Security Requirements Engineering with STS-Tool

Elda Paja[1], Mauro Poggianella[1], Fabiano Dalpiaz[2],
Pierluigi Roberti[1], and Paolo Giorgini[1]

[1] University of Trento – Department of Information Engineering and Computer Science, Via Sommarive 5, 38123, Povo, Trento, Italy – {elda.paja, mauro.poggianella, pierluigi.roberti, paolo.giorgini}@unitn.it
[2] Utrecht University – Department of Information and Computing Sciences, Princetonplein 5, De Uithof, 3584 CC Utrecht, The Netherlands – f.dalpiaz@uu.nl

Abstract. In this chapter, we present STS-Tool, the modelling and analysis support tool for STS-ml, an actor- and goal-oriented security requirements modelling language for socio-technical systems. STS-Tool is a standalone application written in Java and based on the Eclipse RCP Framework. It supports modelling a socio-technical system in terms of high-level primitives such as actor, goal delegation, and document exchange; to express security constraints over the interactions between the actors; and to derive security requirements once the modelling is done. It also supports analysing the created STS-ml models in terms of (i) well-formedness, (ii) violation of security requirements, and (iii) threats impact over actors' assets. We also present the architecture of STS-Tool together with its main features and provide technical details of the modelling and analysis capabilities.

1 Introduction

STS-Tool [6,7] is the graphical modelling and analysis support tool for STS-ml (Chapter 5). STS-ml [1] (Socio-Technical Security modelling language), an actor- and goal-oriented security requirements modelling language for socio-technical systems, which relies on the idea of relating security requirements to interaction. STS-ml allows stakeholders (reified as actors) to express *security needs* over interactions to constrain the way interaction is to take place, and uses the concept of *social commitment* [9] among actors to specify security requirements. For example, if a buyer sends its personal data to a seller, the buyer may require the data not to be disclosed to third parties. Commitments [9] are a pure social abstraction used to model interaction, and in STS-ml they are used to capture security requirements, in terms of promises (social contracts) for the satisfaction of *security needs*. This means that, in STS-ml security requirements are specified as follows: one actor (*responsible*) commits to another (*requestor*) that it will comply with the required *security need*. In the previous example, the seller commits not to disclose personal data to other parties.

In previous work [7] we have shown the use of *social commitments* in specifying security requirements; we have explained how STS-Tool supports modelling and the automated derivation of commitments; we have presented the automated analysis techniques in [5], and illustrated their integration and implementation in STS-Tool to detect

violations of security requirements in [8]. We present details on the underlying modelling language, STS-ml in Chapter 5. Here, we provide the technical details behind the features supported by STS-Tool.

2 Overall features of STS-Tool

STS-Tool offers the following features:

- *Specification of projects*: STS-ml models are created within the scope of project containers. A project refers to a certain scenario, and contains a set of models. Typical operations on projects are supported: create, rename, import, and export.
- *Project Explorer*: a feature of STS-Tool developed as a customization of the Eclipse RNF (Resource Navigator Framework). Since it is not designed to be used inside an RCP application, the integration was a bit challenging. The project explorer allows the user to manage files better, organising them into folders and projects.
- *Diagrammatic modelling*: the tool enables the creation (drawing) of diagrams (models). Diagrams are created only within a project and typical create/modify/ save/load/ undo/redo operations are supported. In particular, STS-Tool supports *multi-view modelling*—different tabs are provided in the tool to allow modelling the various models of a socio-technical system diagram, namely *social*, *information*, and *authorisation model*, following STS-ml's multi-view feature. Each tab (referred to as view) shows specific elements and hides others, while keeping always visible elements that serve as connection points between the models (e.g. roles and agents). Inter-model consistency is ensured by for instance propagating insertion (deletion) of certain elements to all models (social, information, and authorisation) composing the overall STS-ml model.
- *Export diagram to different file formats*: STS-ml models (or parts of models, i.e., specific elements), as well as analysis results, can be exported to various formats, such as jpg, png, pdf, svg, eps, etc.
- *Derivation of security requirements*: the tool allows the automatic derivation of security requirements in terms of relationships between a *requestor* and a *responsible* actor for the satisfaction of a *requirement*.
- *Automated reasoning*: two automated reasoning techniques (*security analysis* and *threat analysis*) are integrated in and supported by STS-Tool. Note that the execution of automated reasoning is to be performed over well-formed models. We verify well-formedness in two steps, depending on the complexity of the check: (i) online or on-the-fly, while the model is being drawn, or (ii) offline, upon user explicit request for computationally expensive checks (embedded within *well-formedness analysis*). *Security analysis* and *threat analysis* are performed upon request of the end-user (security requirements engineer).
- *Visulisation of analyses' results*: the tool visualises the results of the analyses (per each analysis) and provides details of the findings.
- *Generation of requirements documents*: the tool allows the generation of a security requirements document that contains the list of security requirements derived from the model. This document contains information describing the models, information that is customisable by the designer (by choosing which model features to include).

It is good practice to generate the requirements document at the end of the modelling, and after refining the models in order to fix eventual errors detected by the automated analyses. This document is helpful especially when communicating with stakeholders, for it provides details about the different elements of the diagram.

3 Architecture

STS-Tool was developed using the Java programming language, and built on top of different frameworks produced by the Eclipse community. The overall architecture for STS-Tool is depicted in Fig. 1. As shown in this figure, the architecture of STS-Tool is composed of three macro blocks. Starting from the bottom, we find the *System Component* block that contains the underlying operating system (Windows, Linux or OsX) and the Java virtual machine that executes the Java code.

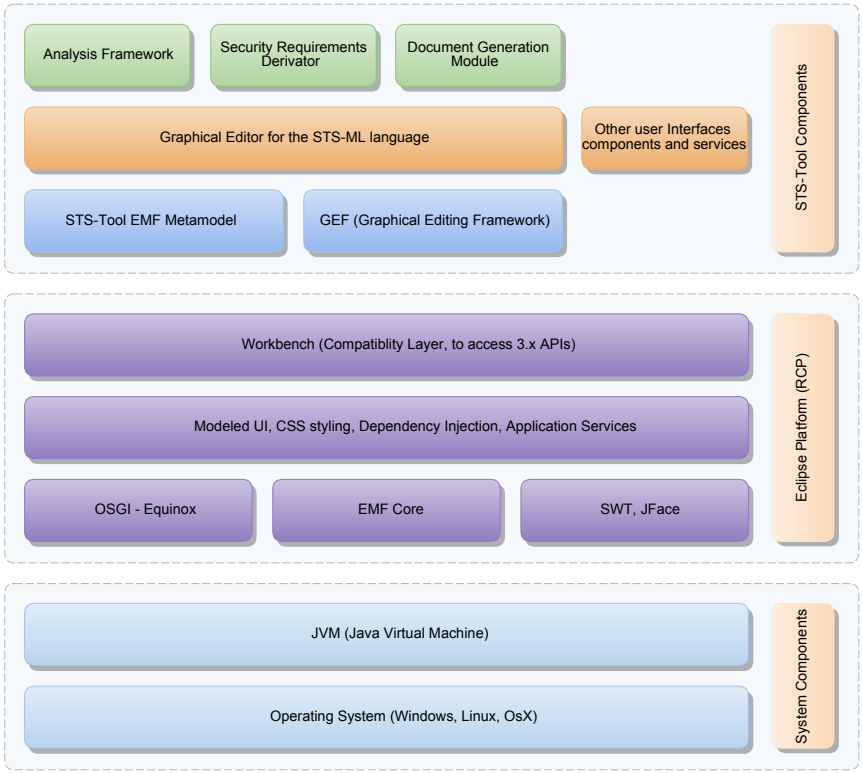

Fig. 1: STS-Tool architecture

At the second layer, we find the *Eclipse Platform*, also known as Eclipse Rich Client Platform (RCP). The Eclipse Platform is developed and maintained by the Eclipse com-

munity[3] and is a powerful framework for building multi-platform standalone applications. An Eclipse application consists of individual software components. According to Vogel in [11], the most important components are:

- OSGi: a specification that describes a modular approach for Java application. The programming model of OSGi allows defining dynamic software components, namely OSGi services.
- Equinox: one implementation of the OSGi specification and is used by the Eclipse platform. The Equinox runtime provides the necessary framework to run a modular Eclipse application.
- SWT: the standard user interface component library used by Eclipse. JFace [12] provides some convenient APIs on top of SWT, while the workbench provides the framework for the application. It is responsible for displaying all other UI components.
- Eclipse 4: provides a compatibility layer which allows that plug-ins using the Eclipse 3.x programming model can be used unmodified in an Eclipse based application.

One of the major advantages of this platform is *modularity*. To achieve this Eclipse uses plugins. Each plugin is an independent module that provides a specific functionality inside the application, and can be easily added or replaced. Moreover, every plugin can define or consume *extension points* that allow other plugins to contribute functionality to the defined plugin. Due to the high modularity of the system it is possible to add new features with little effort and maintain code easily.

Eclipse provides the SWT [4] graphical library, which allows to build efficient and portable applications that directly access the user-interface facilities of the operating systems it is implemented on. This revolutionary technology makes it possible to create Java-based applications that are indistinguishable from a platform's native applications.

Last but not least, the Eclipse community develops a lot of parallel projects for various purposes that can be integrated to the Eclipse Platform, making the entire system more powerful.

These are some of the reasons that Eclipse was chosen as the underlying platform for developing the STS-Tool.

Finally, at the third layer, we find the *STS-Tool Components*. The STS-Tool allows the user to create and modify STS-ml models (i.e., diagrams) described using a specific language, namely STS-ml. To support the particular specification of STS-ml, a graphical editor was implemented using the GEF Framework [3]. The STS-ml metamodel is incorporated to ensure that diagrams follow the syntax of STS-ml. The rest of STS-Tool components correspond to the features it supports, such as analysis, security requriements derivation, and security requirements document generation. We provide more details on the implementation of each in Sec. 5.

4 Installation details

The STS-Tool is distributed as a compressed archive for multiple platforms and it is free to download from the STS-Tool website [4]. The tool is available in both source and

[3] `https://www.eclipse.org/`

[4] `http://www.sts-tool.eu/`

binary form, and the license is APGL (Affere General Public License). As prerequisite, at least version 6.26 of the Java Virtual Machine is needed. Previous versions of the tool are also available online in Archive [5].

The installation of STS-Tool requires no setup, it is enough to download the version suitable for the machine and operating system under consideration, extract the content of the archive containing the tool, to finally run the launcher file.

The STS-Tool comes with *Online Help*. Help is produced by the Eclipse project, we provide only the content of the help.

To obtain updates of the STS-Tool, one does not need to download the latest version from the website, rather an *update system* is already integrated in the tool. The update system is a customization of the Eclipse P2 (Eclipse update system). A public web site was expressly created in order to update the new versions of the tool automatically. The STS-Tool checks for updates and if any are found, it asks the user to install them. However, the user has the choice of activating this feature (configuring updates from the menu: Windows – Preferences – Updates) or getting updates manually.

5 Technical implementation details of STS-Tool

We provide technical details on how the STS-Tool supports modelling (Sec. 5.1), security requirements derivation (Sec 5.2), analysis (Sec 5.3), and security requirements document generation (Sec 5.4).

5.1 Modelling with STS-Tool

To implement the graphical editor for the STS-ml language, the GEF Framework [3] was chosen. The GEF Framework is an interactive Model-View-Controller (MVC) framework, which fosters the implementation of SWT–based tree editors [11], and Draw2d–based [3] graphical editors for the Eclipse Workbench UI [13]. One of the challenges faced in the development of the graphical editor was related to the fact that the GEF framework is a single view editor, while the STS-Tool editor had to be a multi-view editor in order to support the multi-view modelling of STS-ml models. The problem was solved by implementing a custom multi-view editor starting from the class MultiPageEditorPart (org.eclipse.ui.part.MultiPageEditorPart) [6].

Currently, STS-ml supports three views. However, considering a possible evolution of the language and tool, we took advantage of the modular nature of the platform to create a new extension point in order to allow the automatic addition (insertion) of new views (should there be any in the future). The MultiPageEditor reads the extension point and creates the required objects to then allow their initialization.

The extension point id is it.unitn.disi.ststool.editor.subparts and can have an infinite number of children (one for each view) of 2 different types: *Subeditor* or *View*, see Fig. 2. The difference between them is the type of interface they must implement. The two

[5] http://www.sts-tool.eu/Archive/

[6] http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fui%2Fpart%2FMultiPageEditorPart.html

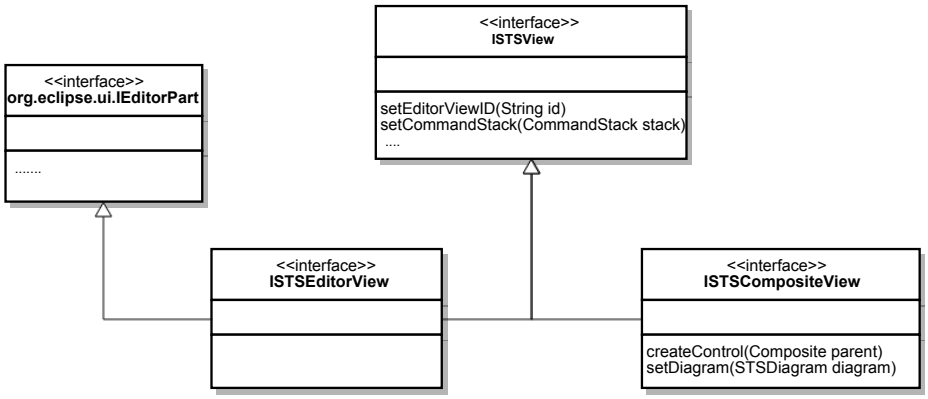

Fig. 2: STS-Tool Graphical Editor Support via Extension Points

interfaces are similar. They both extend the common interface ISTSView, which defines the common methods each view must have, but the editor interface the ISTSEditorView extends also from the org.eclipse.ui.IEditorPart allowing to add Eclipse editors as view (as the GEF Editor), while the ISTSCompositeView allows to add simple composites as views, useful to create a textual view. Each view must declare a unique id to make one view distinguishable from the others. This id is subsequently used in the code to identify the correct objects GraphicalConstraint (see Sec. 5.1–**The Model**).

After implementing the multi-view feature, we concentrated on the fundamental modelling features of the editor. As already mentioned above, a GEF Editor was extended. Since GEF is a MVC Framework, three main objects should be provided for each, namely *the model*, *the view* and *the controller* (see Fig. 3).

In the following paragraphs, we discuss in detail each and every one of the objects.

The Model. Each diagram the user displays or edits is described by an underlying Java model. Each Java object contains its own properties, and represents a specific element that will be displayed on a graphical canvas. The STS-Metamodel was implemented using the EMF Framework [2]—a modelling framework and code generation facility for building tools and applications based on a structured data model. It provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model.

Apart from the Java code creation facility, EMF provides a powerful notification system that is necessary when working with MVC frameworks. The notification system notifies all the registered listeners when a property of one object changes so a graphical update can occur. The STS-Metamodel (see Fig. 1) was created using this framework, and was adapted to reflect the STS-ml language properties. The model was included in the it.unitn.disi.ststool.model plugin.

To avoid mixing model properties with graphical properties the model was logically separated in two parts:

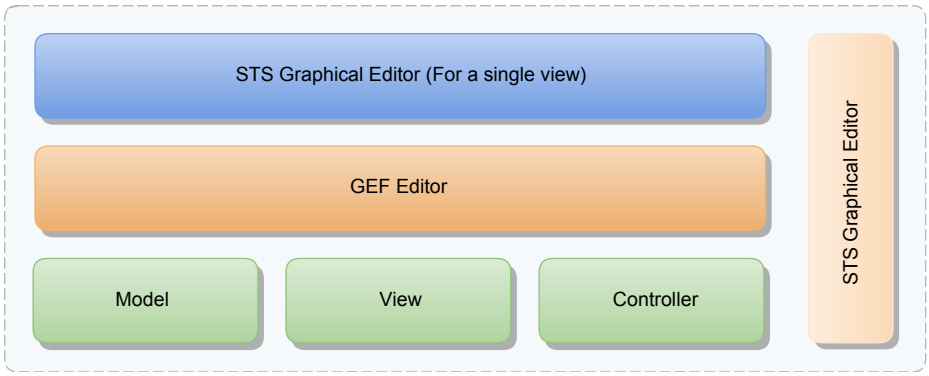

Fig. 3: STS-Tool Graphical Editor

1. The abstract class does not specifically represent an STS-ml element, while the main class from which all other elements derive is the STSElement. This class defines the common element attributes, such as the unique identifier or the description properties, to be inherited by all subclasses [7]. Subsequently, STS-ml elements (Nodes) and relationships (Links) are represented through two dinstinct classes: STSNode and STSLink. The class GraphicalConstraint storer graphical information such as node bounds or link binding points, which is useful to maintain separated the graphical information from the model information. Given that such constraints are specified over different elements and relationships (representing interaction), the class is extended to differentiate among Nodes', Links', and Diagram's GraphicalConstraint. This is useful to avoid continuous casting in the code too. GraphicalConstraint can have multiple properties, GraphicalProperty, in the form of key/value, which facilitates the creation/addition of new non-default properties.
2. The real STS-ml model representation. Each STS-ml model object is reproduced in the STS-EMF model. A plugin was created from the metamodel description: the it.unitn.disi.ststool.edit plugin. This plugin is also generated from the EMF Framework and contains a set of classes that allow to display and edit a model object through the property view (as supported by Eclipse).

The View. The views for the GEF editor are made of simple shapes. These shapes are created using the eclipse draw2d library and each shape extends the org.eclipse.draw2d.IFigure interface. The IFigure was not implemented from scratch, rather the Shapes figure provided by the draw2d library was extended. Views as such do not have any information about the object model they are representing, but they are connected to them through the *Controller*. This is useful to keep the model separated from the view and maintain the code clear and easily reusable.

[7] Class STSDiagram is an exception, as it defines the overall diagram. Thus, it is the parent of all other objects in the diagram, serving as a root container so the entire model is represented by this object.

The Controller. The controller part is composed of classes that extend the org.eclipse.gef.editparts.AbstractGraphicalEditPart class. Each editPart knows the *model object* it has to represent and the *figure* associated to it, and is able to *link* them. Editpart is responsible for tracking model (diagram) changes and update the view in which changes occur. The user interaction, on the other hand, is tracked by the EditingDomain and forwarded to the corresponding editPart with a org.eclipse.gef.Request describing that the user event occurred. The editPart processes the request through its policy and produces a org.eclipse.gef.commands.Command that contains the code to modify the model object (and also the code to undo the modifications). The command is successively passed to the commandStack that finally executes the command. The purpose of the command stack is to keep a history of the executed commands and allow undoing them. Once the command is executed the model object is modified, and an event is propagated to the editPart to update the respective view.

Each view editor shares with the other editors two important things: the *diagram model object* and the CommandStack, while each has its own palette that is populated during the construction of the editor. This allows the different views to have distinct palettes. View filtering is made in the EditPart: each EditPart is responsible for listing the children objects and the connections that start and end to the represented model element. Filtering this list supports the filtering of what the editor displays.

5.2 Security Requirements Derivation

In this section, we present how security requirements are generated in STS-Tool starting from security needs (Chapter 5, Section 2.2). Each Element in the STS Metamodel derives from the STSElement class which defines a containment relation (0..*) with an object of type ElementNeed, which keeps track of the security need specified over the element. Therefore, each Element in the model is related to its element need. Each element need has an id that identifies the type of the element need, and a map of $key \rightarrow values$ of properties, used to store specific values of the security need, which allows having multiple values for a single element need. This solution was chosen over the one of binding specific properties on each model object, to make it possible to add requirements through an extension point, without modifying and sequentially regenerating the metamodel code. Moreover, this option supports the automatic creation of menus without modifying the editor code. That is, if a new security need is required to be supported by STS-Tool, it can be added through a new plugin. This is made possible by the it.unitn.disi.ststool.development.model. elementneed extension point. This extension point allows two types of children, an elementNeedGroup and an ElementNeed respectively.

The elementNeedGroup requires 3 parameters:

1. id: uniquely identifies the group,
2. $name$: is displayed in the menu, and
3. $parent\ group\ id$: an optional id to create submenus.

The ElementNeed ,instead, requires 6 parameters:

1. id: uniquely identifies the element need type,

2. $groupId$: points to an elementNeedGroup ID,
3. $name$: is displayed to the user,
4. $short\ name$: used in the graphical representation,
5. $color$: used in the graphical representation,
6. $applicableTo$: string value that contains a list of comma separated Class names on which the elementNeed will be applied (will be displayed in is context menu).

Furthermore the ElementNeed allows five types of children, which describe the properties of the ElementNeed. Each ElementNeed can have multiple of these children allowing an element need to support multiple properties. These children are:

1. $singlechoice\_value$: allow to choose a single value from a list;
2. int_value: allow to insert an integer value;
3. $string\_value$: allow to insert a text value;
4. $bool\_value$: allow to insert a Boolean value;
5. $percent\_value$: allow to insert a value in range 0-100.

For now these are the supported values, but in future versions of the STS-Tool, newer value types could be added. In STS-Tool, security requirements are a specialization of the ElementNeed. They are defined in a separate plugin unitn.disi.ststool. securityrequirements. This plugin defines the ElementNeed used in the STS-ml language, and also provides a security requirement generator and a view (the security requirements view) to display the evaluated security requirements. To perform an evaluation, two different components are involved. The first component is the SecurityRequirementsManager singleton. This component tracks the current active editor and if a valid editor is found it retrieves the associated STSDiagram Model object and delegates the evaluation of the security requirements to the second component the SecurityRequirementEvaluator. The Manager also tracks the changes on the model and when they occur it asks the evaluator to perform a new evaluation. When the evaluation completes the result of the evaluation is stored and registered listeners to the manager, as the security Requirements View, are updated with the new evaluation result. The SecurityRequirementEvaluator instead takes an STSDiagram as parameter and when started recursively iterates over the entire model and for each element it evaluates its ElementNeeds and generates one (or multiple) objects of type ISecurityRequirement that describe(s) the security requirements associated to the ElementNeed. When no more elements are found, the complete list of ISecurityRequirement is returned. A possible improvement of this implementation: the SecurityRequirementEvaluator contains an hardcoded set of rules to evaluate and create the correct implementation of the ISecurityRequirement, in the future this could be moved to an extension point. The SRS Generator is another component included in the Security Requirements plugin. When invoked by the user (through a generate SRS button) it retrieves from the SecurityRequirementsManager the list of the security requirements and transform them into an XML file.

5.3 Reasoning about Security Requirements

Here we show the automated reasoning capabilities implemented in STS-Tool.

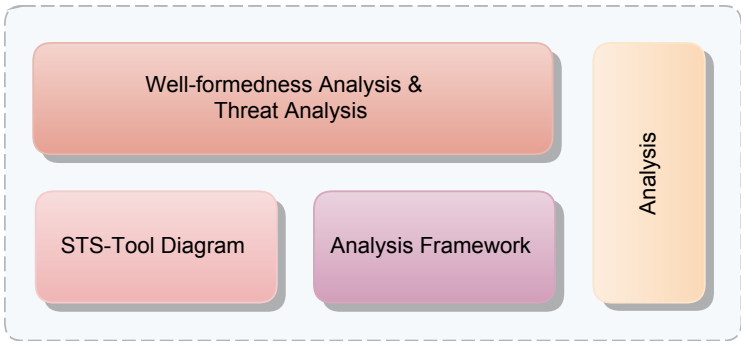

Fig. 4: STS-Tool Automated Analysis

STS-Tool supports analysis activities through a dedicated analysis module, as depicted in Fig. 4. Similarly to the other STS-Tool supported features, the analysis module was developed and integrated through specific plugins. In the following, we describe each and every plugin used for the analysis module.

1. The it.unitn.disi.ststool.analysis plugin supports the execution of analysis over an STSDiagram object. This is achieved through a small framework, which has the following main interfaces:
   (a) IAnalysis: describes an analysis and contains a list of ITasks to be executed.
   (b) ITask: defines a generic task in the analysis, and the dependencies to other tasks. This interface cannot be directly implemented (see IComposedTask, IExecutableTask).
   (c) IComposedTask: defines a composite task that contains subtasks. When this task is started the subtasks are executed and the result of the task is the worst result of the children. This is useful when a single task is composed of multiple tasks that have dependencies between them (e.g., in the security analysis, the *AuthorityViolations task* is composed of other subtasks. The first subtask named *preanalysis*' executes the ViolationAnalysis, but if for some reason this fails, the other sibling tasks are skipped because they need the results of the *preanalysis*).
   (d) IExecutableTask: defines a task that will execute a piece of code performing an evaluation and returning a result.
   (e) IResult: describes a result.
   (f) IAnalysisEngine: retrieved from the AnalysisEngineFactory singleton is the Engine that will execute the IAnalysis.
   (g) ITaskEvent: the Analysis engine also supports events to notify registered listeners of the analysis progress. The progress of the analysis is made through this event object. The it.unitn.disi.ststool.analysis plugin also provides the graphical user interfaces to display the analysis results and other utilities classes, useful when performing analysis.
2. The it.unitn.disi.ststool.analysis.wellformedness plugin and the it.unitn.disi.ststool.analysis.threat plugin contain respectively the well-formedness analysis

implementation and the threat analysis implementation. These plugins provide the ITask implementation needed to perform the analysis. These analyses are performed completely in Java, analysing the STSDiagram Model object.

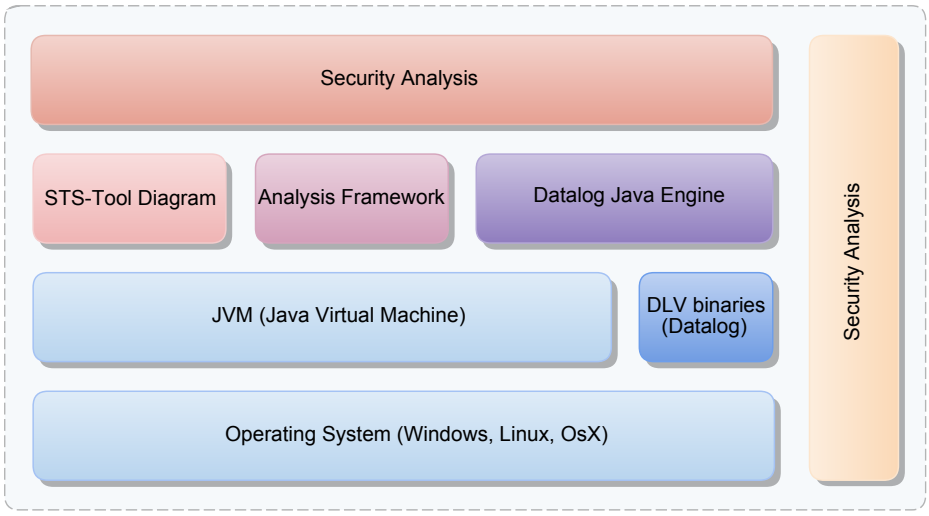

Fig. 5: STS-Tool Security Analysis

3. For the security analysis things get complicated. Since the security analysis is implemented in disjunctive Datalog (see Fig. 5), it requires the use of Datalog program and engine. But, the Datalog program is released only in native OS executables, and thus, a Java wrapper had to be implemented. To make it reusable, this wrapper was developed in a separate plugin, namely it.unitn.disi.ststool.analysis.dlv.

The java DLVEngine class is in charge of recognizing the current operating system in use and selecting the correct executables. It creates the required files on the filesystem and when requested, it executes the program using the Runtime.exec(String params) instructions. To make the DLVEngine more flexible, another class was inserted, namely EngineExecutionParameters, which contains methods to configure the DLVEngine, such as setting the maximum number of models or setting filters, and also contains the Datalog code that has to be executed. The output produced by the DLVEngine is parsed by a provided implementation of the EngineOutputReader class. This set of classes makes the use of the DLV binaries transparent to the Java code. The it.unitn.disi.ststool.analysis.security plugin contains the task that executes the security analysis. The fundamental tasks of the security analysis rely on a particular analysis (that is made internally and not in a separate plugin) called ViolationsAnalysis. This analysis uses the Datalog engine in order to be executed and completed. Some other classes have been added to support this analysis:

– Predicate: this class object represents a Datalog predicate; it has a name and contains parameters that are mapped to a model object.

– Violation: a wrapper for a predicate. This class, apart from containing the predicate, contains also other values derived from the analysis, such as the total number of occurrences of the predicate, and the total number of models generated by the DLV engine.
– ViolationDefinition: while Violation is used to wrap the result of the analysis, the ViolationDefinition class serves the purpose of containing the required values to discover a Violation. In particular, this class contains two attributes: (i) a predefined list of Datalog predicates that will compose the final Datalog program code, and (ii) the name of the predicate that will be generated by the Datalog program execution when a violation is found.

The following schema (see Fig. 6) summarizes the process guiding the security analysis and the components involved in the process, which are integrated in STS-Tool.

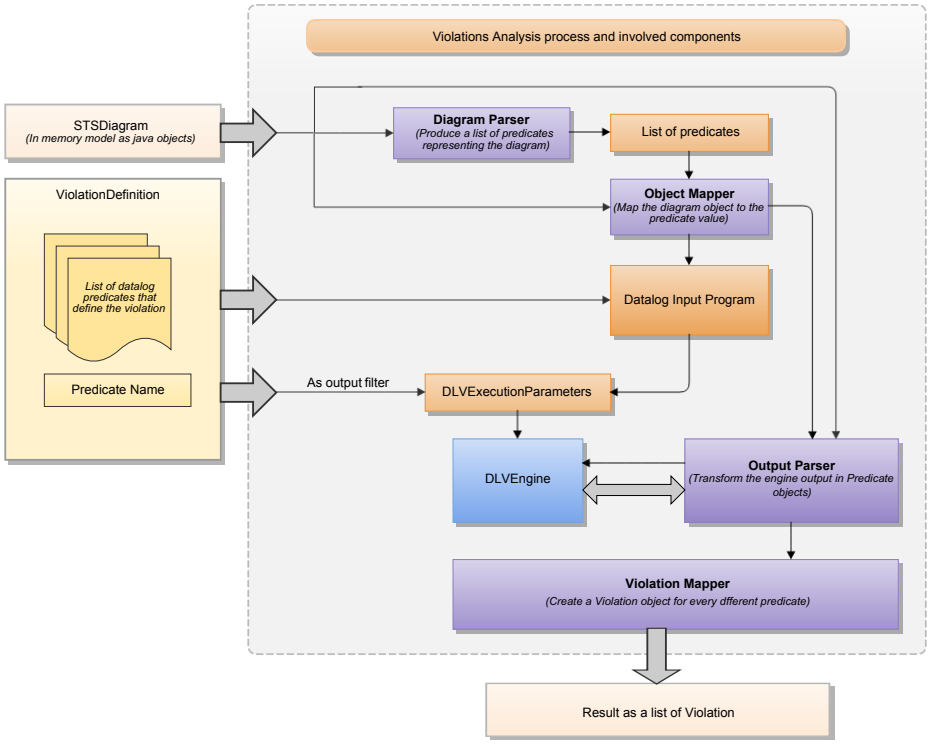

Fig. 6: STS-Tool Security Analysis: Verification of Security Requirements Violations

The ViolationsAnalysis requires two input parameters: an STSDiagram and a ViolationDefinition. When the analysis is started some events occur. First of all, a specific parser containing a set of rules iterates over the entire STSDiagram transforming each element in a Datalog predicate. The generated predicates are transformed one by one into String and the predicate values are mapped in a temporary Map. This allows us

to be able to reconstruct the predicates when the DLV output is parsed. At this point the String that represents the predicates derived from the diagram and the predicates provided by the ViolationDefinition are merged together and passed as InputProgram through an instance of a DLVExecutionParameters to the DLVEngine. Afterwards the predicate name, which is retrieved from the ViolationDefinition, is set as the filter option to the DLVEngine. Setting the output filter drastically reduces the output produced by the DLV program and consequently improves memory footprint and efficiency. The engine can be started with the configured parameters. The output parser reads the output produced by the engine, interprets it and creates instances of Predicate. When the DLVEngine completes its execution, all the generated predicates are transformed in Violation instances and returned to the specific Security analysis task that requested them, and are used to produce the analysis results.

Visualising analysis results. The visualisation of the analysis results passes through a singleton object, specifically designed to manage the analysis results. Each analysis, once completed, provides to the ResultManager its results. Results are collected, stored, and displayed into the analysis view. Every result object has some properties: a *text* and a *description* that are used in the user interface to describe the result, the gravity of the result (OK,WARNING,ERROR), and a list of elements (or model objects) that have to be highlighted when the user wants to display them over the STS-ml model (diagram). After the results are displayed in the analysis view, the user can select one or more results to show them over the diagram. Once a result is selected, this action causes a ResultManager retrieve from the selected results, which returns the list of objects that need to be highlighted and modified over the diagram (by setting a graphical property to each of this object graphical constraint), so the editor can change the displayed colour of the element to highlight it on the diagram.

5.4 Generating the Security Requirements Document

The STS-Tool supports the generation of the security requirements document, which is supported by the *report module*, as shown in Fig. 7. Similarly to the analysis module, the report module was distributed into multiple plugins.

The main plugin is the it.unitn.disi.ststool.documents. This plugin contains only the Java aspose libraries, which allow editing and creating text documents and presentation documents in multiple formats, as well as some UI classes and a set of classes that generate the final documents. While this plugin is involved in managing the creation of the documents, the content of the final document is obtained (contributed) via an extension point it.unitn.disi.ststool.documents.report.contribution that other plugins can use to add their own content. This extension point accepts multiple children of type contribution. A contribution must have a unique id, a priority value (a number) to make contributions sortable and a class that will perform the contribution (add the content of the document). Moreover the contribution type must have at least one child of type part. The part object is used in the graphical selection of the parts that can be generated, to allow customisation of the security requirements document by the analyst. This object has a unique id that can be used later in the code to retrieve the information about the selection, a name that is used in the UI, and

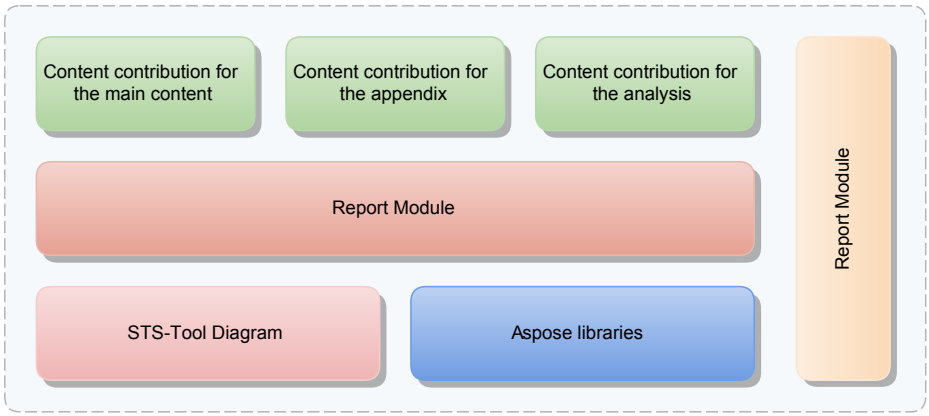

Fig. 7: STS-Tool Security Requirements Document Generation

some Boolean properties. part can also have part children to allow multi-level part-selection. While the it.unitn.disi.ststool.documents plugin contributes only to the report generation functionality, the it.unitn.disi.ststool.documents.report.analysis.security, it.unitn.disi. ststool.documents.report.analysis.wellformedness, it.unitn.disi.ststool. documents.report.securityreqirements, and it.unitn.disi.ststool.documents.report.view contribution plugins contribute to the content providing it through the extension points previously described. The content is generated analysing the STSDiagram; static code retrieves from the model the required information, caches them and at the end generates the content that has to be inserted into the document.

6 Conclusions

The STS-Tool is quite a stable graphical requirements engineering modelling tool. It is based on the Eclipse EMF framework and supports modelling and reasoning over the created models. The latest version of the tool is the result of an iterative development process, having been tested on multiple case studies and evaluated with practitioners [10] in the scope of Aniketos. STS-Tool was proved suitable to model and reason over models of a large size from different domains [5], such as eGovernment (see Chapter 15) or Air Traffic Management Control (see Chapter 14).

References

1. Fabiano Dalpiaz, Elda Paja, and Paolo Giorgini. Security requirements engineering via commitments. In *Proceedings of STAST'11*, pages 1–8, 2011.
2. The Eclipse Foundation. Eclipse modeling framework project (emf), 2014. Lastchecked: March, 2014.
3. The Eclipse Foundation. Gef (mvc), 2014. Lastchecked: March, 2014.
4. Steve Northover and Mike Wilson. *Swt: the standard widget toolkit, volume 1*. Addison-Wesley Professional, 2004.

5. Elda Paja, Fabiano Dalpiaz, and Paolo Giorgini. Managing security requirements conflicts in socio-technical systems. In *Proceedings of ER'13*, 2013. To Appear.
6. Elda Paja, Fabiano Dalpiaz, Mauro Poggianella, Pierluigi Roberti, and Paolo Giorgini. STS-Tool: socio-technical security requirements through social commitments. In *Proceedings of RE'12*, pages 331–332, 2012.
7. Elda Paja, Fabiano Dalpiaz, Mauro Poggianella, Pierluigi Roberti, and Paolo Giorgini. STS-Tool: Using commitments to specify socio-technical security requirements. In *Proceedings of ER'12 Workshops*, pages 396–399, 2012.
8. Elda Paja, Fabiano Dalpiaz, Mauro Poggianella, Pierluigi Roberti, and Paolo Giorgini. Specifying and reasoning over socio-technical security requirements with sts-tool. In *Proceedings of the 32nd International Conference on Conceptual Modeling, ER Workshops*, pages 504–507, 2013.
9. Munindar P. Singh. An ontology for commitments in multiagent systems: Toward a unification of normative concepts. *Artificial Intelligence and Law*, 7(1):97–113, 1999.
10. Sandra Trösterer, Elke Beck, Fabiano Dalpiaz, Elda Paja, Paolo Giorgini, and Manfred Tscheligi. Formative user-centered evaluation of security modeling: Results from a case study. *International Journal of Secure Software Engineering*, 3(1):1–19, 2012.
11. Lars Vogel. Building eclipse rcp applications based on eclipse 4, 2013. Revision history: Revision 0.1 - 6.9 14.02.2009 - 04.07.2013.
12. Lars Vogel. Eclipse jface tree - tutorial, 2013. Revision history: Revision 0.1 - 0.1 - 3.3 22.08.2010 - 15.10.2013.
13. Stefan Xenos. Inside the workbench a guide to the workbench internals, October 2005. Lastchecked: March, 2014.