SEWOL Tutorial

Documentation & Code Examples for the Security-oriented Workflow Library

IIG Telematics, University of Freiburg

Contents

1 Introduction **1**
1.1 Library Dependencies . . . 2
1.2 Package Structure . . . 2

2 Access Control Models **2**
2.1 ACL: Access Control List . . . 3
2.2 RBAC: Role-based Access Control . . . 4

3 SOA Base & Contexts **5**

4 Format of Workflow Traces **8**
4.1 Log Entry . . . 8
4.2 Log Trace . . . 9
4.3 Log . . . 10

5 Serializing & Parsing Logs **11**

1 Introduction

The Security-oriented Workflow Library (SEWOL) provides support for the handling of workflow traces. It allows to specify the shape and content of process traces in terms of entries representing the execution of a specific workflow activity. SEWOL also allows to write these traces on disk. For this it uses a specific file writer for process logs. Currently it supports plain text, Petrify, MXML and XES log file types.

In order to specify security-related context information, SEWOL provides access control models such as access control lists (ACL) and role-based access control models (RBAC). All types of models can be conveniently edited with the help of appropriate dialogues.

This document is a mix of programming tutorial and library documentation, where the features are demonstrated with some visualisations and minimal programming code examples.

The examples in this document have been tested against the SEWOL release version 1.0.0, which can be found at `https://github.com/iig-uni-freiburg/SEWOL`. This document was last updated on July 29, 2015.

1.1 Library Dependencies

SEWOL builds upon the following tools. To use SEWOL, make sure all these libraries are included in the classpath.

- TOVAL, located at `https://github.com/GerdHolz/TOVAL`
- JAGAL (Java Graph Library), located at `https://github.com/iig-uni-freiburg/JAGAL`
- OpenXES, located at `http://www.xes-standard.org/openxes/` and enclosed by the SEWOL library
- Spex, located at `http://code.deckfour.org/Spex/` and enclosed by the SEWOL library
- Google Guava, located at `https://github.com/google/guava`
- XStream, located at `http://xstream.codehaus.org/`
- Jung 2, located at `http://jung.sourceforge.net/`

1.2 Package Structure

The packages in `de.uni.freiburg.iig.telematik.sewol` are logically divided into the following sub-packages.

The package `accesscontrol` defines different access control models like access control lists (ACL) and role-based access control models (RBAC). The sub-package `graphic` contains graphical dialogues to conveniently edit the access control models. The package `context` contains a data usage context for processes (`ProcessContext`), which extends TOVAL's `SOABase` (list of possible subjects, objects, and activities) by data usage modes (read, write, create, and delete) for activities.

The data structure of workflow traces is defined within the package `log`. The package `format` contains the file format definitions for logs in the plain (whitespace-separated lists of activities), Petrify, MXML, and XES file format. These file formats are used by the parsing classes in the package `parser` and the serializing classes in the package `writer`. Helper classes can be found in the `util` package.

2 Access Control Models

The term "access control" refers to the policing and control of the access on specific resources to achieve information integrity and confidentiality together with information availability. To reach these aims, different access control models like ACL and RBAC have been established.

Both access control models inherit from the class `AbstractACModel`, which has a `name` as parameter, a list of valid data usage modes, and contains a list of activities, subjects, and objects in form of a `SOABase`. A `SOABase` could look like this:

```
SOABase base = new SOABase("base");
base.setActivities("act_1", "act_2", "act_3");
base.setSubjects("sub_1", "sub_2", "sub_3");
base.setObjects("obj_1", "obj_2", "obj_3");
```

2.1 ACL: Access Control List

An access control list (ACL) represents user permissions in form of a list. For each object and activity it contains rights of subjects in form of access operations. For an object `obj_x`, e.g, the ACL could contain the rights `{sub_1: read; sub_2: read,write}`.

An ACL can easily be built with the following code:

```
ACLModel acl = new ACLModel("ACL", base);
acl.addActivityPermission("sub_1", "act_1");
acl.addActivityPermission("sub_1", "act_2");
acl.addActivityPermission("sub_2", "act_3");
acl.addActivityPermission("sub_3", "act_2");
acl.addObjectPermission("sub_1", "obj_1", DataUsage.READ);
acl.addObjectPermission("sub_1", "obj_2", DataUsage.READ);
acl.addObjectPermission("sub_1", "obj_3", DataUsage.READ,DataUsage.CREATE);
acl.addObjectPermission("sub_2", "obj_2", DataUsage.DELETE);
acl.addObjectPermission("sub_3", "obj_3", DataUsage.WRITE);
System.out.println(acl);
```

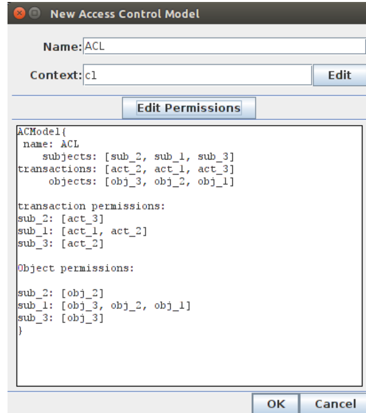
Corresponding output of the code above:

```
ACModel{
 name: ACL
    subjects: [sub_2, sub_1, sub_3]
transactions: [act_2, act_1, act_3]
     objects: [obj_3, obj_2, obj_1]

transaction permissions:
sub_2: [act_3]
sub_1: [act_1, act_2]
sub_3: [act_2]

Object permissions:
sub_2: [obj_2]
sub_1: [obj_3, obj_2, obj_1]
sub_3: [obj_3]
}
```

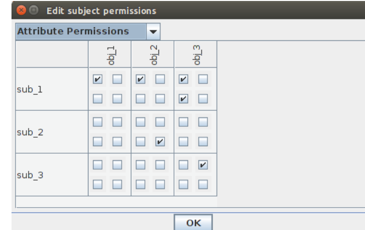
For the convenient editing of ACLs SEWOL comes with a graphical dialogue. One can either provide a `SOABase` for editing or can create a new one using the dialogue. The following code creates a new Java Swing dialogue with an existing base:

```
ACModelDialog.showDialog(null, "ACL", ACModelType.ACL, base);
```

The resulting Java Swing dialogue can be seen in Figure 1a. With a click on the *Edit permissions* button, a new dialogue opens, which allows to edit activity and attribute permissions (see Figure 1b). The permissions can be seen in the preview window of the dialogue, where lists of activities and objects are assigned to subjects.



(a) ACL editing dialogue with preview.



(b) Dialogue for editing attribute permissions.

Figure 1: Dialogue for editing ACLs in SEWOL.

2.2 RBAC: Role-based Access Control

ACLs can become very complex and cumbersome to manage with a growing number of subjects, activities, or objects. Role-based access control models add roles as an additional abstraction level. In a role-based access control model, subjects are assigned to (multiple) roles. The permissions to perform certain activities or use certain objects are assigned to roles instead of assigning them directly to subjects. Thus, the management of a subject's permissions is confined to assigning users to appropriate roles. This also simplifies common operations like adding a subject or changing a subject's responsibilities. Further roles can inherit permissions from other roles.

The following code defines an RBAC model, where role `r1` dominates `r2` and therefore inherits its permissions. This right propagation must be enabled manually using the method `setRightsPropagation`. The relations between roles are specified by a `RoleLattice` object, whose inner representation is a graph with the role names as nodes.

```
// Role definition with their relations
RoleLattice lattice = new RoleLattice(Arrays.asList("r1", "r2"));
lattice.addRelation("r1", "r2");
// RBAC model definition
RBACModel rbac = new RBACModel("RBAC", base, lattice);
rbac.setRightsPropagation(true);
rbac.setRoleMembership("r1", Arrays.asList("sub_1", "sub_3"));
rbac.setRoleMembership("r2", Arrays.asList("sub_2"));
rbac.setActivityPermission("r1", "act_1","act_2");
rbac.setActivityPermission("r2", "act_3");
rbac.setObjectPermission("r1", "obj_1", DataUsage.WRITE);
rbac.setObjectPermission("r1", "obj_3", DataUsage.READ,DataUsage.CREATE);
rbac.setObjectPermission("r2", "obj_2", DataUsage.DELETE);

rbac.isAuthorizedForObject("sub_1", "obj_2", DataUsage.DELETE);
// returns true
rbac.isAuthorizedForTransaction("sub_1", "act_3");
// returns true
```

```
System.out.println(rbac);
```

The code above produces the following output.

```
    subjects: [sub_2, sub_1, sub_3]
transactions: [act_2, act_1, act_3]
     objects: [obj_3, obj_2, obj_1]
       roles: [r1, r2]

Role transaction permissions:
r1: [act_2, act_1, act_3]
r2: [act_3]

Role object permissions:
r1: [obj_3, obj_2, obj_1]
r2: [obj_2]

role assignments:
r1: [sub_1, sub_3]
r2: [sub_2]
```

At the top, the SOA base is shown, which has been extended by roles. Afterwards the roles' permissions on transactions and objects are listed and at the end, the assignment of subjects to roles can be found.

The graphical dialogue `ACModelDialog` can also be called for creating and editing an RBAC model. This is done using the following code:

```
RBACModel m = ACModelDialog.showDialog(null,"RBAC",ACModelType.RBAC,base);
```

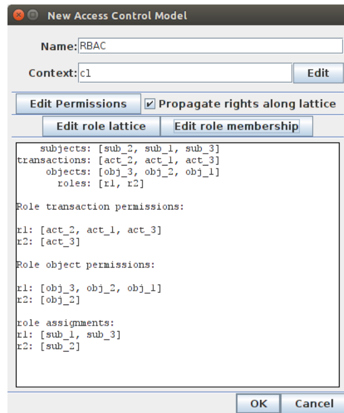
The resulting user interface for editing the RBAC model can be seen in Figure 2a. When clicking the button with the label *Edit role lattice* a new window opens, in which one can add and remove roles and specify inheritance relations between them (see Figure 2b).

Since the checkbox *Propagate rights along lattice* is selected and `r1` dominates `r2` in Figure 2b, the role `r1` inherits permissions for activity `act_3` and object `obj_2` from `r2`.

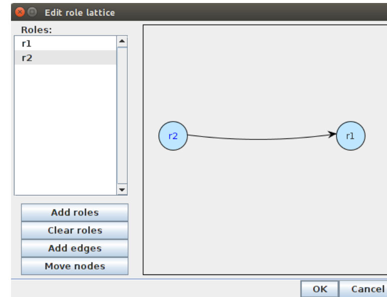
3 SOA Base & Contexts

Some components in SEWOL, like access control models, use `SOABase`s, which contain lists of possible activities, subjects, and objects. In the access control models in section 2 it has been used implicitly. SEWOL defines a `ProcessContext`, which supplements the `SOABase` class by an access control model and a mapping of data usage modes for specified objects in an activity. This way permissions of subjects to execute and process activities to use objects with specified usage mode can be defined. Figure 3 shows the relation of SOA base, process context, and access control model.

A process context based on an existing SOA base can be built using the following code snippet. As underlying access control model the formerly defined RBAC model is used.



(a) Dialogue for editing RBAC with preview of the access control model.



(b) Dialogue for editing roles and the inheritance hierarchy among them.

Figure 2: Dialogue for editing RBAC models in SEWOL.

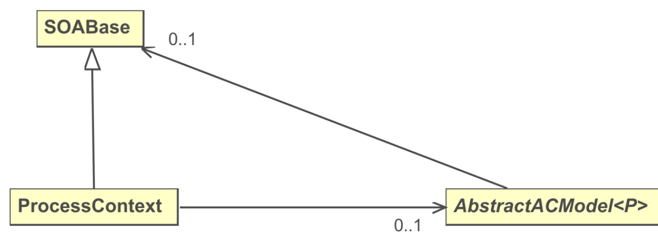


Figure 3: Dependencies between the `SOABase`, `ProcessModel` and the access control models.

```
ProcessContext context = new ProcessContext(base);
context.setACModel(rbac);
context.setDataUsageFor("act_1", "obj_1", DataUsage.WRITE);
context.setDataUsageFor("act_1", "obj_3", DataUsage.READ,DataUsage.CREATE);
context.setDataUsageFor("act_2", "obj_1", DataUsage.WRITE);
context.setDataUsageFor("act_2", "obj_3", DataUsage.READ,DataUsage.CREATE);
context.setDataUsageFor("act_3", "obj_2", DataUsage.DELETE);
System.out.println(context);
```

The corresponding output is the following:

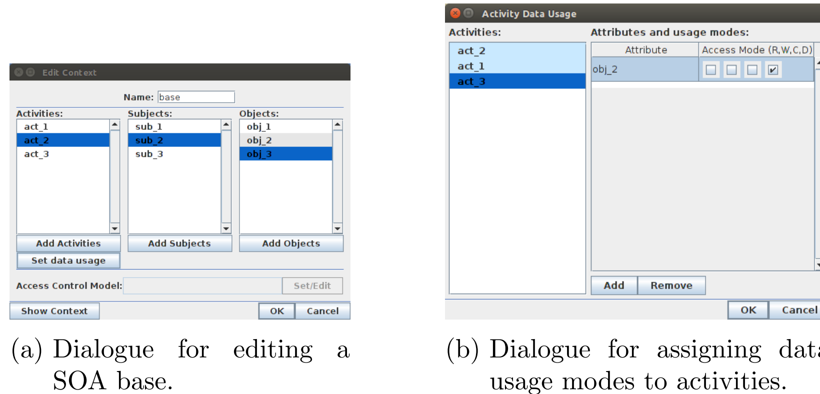
```
Context{
      name: base
activities: [act_2, act_1, act_3]
  subjects: [sub_2, sub_1, sub_3]
   objects: [obj_3, obj_2, obj_1]

activity data usage:
act_2: {obj_3=[CREATE, READ], obj_1=[WRITE]}
act_1: {obj_3=[CREATE, READ], obj_1=[WRITE]}
act_3: {obj_2=[DELETE]}

activity permissions:
act_2: [sub_1, sub_3]
act_1: [sub_1, sub_3]
act_3: [sub_2, sub_1, sub_3]

object permissions:
obj_3: [sub_2[] sub_1[CREATE, READ] sub_3[CREATE, READ] ]
obj_2: [sub_2[DELETE] sub_1[DELETE] sub_3[DELETE] ]
obj_1: [sub_2[] sub_1[WRITE] sub_3[WRITE] ]

execution authorization:
act_2: [sub_1, sub_3]
act_1: [sub_1, sub_3]
act_3: [sub_2, sub_1, sub_3]
}
```

At the beginning of the output the SOA base with the possible subjects, objects, and activities is shown. Under the heading *activity data usage*, the data usage modes of process activities are listed. The following headings *activity permissions*, *object permissions*, and *execution authorization* represent the underlying access control model, where the execution authorization depends on activity and object permissions. Subjects who are authorized to execute an activity also need permission to access data objects in the way the activity does.

To conveniently edit process contexts, SEWOL comes with a corresponding Java Swing dialogue. It can be created using the following command:

```
ProcessContextDialog.showDialog(null, context);
```

The resulting dialogue can be seen in Figure 4a, where the underlying SOA base can be edited. By selecting the button *Set data usage*, a new window opens (see Figure 4b), where data usage modes can be assigned to objects for activities.



(a) Dialogue for editing a SOA base.

(b) Dialogue for assigning data usage modes to activities.

Figure 4: Dialogue for editing process contexts in SEWOL.

4 Format of Workflow Traces

The data structure of workflow traces is defined in the package `log`. Basically, a log consists of multiple traces, which in turn contain a list of log entries. The entries represent a single activity execution and a trace brings several of these activity executions in order.

4.1 Log Entry

The smallest entity in a log is an event, which represents the execution of an activity. Additionally to the activity name it carries information like time of execution, originator (usually a person or system responsible for the activity execution), the originator's role, and an event type. The log entry is defined in the class `LogEntry` and the event types are defined in the enumeration `EventType`.

The following code defines two log entries:

```
LogEntry entryA = new LogEntry("act_1");
entryA.setEventType(EventType.start);
entryA.setOriginator("sub_1");
entryA.setRole("r1");
entryA.setTimestamp(new Date(1445405280000l));

System.out.println(entryA);

LogEntry entryB = new LogEntry("act_2");
entryB.setEventType(EventType.complete);
entryB.setOriginator("sub_1");
entryB.setRole("r1");
entryB.setTimestamp(new Date(499161600000l));

System.out.println(entryB);
```

This code sample produces the following output:

```
[10/21/2015 07:28:00|act_2|sub_1]
[10/26/1986 09:00:00|act_1|sub_1]
```

Log entries can be enriched with meta information in form of `DataAttributes`. Data attributes have a string key and a value of type `Object`. The following code adds event descriptions to log entries:

```
entryA.addMetaAttribute(new DataAttribute(
    "desc",
    "Marty McFly hits the road to the year 1985." ));
entryB.addMetaAttribute(new DataAttribute(
    "desc",
    "Marty McFly arrives at the year 1985." ));
```

To prevent editing of fields of the log entry, they can be locked. If a setter gets called for a locked field the method throws a `LockingException`. The following code locks the log entry's activity field:

```
entryB.lockField(EntryField.ACTIVITY, "arbitrary reason");
System.out.println(entryB.isFieldLocked(EntryField.ACTIVITY));   // true
entryB.setActivity("act_3");   // throws exception
```

The class `LogEntryUtils` contains the method `lockFieldForEntries` which locks a specified field in a list of log entries. Additionally, it contains some convenient methods to retrieve only entries with a particular activity or filtered by their locking status. It is also possible to group log entries according to their activities or cluster originators according to the activities they are responsible for.

The class `DULogEntry` is a subclass of `LogEntry` and complements it by a list of data attributes which are affected during the execution of the logged activity (data usage). Each data attribute gets assigned a set of data usage clearances, represented by the enumeration `DataUsage`. The following code creates a new `DULogEntry` object and specifies a new data attribute:

```
DULogEntry entryC = new DULogEntry("act_3");
entryC.setEventType(EventType.start);
entryC.setOriginator("sub_1");
entryC.setRole("r1");
entryC.setTimestamp(new Date(499134120000l));
entryC.addMetaAttribute(new DataAttribute("desc",
    "Marty McFly hits the road to the year 2015."));
entryC.addDataUsage(
    new DataAttribute("fluxcompensatorsettings", settings),
    DataUsage.READ);
```

4.2 Log Trace

A log trace is the result of one process execution. In these terms each trace refers to exactly one process instance/case. It is a container for multiple log entries and is represented as

an ordered list. The log traces can be complemented by case numbers by which they can be identified. In the SEWOL framework log traces are defined using the class `LogTrace`. The following code creates log traces for the formerly defined log entries. Since the type of log entries must be set using Java Generics, we consider all log entries being of the type `LogEntry`.

```
LogTrace<LogEntry> traceA = new LogTrace<LogEntry>(2);
traceA.addEntry(entryA);
traceA.addEntry(entryB);

LogTrace<LogEntry> traceB = new LogTrace<LogEntry>(1);
traceB.addEntry(entryC);

System.out.println(traceA);
System.out.println(traceB);
```

The corresponding output is the following:

```
[[10/21/2015 07:28:00|act_1|sub_1], [10/26/1985 09:00:00|act_2|sub_1]]
[[10/26/1985 01:22:00|act_3|sub_1]]
```

For the quick creation of log traces the class `LogTraceUtils` contains some static methods which create log traces by a sequence of activity names:

```
LogTrace<LogEntry> traceC = LogTraceUtils.createTraceFromActivities(
    3, "act_4", "act_5", "act_6", "act_7", "act_8");
System.out.println(traceC);
```

The traces only contain the activity name. The corresponding output looks like this:

```
[[-|act_4|null], [-|act_5|null], [-|act_6|null], [-|act_7|null], [-|act_8|null]]
```

4.3 Log

Finally, a log combines multiple instances of one workflow in an ordered list. It is represented by the class `Log` and additionally contains an object of `LogSummary`, which contains meta information about the log traces like the distinct sets of activities, subjects, and roles.

The following code creates a log and adds the formerly defined log traces:

```
Log<LogEntry> log = new Log<LogEntry>();
log.addTrace(traceA);
log.addTrace(traceB);
log.addTrace(traceC);
```

The summary now contains some meta information of the log traces and their entries:

```
System.out.println(log.getSummary().getActivities());
// [act_3, act_2, act_1, act_4, act_5, act_6, act_7, act_8]
System.out.println(log.getSummary().getOriginators());
// [null, sub_1]
System.out.println(log.getSummary().getRoles());
// [null, r1]
System.out.println(log.getSummary().getAverageTraceLength());
// 2.6666666666666665
```

5 Serializing & Parsing Logs

The SEWOL framework already comes with functionalities to read logs from other frameworks or serialize and use them with other tools.

Serializing Logs

The serialization classes for logs are located in the package `writer`. The class `LogWriter` takes a path to the output file and a subclass of `AbstractLogFormat` to define the target file format as parameters. Currently SEWOL supports the export of logs in the MXML and plain (whitespace-separated lists of activities) file formats.

The following code serializes the formerly defined traces in the MXML file format and writes them to the file `traces.mxml` under the path `/path/`:

```
LogWriter w = new LogWriter(LogFormatFactory.MXML(), "/path/traces");
w.writeTrace(traceA);
w.writeTrace(traceB);
w.writeTrace(traceC);
w.closeFile();
```

Parsing Logs

Logs can also be imported using SEWOL's built-in parser. It currently supports the XES, Petrify, and plain file formats. Since some file formats allow multiple logs in a single file, the parser returns a list of lists of log traces. All parser classes can be found in the package `parser`. The class `LogParser` "guesses" the file format by the file extension.

The following code parses the file `logs.xes` under the path `/path/`:

```
List<List<LogTrace<LogEntry>>> logs = LogParser.parse("/path/logs.xes");
```