Tele Assistance System (TAS) v. 1.61 - Developers' Guide.

Introduction

The Tele Assistance System (TAS) is an exemplar system developed for research on self-adaptation in the domain of Service-Based Systems (SBS). It comes with pre-defined scenarios for comparing the effectiveness of different self-adaptation solutions. Other researchers can easily exploit the underlying service platform, reusable components, and development method we devised for the TAS to speed up research and development in the field of self-adaptive and service-based system. This document provides researchers and developers a quick start guide to download the TAS source code, understand, use and extend if needed, the provided services and features in their work.

Prerequisites

1. **Java:** TAS code is written in Java, thus JavaSE-1.8 or the latest version is needed.
2. **TAS source code**: TAS source code can be downloaded at its homepage[1]. You should download the latest version, which is 1.61.
3. **Eclipse:** The TAS is developed using Eclipse, thus, we recommend to use Eclipse to setup your codebase and play with the tool. Instructions to setup a codebase for the TAS using Eclipse are as follows.

Codebase Setup

1. Download source code for the TAS version 1.61 which is archived in a zip file, *TAS.v1.61*.zip.
2. Un-archive the zip file. As a result of the un-archive, you will get a root directory or folder named "TAS.v1.61". The folder contains the following three sub-folders.

    a) ResearchServicePlatform
    b) TAS_gui
    c) TeleAssistanceSystem

3. The TAS exemplar is developed using Eclipse and is organized into three Eclipse projects that are mapped to the above folders, i.e., the folders contain all the Eclipse project files including source code and other resources required for the TAS. Thus, you can setup your own codebase to test, experiment, and maybe extend features of the TAS, by importing the source code to any IDE of your choice. However, we recommend using Eclipse to avoid path settings and other configuration related issues.
4. Using Eclipse, the TAS codebase can be setup by importing the TAS source code through different options, such as "Import" and "Open Projects from File System…" options under the "File" menu. However, a simple and quick way to do this is by switching to the TAS root directory (directory containing the downloaded TAS source code) as your Eclipse Workspace. This can be done

[1] https://people.cs.kuleuven.be/~danny.weyns/software/TAS/

https://lnu.se/en/research/searchresearch/adaptwise/
Contact: Danny Weyns (danny.weyns@lnu.se), Nadeem Abbas (nadeem.abbas@lnu.se)

by choosing "File → Switch Workspace → Other…" option, and selecting the TAS root directory as your workspace. Click the "Launch" button in the end. This will (re)launch the Eclipse with three projects that constitute the TAS exemplar system. A brief overview of the three projects that define TAS API is given below:

TAS Application Programming Interface (API)

a) ResearchServicePlatform

This project defines a Research Service Platform (ReSeP). The ReSep provides a set of high-level reusable components, such as probes, effectors, and atomic and composite service components, to speed up the engineering of new service-based self-adaptive software systems. The ReSeP provides the basis to the TAS. Most of the TAS services and components, such as the main Assistance Service, probes, effectors, are defined by selecting and specializing (reusing) the ReSeP components. A few of the important ReSeP classes and interfaces are briefly described as follows:

- **`service.adaptation.probes.interfaces package`**

  The package is composed of two interfaces:

  i. **WorkflowProbeInterface:** The WorkflowProbeInterface defines a public interface, a set of methods, that can be used to monitor events related to a service execution workflow, for instance, start, stop, service invocation, service not found, etc.

  ii. **CostProbeInterface:** The CostProbeInterface defines a public interface, a set of methods that can be used to monitor a service cost.

  The class tas.services.assistance.AssistanceServiceCostProbe defined in the project TeleAssistanceSystem implements both of the above two interfaces.

- **`service.adaptation.probes package`**

  The package is composed of following three classes: **AbstractProbe<E>, CostProbe, and WorkflowProbe.** The three classes implement the subject part of the observer pattern.

  i. **AbstractProbe<E>** class defines a generic subject type that maintains a list of subscribers (observers) for various kinds of information, such as service cost, delay data, workflow events, etc. It provides public methods to register/unregister subscribers.

  ii. **CostProbe** class extends the AbstractProbe<CostProbeInterface> class with a method that notifies subscribers (observer components that want to monitor) about cost of a service.

  i. **WorkflowProbe** class extends the AbstractProbe<WorkflowProbeInterface > class with a method that notifies subscribers (observer components that want to monitor) different workflow events such as start, stop/end, of a workflow and events related to service operations.

- **`service.adaptation.effectors package`**

  The package is composed of the following four classes:

  i. **AbstractEffector** class encapsulates a reference to composite service that need to be adapted and provides a getter to retrieve a reference to the composite service. There are no actual effector operations defined in this class.

ii. **CacheEffector** defines a set of effector operations to perform cache-related adaptive actions, such as remove/refresh (reload) atomic services, upon a composite service (service-based system).
iii. **WorkflowEffector** class extends the AbstractEffector class. It defines a set of effector operations (methods), such as updateWorkflow, removeService, updateServiceDescription, stopRetrying, etc. to perform adaptive actions on a composite service.
iv. **ConfigurationEffector** extends the AbstractEffector class. It defines a set of effector operations to perform configuration related adaptive actions such as updating *service timeout* and *retry* parameters in case of a service failure.

Below are a few of the other important classes in the ResearchServicePlatform project.

i. **service.auxiliary.AbstractService:** As the name indicates this is an abstract class that defines basic attributes and methods, such as startService, stopSertvice, register, unRegister, invokeOperation, etc., needed for both composite and atomic services.
ii. **service.composite.CompositeService** class extends the above described AbstractService class and provides an abstraction to create composite services. It defines fundamental attributes and functionality (methods) required for a composite service.
iii. **service.atomic.AtomicService** class extends the above described AbstractService class and provides an abstraction to create atomic services. It defines fundamental attributes and functionality (methods) required for an atomic service.

b) TeleAssistanceSystem

This project defines the Tele Assistance Service (TAS) system by reusing components from the ReSeP and adding additional components needed for self-adaptation, such as an adaptation engine (managing system). The TAS is a composite service composed of three atomic services: 1) Medical Analysis, 2) Alarm, and 3) Drug. Details about the TAS are given at the TAS homepage. A few of the important TAS classes and interfaces are briefly described as follows:

- **`tas.services.assistance package`**

The package consists of the following two classes:

i. **AssistanceService** defines the main Tele Assistance Service. It extends the abstract CompositeService class from the ReSeP platform. It defines two methods, getVitalParameters and pickTask, which respectively simulate taking periodical measurement of vital parameters of a patient and selecting among Medical Analysis, Alarm, and Drug services.
ii. **AssistanceServiceCostProbe** class implements the CostProbeInterface and the WorkflowProbeInterface interfaces from the ReSeP platform. It provides definitions for the cost and workflow probe operations. For example, the serviceCost method probes cost of a specific service and the operation invoked on that service. The service cost is added to the total cost of the TAS services workflow.

- **`tas.services.medical package`**

The package contains only one class, MedicalAnalysisService. The MedicalAnalysisService extends the AtomicService class, i.e., it is an atomic service. It simulates the Medical Analysis Service and defines a method, analyzeData, which

simulates functionality to analyze the vital parameters of a patient and to invoke either Drug or Alarm service.

- **`tas.services.drug package`**
  The package contains DrugService class that extends the AtomicService class, i.e., it is an atomic service. It simulates the Drug Service and defines change drug and change doses operations.

- **`tas.services.alarm package`**
  The package contains AlarmService class that extends the AtomicService class, i.e., it is an atomic service. It simulates the Alarm Service and defines triggerAlarm method which simulates the emergency alarm.

- **`tas.services.qos package`**
  The package consists of the following three classes that specify the Quality of Service requirements (QoS) for the TAS services.
  i. **MinCostQoS** class defines "Cost" quality attribute that requires invoking TAS services with minimum cumulative cost. It defines the applyQoSRequirement method that as shown in the code snippet below uses service descriptions to select a service instance with minimum cost.

```
for (int i = 0; i < serviceDescriptions.size(); i++) {
   properties = serviceDescriptions.get(i).getCustomProperties();
   if (properties.containsKey("Cost")){
         cost = (double) properties.get("Cost");
         if (cost < minCost){
               minCost = cost;
               index = i;
         }
   }
}
```

  ii. **ReliabilityQoS** class defines "Reliability" quality attribute that requires invoking TAS services with minimum failure rate. It defines the applyQoSRequirement method that inspects all available services and selects service instances with minimum failure rate.

  iii. **PreferredQoS** class defines "Preference" quality attribute that requires invoking TAS services that are preferred by a client, i.e., only specific service instances. It defines the applyQoSRequirement method that selects only those service instances that are marked (preferred) by a client.

- **`tas.services.adaptation package`**
  The package consists of following four classes:

  i. **AdaptationEngine** defines a basic interface of an adaptation engine (managing system).

  ii. **DefualtAdaptationEngine** implements the AdaptationEngine interface. It just simulates an adaptation engine and does not do any adaptive actions.

  iii. **SimpleAdaptationEngine** implements the AdaptationEngine interface. It defines a basic adaptation engine that handles and adapts the TAS exemplar when a service fails or is not found. The failed or not found services or probed using a

workflow probe and are adapted with the help of an effector. As shown in the following code snippet, constructor of the SimpleAdaptationEngine connects a workflow probe and an effector to the adaptation engine.

```
public SimpleAdaptationEngine(AssistanceService
assistanceService) {
    this.assistanceService = assistanceService;
    myProbe = new MyProbe();
    myProbe.connect(this);
    myEffector = new WorkflowEffector(assistanceService);
}
```

The SimpleAdaptationEngine handles a service failure by removing failed service from the service registry so that some other available service instance can be invoked.

```
public void handleServiceFailure(ServiceDescription service,
String opName){
         this.myEffector.removeService(service);
}
```

As shown in the above code snippet, the SimpleAdaptationEngine uses an effector instance, `myEffector` of type `WorkflowEffector`, to remove the failed service. After removing the failed service, if there exists no other instance of the service that has failed, service not found event would be triggered. This event is detected by the workflow probe linked to the adaptation engine. As shown in the code snippet below, the adaptation engine handles such events by reregistering all the previously registered services of a specific service type.

```
public void handleServiceNotFound(String serviceType, String
opName)
{
   myEffector.refreshAllServices(serviceType, opName);
}
```

- **`tas.start package`**
  This package contains only one class, **TASStart**, which prepares and makes the TAS services ready for execution. The TASStart class has a constructor that invokes the *initializeTAS* method. The *initializeTAS* method creates a service registry, instantiates, initializes, and registers the TAS services. It first creates and initializes atomic services (Medical Analysis, Drug, and Alarm). The following code snippet shows how an atomic service is instantiated, how service properties, for instance, *cost*, *failure rate*, are set, and how a service instance is registered.

```
// Create and register various instances of the Alarm Service
public void setupAlarmServices() {
    // ALarm Service 1
    alarm1 = new AlarmService("AlarmService1", "service.alarmService1");
    alarm1.getServiceDescription().getCustomProperties().put("Cost", 4.0);
    alarm1.getServiceDescription().getCustomProperties().put("preferred", true);
    alarm1.getServiceDescription().setOperationCost("triggerAlarm", 4.0);
    alarm1.getServiceDescription().getCustomProperties().put("FailureRate", 0.11);
    alarm1.addServiceProfile(new ServiceFailureProfile(0.11));
    alarm1.startService();
    alarm1.register();

    // ALarm Service 2
    ...
```

```
    // ALarm Service 2
    ...

}
```

After registering the atomic services, the *initializeTAS* method then instantiates, initializes, and register the composite Tele Assistance Service. Note the composite Tele Assistance Service is implemented by the class *tas.services.assistance.AssistanceService*. The following code snippet shows how an instance of the composite AssistanceService is created, and how atomic services and QoS requirements are linked.

```
// Assistance Service. Workflow is provided by TAS_gui through executeWorkflow
method
assistanceService = new AssistanceService("TeleAssistanceService",
                "service.assistance", "resources/TeleAssistanceWorkflow.txt");

this.addAllServices(alarm1,alarm2,alarm3,medicalAnalysis1,medicalAnalysis2,
                            medicalAnalysis3,drugService);

//add QoS Requirements for the TAS services
assistanceService.addQosRequirement("ReliabilityQoS", new ReliabilityQoS());
assistanceService.addQosRequirement("PreferencesQoS", new PreferencesQoS());
assistanceService.addQosRequirement("CostQoS", new MinCostQoS());
```

The TAS exemplar requires probes and effectors to adapt the services at runtime. For this purpose, the following code snippet shows how probes and effectors can be instrumented.

```
//probes instrumentation
monitor = new AssistanceServiceCostProbe();
assistanceService.getCostProbe().register(monitor);
assistanceService.getWorkflowProbe().register(monitor);

//effectors instrumentation
workflowEffector = new WorkflowEffector(assistanceService);
```

c) TAS_gui

As the name indicates, this project defines a Graphical User Interface (GUI) of the TAS exemplar system. It uses JavaFX[2] platform to define the GUI elements and develops the TAS system as a JavaFX application. The JavaFX is a set of graphics and media packages to design, develop, test and deploy rich client applications. This project has not much to do with the TAS core application logic. It mainly deals with presentation logic and few application logic concerns such as logging. A few of the important packages to look in this project include *application, application.view, and application.log*.

The package *application.view* contains FXML files. The FXML is a scripting language and is used to develop presentation aspects of the user interface. The use of the FXML allows separating the presentation logic from application logic of the TAS exemplar. The application logic is developed using Java as described in the first two projects of the TAS system.

The *application.log* package contains Java classes that provide functionality to log the application events and report these events. Examples of the events that are logged include service start/stop, service failure, and completion of a workflow.

[2] https://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm

The *application* package defines an application class, MainGui. The MainGui extends the javafx.application.Application class and thus provides an entry point to launch the TAS system as a JavaFX application. You should run this class to execute the TAS application. This can be done in Eclipse, for example, by right clicking on the MainGui.java and selecting "Run as" → "Java Application".
The main entry point for JavaFX applications is the Application.start method that sets the primary stage for the application and other GUI elements. The MainGUI class has a *main* method, which invokes the launch method. The launch method then invokes the start method to launch the application.

Limitations
The TAS system is yet in its early phases of development and the developers already know a couple of its limitations that are mainly related to the GUI features. These limitations are listed below. The system is offered as an open-source educational resource, thus, we encourage users to contribute towards further development, for instance by addressing current limitations, suggesting improvements and adding new features.

1. For available service instances, e.g. MedicalService1, MedicalService2, and MedicalService3, changes made to Service Profiles, e.g. ServiceFailureProfile, using the GUI at runtime are not affected unless we change profiles in code. For example, if we set a failure rate of a MedicalService1 to be 0.02, the lowest among all the available medical services and run the ReliabilityQoS, the TAS system does not select the MedicalService1, rather it keeps invoking MedicalService2 with a failure rate of 0.07.
2. The TAS system fails to generate performance graph in the TAS Experimentation panel.

Contact: Danny Weyns (danny.weyns@lnu.se), Nadeem Abbas (nadeem.abbas@lnu.se)