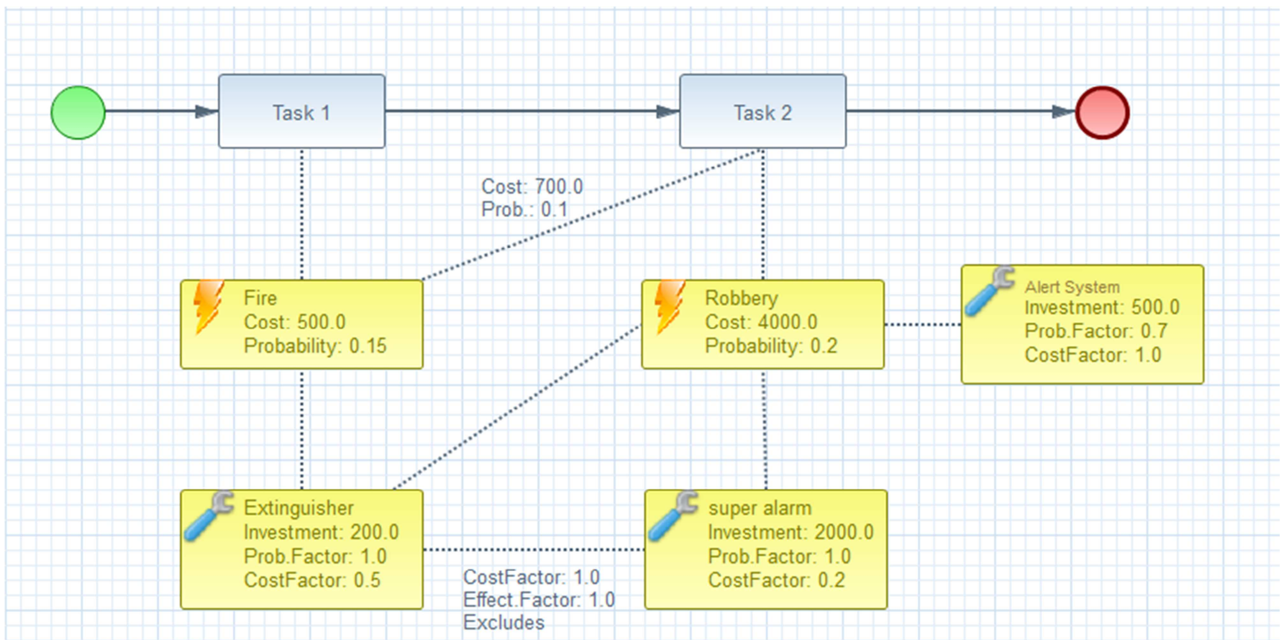


Figure 21: Extended BPMN process with risks and mitigations

Figure 21 shows an extended BPMN model, where the two sequential tasks, Task 1 and Task 2, are threatened by the Risks Fire and Robbery, which are influenced by the Mitigations Alert System, Extinguisher and Super Alarm. See D8.1 [1] section 4.3.6 for more details on the BPMN approach. Based on this first design WP8 will build a risk assessing tool, which will be implemented as a module in the SECONOMICS tool.

5 Summary and next steps

In this deliverable we explained how and why we have changed the perspective of the design of the toolbox. We have implemented a first prototype and sketched a user interface. The structure was first explained generally and, then, each component was described in detail. The internal structure of the tool, including some in-depth details of the code, has been also illustrated.

5.1 Next steps

We determine the next steps in the tool development progress. We have to make the prototype progress into the direction of a fully working tool. The tool has currently limitations, whose overcoming motivates some of the next steps we are currently undertaken.

First we want to focus on quality assurance, as the prototype is just a proof of concept. It has still some weaknesses which have to be delimited and corrected. Many ideas already explained are implemented just conceptually to test the functionality. These integration and implementation progress has to be continued.

Some concepts are explained in the design in this document but are not fully implemented. One of them is the Matlab live debugging. Our partners are interested in debugging the already integrated model. But in the prototype this is not included yet. We plan to build and integrate a Matlab live connection to support this functionality. This will aid in finding bugs in the communication between the user interface through all background processing to the Matlab implementation and back to the result views.

We have explained the Result View and have shown the nice functionality of the JFreeChart framework. So far, there was another framework integrated for that purpose, which has several weaknesses compared to JFreeChart. So we are going to replace the existing framework for JFreeChart. This is one of the main reasons the prototype is intended for, to detect weaknesses and build a proof of concept.

The possibility of including HTML reports was proposed by one of our partners. It is not clear until this stage, if (and how) it is possible to integrate data plots into these reports. We have several ideas, but they have to be tested in scope of the prototype.
The adversarial influence diagram editor shall get some functionality. Until now it is a markup. The input of data has to be possible, stored and transferred to Matlab.

We also have to work further on the implementation of the models of analysis work packages especially in the next step of WP5. We found some serious runtime problems and we have to solve them. Until now it is unclear how this solution might look like. Also other models, especially of work package 6, have to be included. The inclusion process is not fully working right now.

Another thing to do is the actual separation of the Workflow scheme from the Parameter Editor. At the moment, these two concepts are integrated into one big view within our tool framework. As we also want to provide the integration of completely individualized Parameter views, which are completely detached from our Generic Interface, this separation will be essential.

Although these are planned next steps, it always can be that some ideas might change in the further development progress. Some small features might also become obsolete.

6 Bibliography

[1] A. Schmitz et al.: “D8.1 – Requirements and Interface”, SECONOMICS deliverable D8.1

[2] J. Williams et al.: “Deliverable 8.3: Complete design of Prototype – Security Problem Modeller”, SECONOMICS deliverable D8.3

[3] D. Ríos et al.: „D5.1 – Basic Models for Security Risk Analysis”, SECONOMICS deliverable D5.1

[4] Marek J. Druzdzel. SMILE: structural modeling, inference, and learning engine and genie: A development environment for graphical decision-theoretic models. In: *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pp. 342-343, Orlando, Florida, USA, 1999.

[5] Theodor Schnitzler: “Entwicklung eines Eclipse-Plugins für Matlab-unterstützte Auswertungen des Return On Security Investment“, Bachelor thesis. Dortmund, Germany: TU Dortmund, 2012.

[6] Dominik Thalmann: „Erweiterung der Business Process Modeling Notation für Return on Security Investment Analysen”, Diploma thesis. Dortmund, Germany: TU Dortmund, 2013.

[7] Matlab Builder JA User’s Guide. 2.2.4. The MathWorks, Inc. 3 Apple Hill Drive, Natick, MA 01760-2098, 2012.

[8] The Graphiti Framework Website http://www.eclipse.org/graphiti/

[9] The Eclipse Foundation Website http://www.eclipse.org/

[10] JFreeChart (Graphical engine) Website http://www.jfree.org/jfreechart/

[11] JDOM2 (XML-Engine) Website http://www.jdom.org/

[12] BPMN2 Modeler Website http://www.eclipse.org/bpmn2-modeler/

[13] S. Taubenberger, J. Jürjens, Y. Yu, B. Nuseibeh: “Resolving Vulnerability Identification Errors using Security Requirements on Business Process Models”. In: Journal on Information Management and Computer Security (IMCS), Vol. 21, 2013.

[14] T. Humberg, C. Wessel, D. Poggenpohl, S. Wenzel, T. Ruhroth, J. Jürjens: “Ontology-Based Analysis of Compliance and Regulatory Requirements of Business Processes”. In: Proceedings of the 3rd International Conference on Cloud Computing and Services Science (Closer 2013), 2013.

[15] K. Schneider, E. Knauss, S. Houmb, S. Islam, J. Jürjens: “Enhancing Security Requirements Engineering by Organisational Learning”. In: Requirements Engineering Journal (REJ), Vol. 17, pp. 35-56, 2012.

[16] M. Ochoa, J. Jürjens, J. Cuellar: “Non-interference on UML State-charts”. In: Proceedings of the 50th International Conference on Objects, Models, Components, Patterns (TOOLS Europe 2012) Springer Lecture Notes in Computer Science, 2012.

[17] M. Ochoa, J. Jürjens, D. Warzecha: “A Sound Decision Procedure for the Compositionality of Secrecy”. In: Proceedings of the 4th International Symposium on Engineering Secure Software and Systems (ESSOS 2012). Springer Lecture Notes in Computer Science, 2012.

[18] T. Ruhroth, J. Jürjens: “Supporting Security Assurance in the Context of Evolution: Modular Modeling and Analysis with UMLsec”. In: International Symposium on High Assurance Systems Engineering (HASE), IEEE 2012.

[19] J. Jürjens: “Den Nebel lichten: Von Compliance-Regularien zu testbaren Sicherheitsanforderungen“, iqnite 2012.

[20] J. Jürjens, K. Schneider: „On modelling non-functional requirements evolution with UML“. In: Modelling and Quality in Requirements Engineering (Essays Dedicated to Martin Glinz on the Occasion of His 60th Birthday), Verlagshaus Monsenstein und Vannerdat, 2012.

[21] Jürjens, Jan. "UMLsec: Extending UML for secure systems development." UML 2002—The Unified Modeling Language. Springer Berlin Heidelberg, 2002. 412-425.