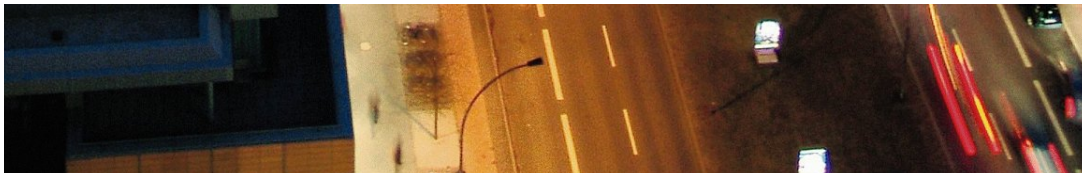
Sitraffic sX DWS

Developer Guide V1.0
A001

Intelligent Traffic Systems

SIEMENS



Contents

List of Figures

References

No.	Document	Content	Where to find?
[1]	Sitraffic sX DWS Installation Guide		Sitraffic_sX_DWS_Installation_Guide_en.pdf
[2]	Controller GUI User Guide		<installdir>\dws\doc\Sitraffic_sX_WebGui_Manual_en.pdf
[3]	Sitraffic SmartCore User Guide		<installdir>\dws\doc\Sitraffic_SmartCore_Manual_de.pdf
[4]	Sitraffic Control Model Description		<installdir>\dws\doc\Sitraffic_sX_Control_Model_en.pdf
[5]	RILSA	Guidelines for Traffic Signals English Version of RiLSA with minor Modifications, FSGV-Nr.321/S, 1992/2003	http://www.fsgv-verlag.de.
[6]	C-Control Javadoc	Default location	C:\dws\doc\ccontrolclientlib-api\index.html
[7]	OCIT-O Documentation	German only	http://www.ocit.org

1. Introduction

1.1. Topic and Scope

This document introduces the Sitraffic Development Environment Workstation (DWS). Purpose of the DWS is to provide a suitable environment for designing and testing traffic actuation and control center applications for the Siemens Sitraffic sX traffic signal controller. This document is mainly use-case driven as opposed to providing a comprehensive reference manual. Therefore only the most basic details of the discussed workflow-examples are given. For a more comprehensive description of the Sitraffic sX platform such as available functionality and underlying traffic engineering terms please refer to the list of reference documents. Readers already familiar with established controller families such as C800/900 might recognize the adoption of some concepts.

Please note that the system is still under construction. The very details of functionality are still changing as the platform development progresses. This especially concerns the interfaces.

Central component of this development environment is a virtualized Sitraffic sX traffic controller that servers as a substitute for the real one. The virtualized software is in major parts identical to that one of the real device. However some differences exist which will be pointed out where necessary.

1.2. Outline

Starting point of the present document is an already existing basic installation of the DWS software components. For detailed instructions please refer to the Installation Guide [1].
The structure of this document follows the sequence of steps needed for configuring and using the development environment.

- Chapter 2 gives a brief overview of the Developer Workstation Environment. The basic components as well as the employed communication techniques for connecting the components are described

- Chapter 3 describes how the example projects for developing control center- and traffic actuated control applications are set up.

- Chapter 4 is about the AppStarter design which is the framework for developing applications such as traffic actuated control or control center applications for the Sitraffic sX
- Chapter 5 focuses on running the two example projects. For several use-cases of control center- and traffic actuated control, step-by-step instructions are given

2. The DWS Environment

2.1. Components

The Sitraffic sX Developer Workstation consists of several components. All of them run on the same Windows 7 machine (see Figure 1, Developer PC Win7). Probably the most basic one is the virtualized Sitraffic sX traffic controller. A virtual image containing the relevant software parts of the real device run on a VMWare Player. That controller provides a Control GUI that can be easily accessed using a HTML5 compatible web browser. Configuration files can be up- and downloaded into/from the controller via that GUI. These files are generated and edited using the configuration tool Sitraffic SmartCore [3]. Purpose of the whole Sitraffic sX Development Environment is to create and test applications for the Sitraffic sX platform using the C-Control interface. In the present development scenario these applications run outside of the virtualized controller. The applications are created, modified and run employing the Eclipse IDE. This is quite different from the regular deployment scenario where applications run on the Sitraffic sX controller directly.

The following diagram illustrates the development scenario of deploying local traffic actuation- and traffic control center applications (see Fig.1).

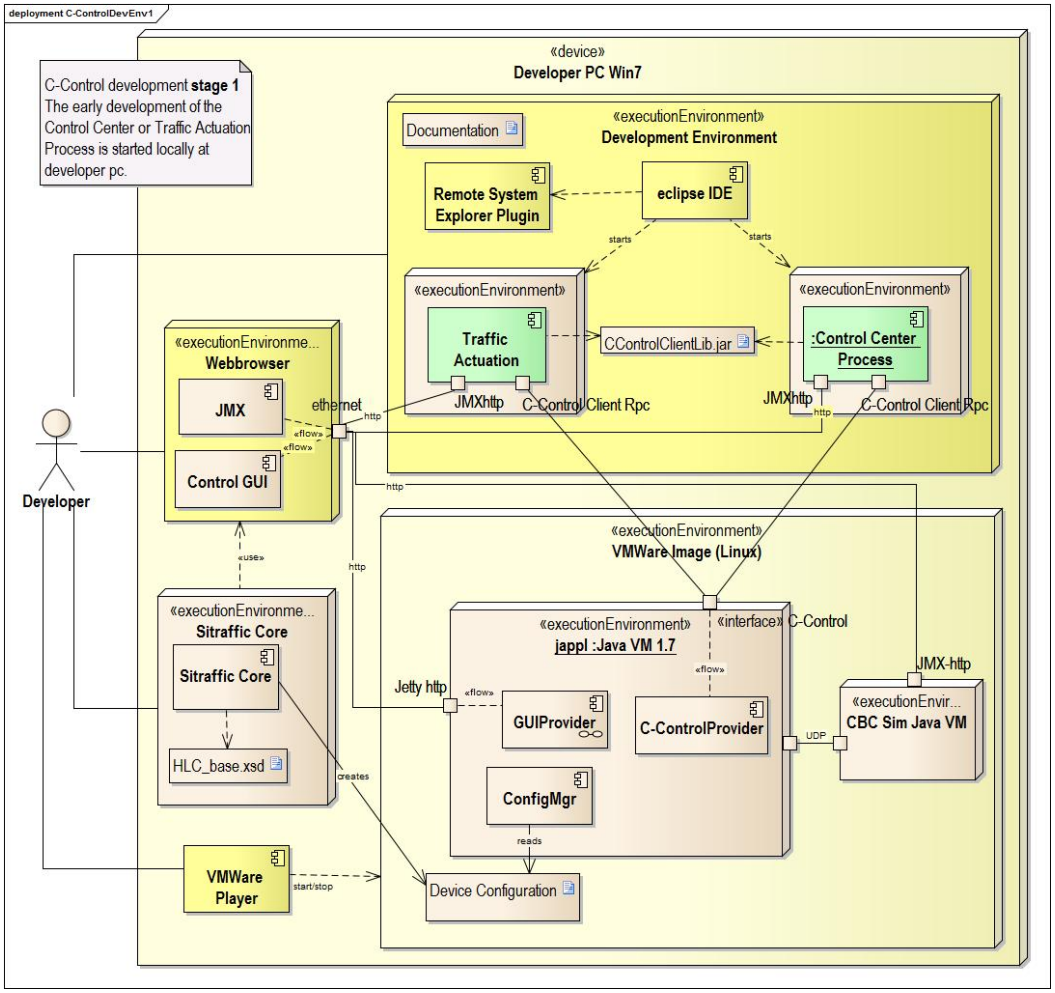


Figure 1: DWS deployment diagram

Overall, the Developer Workstation installation consists of the following components:

- Developer Workstation
  The DWS, here called "Developer PC, Win7" is the developer PC with installed Microsoft Windows 7 as operating system. On this device, all other components are located.

- VMware Image
  The Sitraffic sX VMware image contains the virtualized Sitraffic sX controller. It includes the main firmware of the Sitraffic sX controller and the so called CBC simulation which is the simulation of the signal monitoring unit and which substitutes real hardware as far as needed.

- Sitraffic SmartCore
  Sitraffic SmartCore is the new tool for planning traffic signal controlled intersections. With this tool, you are able to generate traffic configuration files, save them locally on your DWS or deploy and activate them on the virtualized Sitraffic sX controller. Additionally several default configurations are already included in the DWS delivery.

- Web Browser
  The web browser has to support HTML 5. The usage of the Mozilla Firefox is recommended. See the installation guide [1] for more details.

- Controller GUI
  The Controller GUI is the http access to the virtualized Sitraffic sX device. With this web GUI access, you are able to monitor and control the virtualized Sitraffic sX device.

- JMX
  The JMX GUI (Java Management Extension) is the direct access to the Java components via http inside the virtualized Sitraffic sX device or the processes for traffic actuation or control center processes

- Development Environment
  The development environment consists of the eclipse integrated development environment (IDE) to administrate, develop and test the Java example projects of control center and a traffic actuation processes. The example implementations use the C-Control interface realized as RPC (remote procedure call) for inter-process communication with the virtualized Sitraffic sX device. The eclipse "Remote System Explorer Plugin", which is already included in the installed eclipse IDE of DWS, helps you easily to connect to the virtualized Sitraffic sX device.

- Traffic Actuation and Control Center Processes
  As a developer you can implement your own process for traffic actuation or connection to your control center using your own protocols. These client processes can be deployed and started from eclipse on your development environment or can be directly deployed to the virtualized Sitraffic sX device.

- CControlClientLib.jar
  This library includes the RPC generated Java classes, the RPC stub implementations and some additional helper classes.

2.2. Introduction to Controller Application Development

The aim is to implement your own C-Control Client application, which can on the one hand access the controller logic and on the other hand your own remote traffic control center using its own specific communication protocol. C-Control Client applications can be either a traffic actuation or traffic control center connection application. A C-Control Client application is an own process and the whole Controller logic is an own process running in the same controller. The remote procedure calling technology (RPC) is used in order to handle the communication between these processes in the controller.

The traffic actuation- and the control center processes both connect to the controller firmware- process via the C-Control interface. For ease of use the java library "CControlClientLib" is provided as part of the development environment.

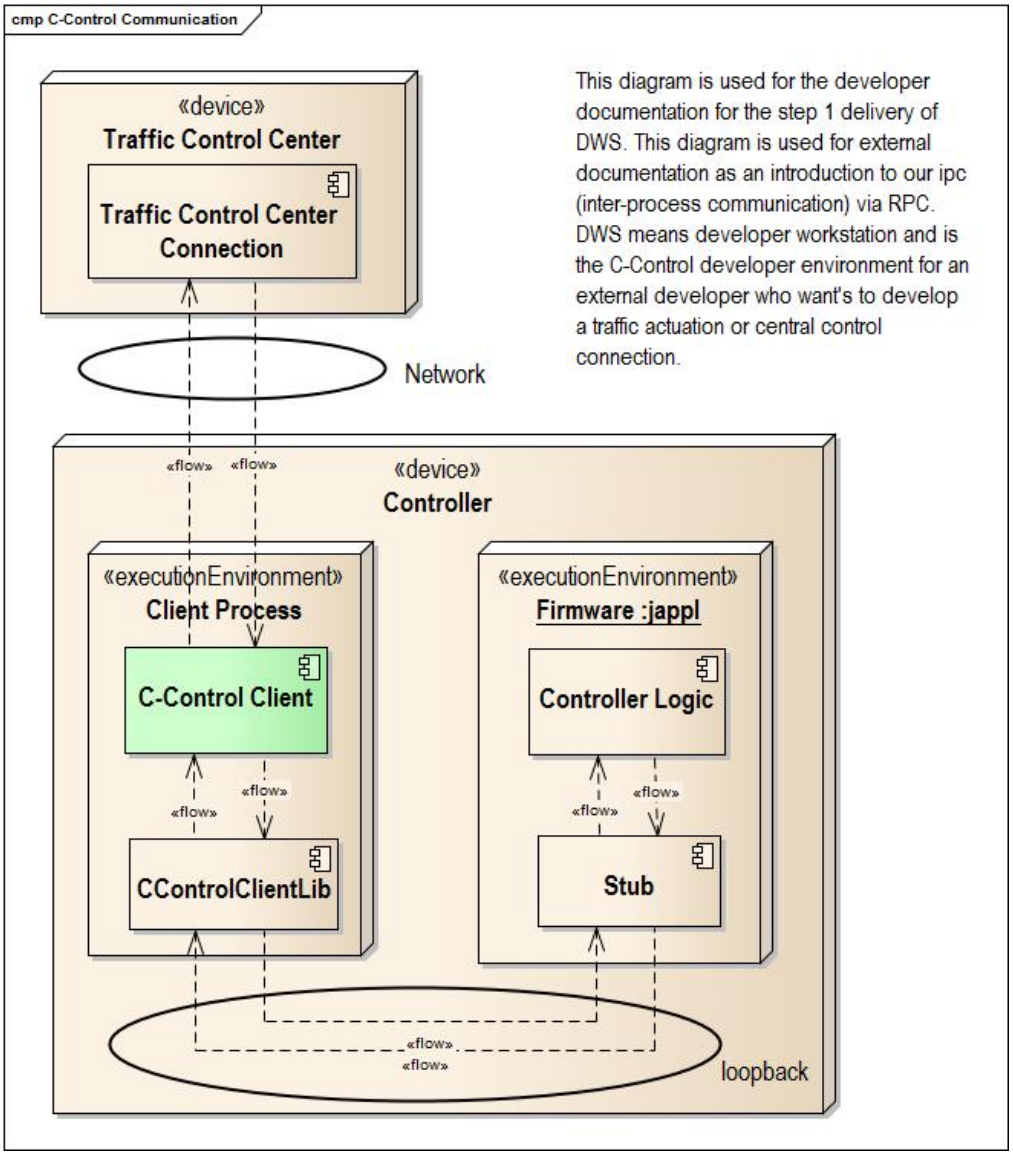


Figure 2: Inter-Process Communication via Remote Procedure Calls

RPC uses a request-and-reply communication model. The client procedure sends request messages to the server procedure, which returns reply messages. The processes communicate by means of two stubs, one for each process. The stub for the C-Control client is called “CControlClientLib”, which is the communication interface that wraps the RPC protocol and specifies how messages are constructed and exchanged. The C-Control javadoc includes the API (application programming interface) description of the C-Control interface.

3. Getting Started

This chapter gives an overview over the usage of the development environment and the example Java projects.

3.1. Preconditions

The main precondition is that all the software of the development environment and all the tools and the example Java projects are successfully installed on your host computer. Additionally the virtualized Sitraffic sX controller should be reachable through the network by your host computer and vice versa. You can find more information in the Sitraffic sX DWS installation guide [1].

3.2. File Structure of DWS Example Projects

3.2.1. The Appfs Project

The example Java projects for the development of a traffic actuation process and the development of a traffic control center process need the base Java project called appfs (stands for application file system). The appfs project represents the runtime environment for the Java example projects on the target platform and is located under your DWS installation directory. It includes one jappl subdirectory (jappl stands for Java applications) which itself includes further subdirectories, which are used during the runtime. The default path to that directory structure is C:\dws\appfs\jappl.

Here a short overview of the content of these subdirectories:


data

This subdirectory includes application specific data. Here you can persist data of your traffic actuation or traffic control center application if necessary. It is recommended to use the following subdirectories for the corresponding processes: jappl\data\extTrafficActuation and ...\extTrafficControlCenter


extern

This subdirectory includes the 3rd party Java libraries which are referenced by the classpath during the runtime and compile time of the example projects. The log4j.jar library is e.g. used for the logging.


internal_releases

This subdirectory includes the libraries CControlClientLib.jar for the C-Control interface connection and the platform.jar for lifecycle control of your application.


log

This subdirectory includes the log files created during the execution of your applications.

3.2.2. The Application Example Projects

Based on the Appfs project are the two application example projects. The projects are named ControlCenterIfProcessExample and the traffic actuation process project is named TrafficActuationControlExample. Both Java projects are located under your DWS installation directory. The default path is C:\dws\ControlCenterIfProcessExample and C:\dws\TrafficActuationControlExample respectively.

The projects have several subdirectories:


.launches

This subdirectory includes a default Eclipse start configuration in order to run the control center process.


bin

This subdirectory is the destination directory where the java compiler stores the compiled java classes of the project.


extern

This subdirectory includes the 3rd party libraries which are needed during compile time but not during runtime. Examples are the JUnit library or the redline library to generate rpm's for the target machine. The remaining 3rd party libraries needed during compile time are referenced from the appfs base project, because they are also needed during runtime.


jappl

This subdirectory is used as the local runtime environment as an extension to the appfs project. The libraries of your example application are build and copied in the subdirectory internal_releases\c10. The configuration files of the example application are located in the subdirectory called cfg. Both subdirectories internal_releases and cfg are packaged into a rpm package which then can be installed on the target machine.


src

This subdirectory includes the example source code of the control center or the traffic actuation application.

3.3. Working with the Example Projects

3.3.1. Importing the Projects into Eclipse

In the next step the example projects need to be imported into the Eclipse IDE. This section describes the setup and handling of your eclipse workspace environment. From here, you can run and extend the example projects for traffic actuation and control center process.

1. If you start the Eclipse the first time, you are asked to choose a path for the workspace. That path is already automatically set relative to your DWS installation root path. We recommend to stay with that choice and just proceeding by clicking “Ok”.

   If you are already using Eclipse:

   The DWS installation provides an Eclipse on its own. If you stay with the default workspace location there is no known interference with preexisting Eclipse installations. Please note that the desktop icon is named “Eclipse Sitraffic sX” for indicating which is which.

2. You have to import the appfs project first, because it is referenced by the traffic actuation and control center example projects. Select „File -> Import...“ in order to open the import dialog. Then select the entry “General” -> „Existing Projects into Workspace“ and click on Next (see Figure 3)

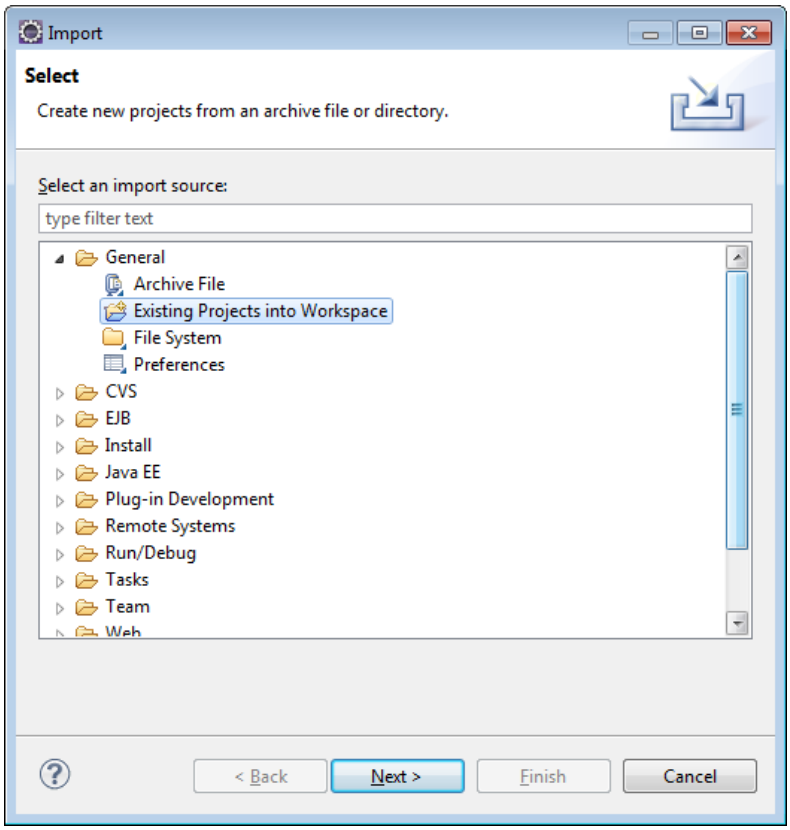


Figure 3: Eclipse Project import dialog 1

3. In the next dialog check the „Select root directory“ and click on “Browse” in order to navigate to the appfs project directory and to select it. If you have chosen the default installation path, the appfs project is located at “C:\dws\appfs”.

4. If the project is found correctly, it is listed and checked in the Projects pane (see Figure 4). With clicking on “Finish” the appfs Eclipse project will be imported and displayed in your Eclipse development environment.

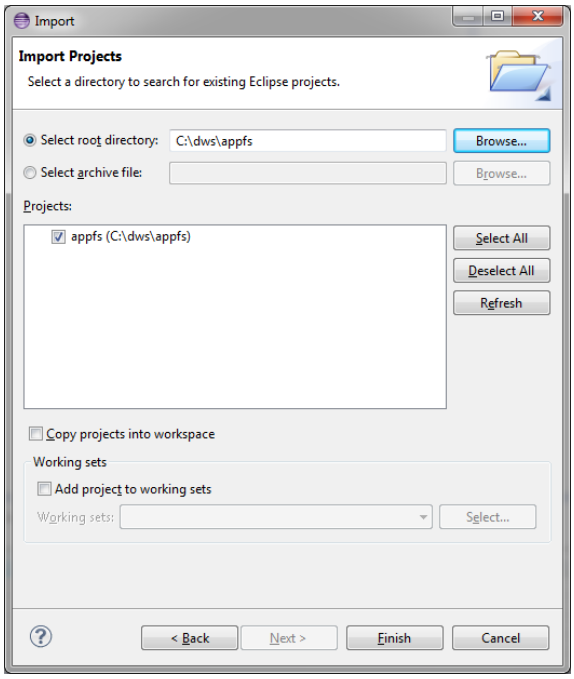


Figure 4: Eclipse Project import dialog 2

5. You have to repeat steps 2 to 4 in order to import the two example projects for control center and traffic actuation applications. If you have chosen the default installation path, they are located at “C:\dws\TrafficActuationControlExample” and “C:\dws\ControlCenterIfProcessExample” respectively.

3.3.2. Using the Code Formatter Profile

Programming styles deal with appearance of source code, with the goal of requiring less human cognitive effort to extract information about the program. Eclipse does the formatting of your source code automatically, so that you can concentrate on naming, logic and higher techniques. If you use the Siemens ITS programming style, it saves time if you need our support. The ITS Siemens code formatter definitions file is located at “C:\dws\eclipse\ ItsFormaterSettings.xml ”. You can import it by following these steps in eclipse:

1. Open the project properties from the menu “Project” -> “Properties”

2. Go to “Java Code Style” -> “Formatter” and click on “Configure Workspace Settings...”

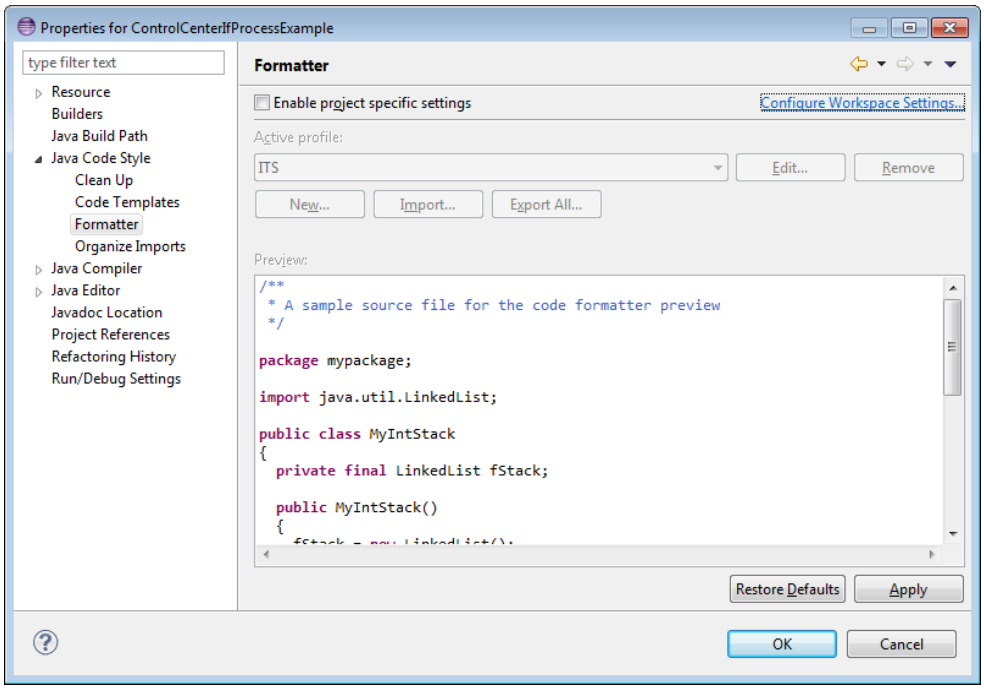


Figure 5: Properties of ControlCenter

3. Import the File “ItsFormaterSettings.xml”

3.3.3. Build the Example Projects

This chapter covers the build of the example projects using the Ant build tool from the Apache Software Foundation. For more information on Ant, see the Apache Ant manual documentation (http://ant.apache.org/manual/index.html).

The example projects ship with two ant build files:

- build.xml
  This build file is auto-created via eclipse export functionality. This file should be generated again if e.g. the classpath of the project is changed.

- build-custom.xml
  This build file is manually written and depends on the build.xml build

file. It just additionally does the packaging of the example project into a jar file.

For auto-generating or running Ant build files in eclipse, please refer to the eclipse help documentation found in http://www.eclipse.org/.

3.3.4. Running the examples

In order to run the examples you don't need to build the projects via Ant. There are defined eclipse run configuration.

The two application examples shipped with this development environment can be run and modified by using the Eclipse IDE. Once the example projects have been imported, the process handling for the control center as well as for the traffic actuation application are the same. You can start the process of each example by choosing: "Package Explorer -> Debug As -> Debug Configurations". Alternatively the examples also can be run the regular way by selecting "Run As -> Run Configurations" (see Figure 6).

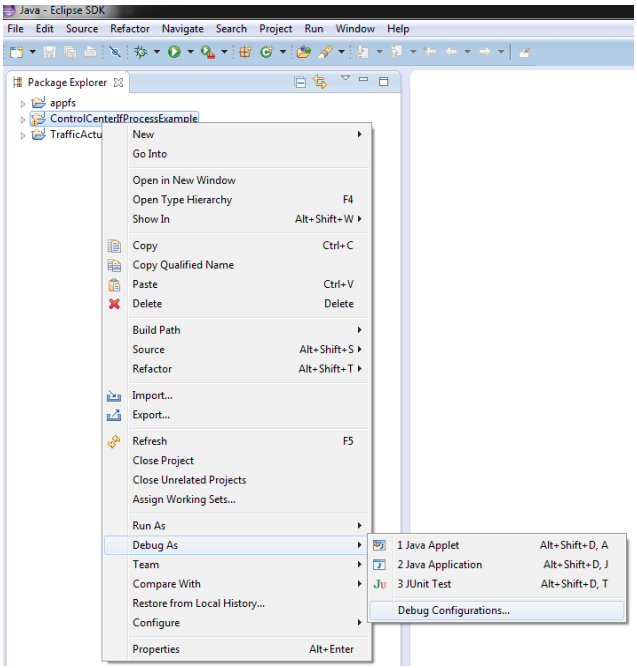


Figure 6: Eclipse Run Debug Configuration

Which run mode you chose depends on your very use case. Chose "Run As" if you primarily want to provide the virtualized Sitraffic sX controller with the connection to a running java process. On the other hand, if you would like to analyze and modify the source code of the example process, chose "Debug As". In this case, you can utilize the integrated debugger of Eclipse. Please note that currently there is no synchronization of the virtualized Sitraffic sX and the external Java processes. The virtualized Sitraffic sX controller is not stopped if you reach a breakpoint. Therefore the signalization as well the operating state might still change while you are debugging.

After choosing the runmode for your application you can customize the employed configuration. The example applications both come with suitable configurations, e.g. “Java Application -> TrafficActuationControlExample” (see Figure 7).

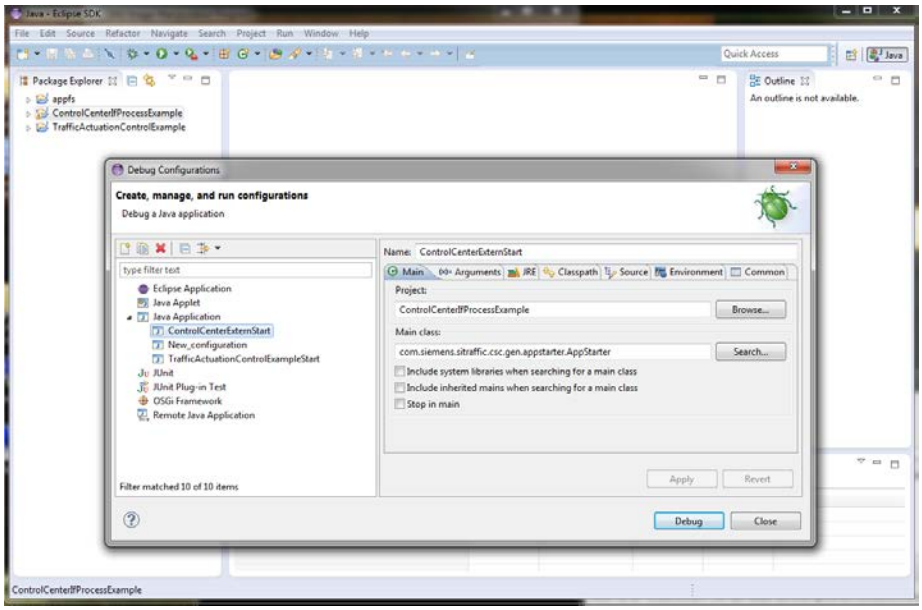


Figure 7: Eclipse Run Debug Configuration Editor

Depending on the launched configuration, various options are available. We recommend to use the default settings and proceeding by pushing the Debug-Button. For selecting the predefined example configurations please selected (left side of window): “Java Application-> ControlCenterIfProcessExample or TrafficActuationControlExample”.

Independent from the chosen configuration for launching the example process, logging information is shown on the integrated Eclipse console. A running process is indicated by the red STOP-Button on the consoles frame.

3.3.5. Logging

The two example processes uses the Log4j API (application programming interface), which is the most popular logging package for Java and distributed under the Apache Software License. The homepage of Log4J is http://logging.apache.org/.

The log outputs of the example processes are saved in a separate file. Both projects define their own log file name and location in their Log4j configuration files. The configuration files are located in the following sub directory of the example Java projects: \jappl\cfg\log4j\

Logging offers several advantages. You can debug your application or analyze them by scanning the output afterwards, which takes less time than stepping through your program. However, please be aware that logging has also its drawbacks. It can slow down your application. Performance is a critical factor in embedded systems.

Therefore, the recommendation is to put and activate tracing to your code for critical places for productive use. If you want to debug your application, you can steer the output via the debug levels.

4. AppStarter Design

The example Java projects are implemented as so called AppStarter components. They have a defined init, startup and shutdown sequence. This chapter shall give just a short and quick overview of the AppStarter design.

The AppStarter implementation is included in the platform.jar package located in the c:\dws\appfs\jappl\internal_releases\csc\ directory of your DWS installation.

The main class is: com.siemens.sitraffic.csc.gen.appstarter.AppStarter

The main class can be used to do various application startup/shutdown steps which can be configured through a XML file. The classpath required to run the components specified within the AppStarter configuration must already be applied to the Java Virtual Machine (Java VM), which runs the AppStarter instance. The configuration file primarily specifies a list of components (classes which implement the AppStarterComponent interface). Each component is called during initialization process and shutdown process. An AppStarter component is responsible for a particular application part or library to be initialized/shutdown correctly (e.g. Log4J initialization).

4.1. Application Rootpath

The application rootpath specifies the directory of the root directory. It has to be available as a system property called "application.rootpath" and can be set during the startup of the server. This setting is mandatory. You can find this setting for the example java projects in their eclipse run configurations in the VM arguments definition.

4.2. Starting the AppStarter

Example invocation:

```
java.exe -cp <classpath> -Dapplication.rootpath=/jappl
com.siemens.sitraffic.csc.gen.appstarter.AppStarter cfg/Cfg.xml
```

To start the Sitraffic sX system you have to define:

- the classpath containing all external and internal libraries
- application.rootpath system property set with the -D java option
- invoke the main routine of the class AppStarter with an AppStarter configuration file as parameter

The startup sequence is divided into an initialization step and one or more startup steps. In the initialization step, the AppStarter instantiates all configured AppStarterComponents and calls their init() method. In this method, a component should check the environment it needs, and the configured AppStarterComponent specific parameters, which can be accessed via the handed over AppStarterComponentCfg object. The components are always instantiated and called in the order, they appear in the configuration file.

If the initialization phase succeeded, the startup phase is initiated. In this phase, all the startup() method of all components is called till all components indicate the RUNNING state. The startup() method of each component is called once per startup step. Within one step, the components are called in the order they appear in the configuration file. If a component indicates an error, the startup process is stopped and all previously executed startup steps are undone by calling the corresponding shutdown() method. If a component indicates the RUNNING state, it is not called in the next startup step anymore.

The basic startup sequence of AppStarter illustrates following sequence diagram:

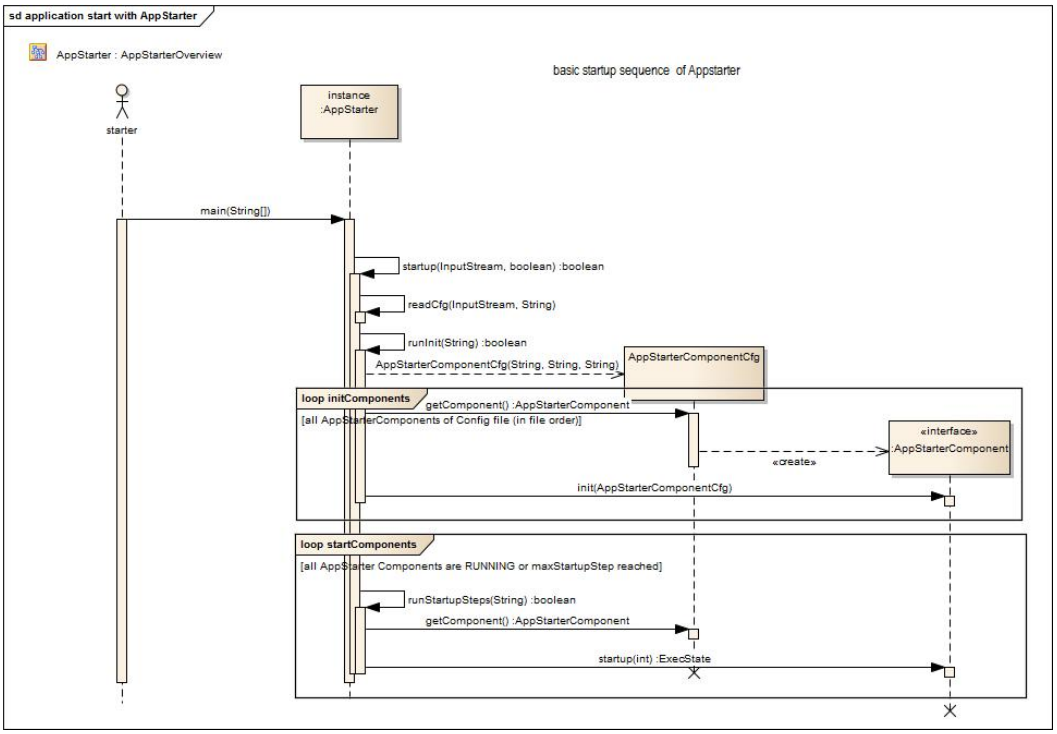


Figure 8: AppStarter start sequence

4.3. AppStarter configuration file

The configuration file defines the AppStarterComponents which need to be called during startup and shutdown and some AppStarter internal configuration parameters. It is based on the following XML schema:

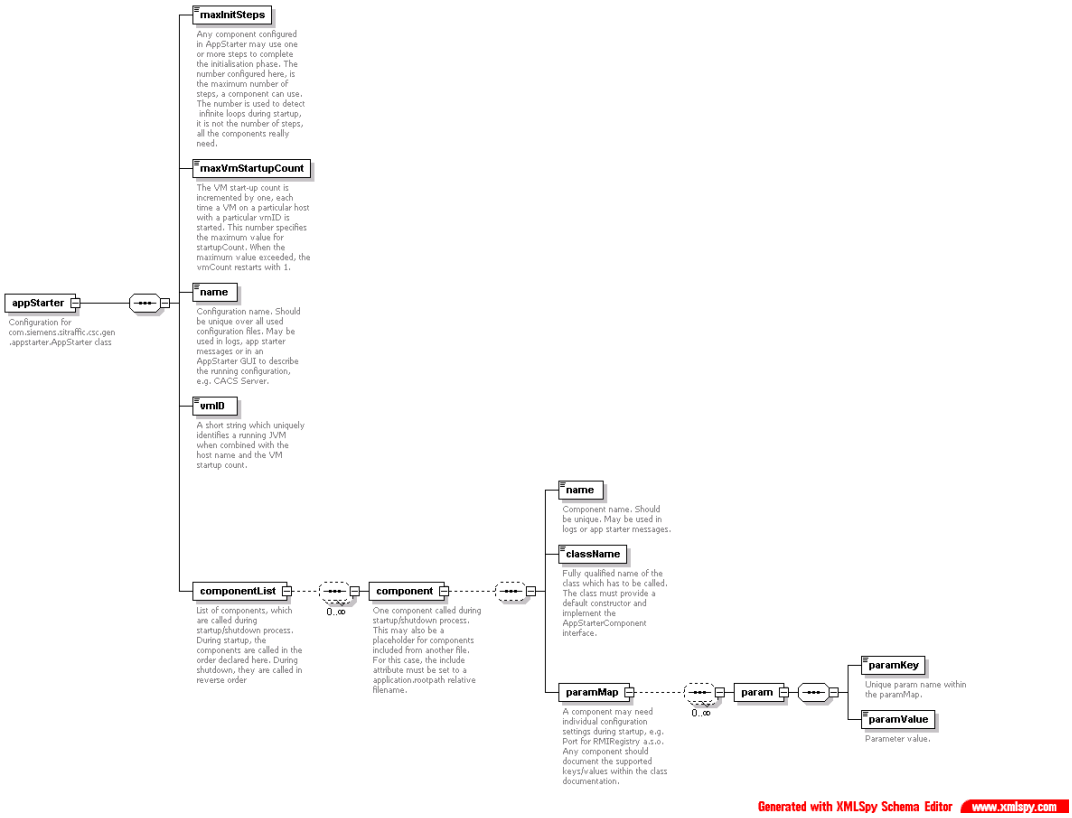


Figure 9: XML Schema for configuration file

AppStarter configuration files and configuration include files may be given as absolute filenames or relative filenames. The AppStarter uses the following search rules for relative files:

Relative to directory stored in application.rootpath system property

Relative to the {classpath}/cfg

The vmID (in combination with a startup count and the hostname) is used to uniquely identify the running VM. It represents one VM configuration.

4.4. Component specific configuration parameters

Each AppStarter component can define its own parameter names. The component can read the parameter value using AppStarterComponentCfg.getParam(paramName, defaultParam-Value). The parameter value may be configured in the configuration file using an fixed string or with placeholders, which are expanded to values of other parameters or to values of system properties. The expansion process can only expand parameter names of the current component or the system properties available, when expansion takes place.

Simple parameter example used for the initialization of the Log4j properties:

```
<paramMap>
  <param>
    <paramKey>START_LEVEL</paramKey>
    <paramValue>1</paramValue>
  </param>
  <param>
    <paramKey>INITLOG4JCFG</paramKey>
    <paramValue>log4j/ControlCenterLoggerProvider.properties</paramValue>
  </param>
</paramMap>
```

The AppStarter library also includes a global property component, which simply exports all parameters configured in this component as system properties. Already set system properties (may be handed over during VM start with the command line option -D) will NOT be overridden and therefore be ignored! Only the initialization phase of the startup steps is used. Parameters are ready for use, after component has finished init() method.

Simple configuration sample for the global property component:

```
<component modes="normal,stop">
  <name>Global Properties</name>
  <className>
    com.siemens.sitraffic.csc.gen.appstarter.globalprop.GlobalPropertyComponent
  </className>
  <paramMap>
    <param>
      <paramKey>localAddr</paramKey>
      <paramValue>192.168.237.1</paramValue>
    </param>
    <param>
      <paramKey>omcAddr</paramKey>
      <paramValue>192.168.237.231</paramValue>
    </param>
  </paramMap>
</component>
```

4.5. How to implement an AppStarter component

This is a very simple example how to create a new AppStarter component containing some business logic. This component shall be deployed as a service, making it accessible throughout the system. It is just a minimalistic example.

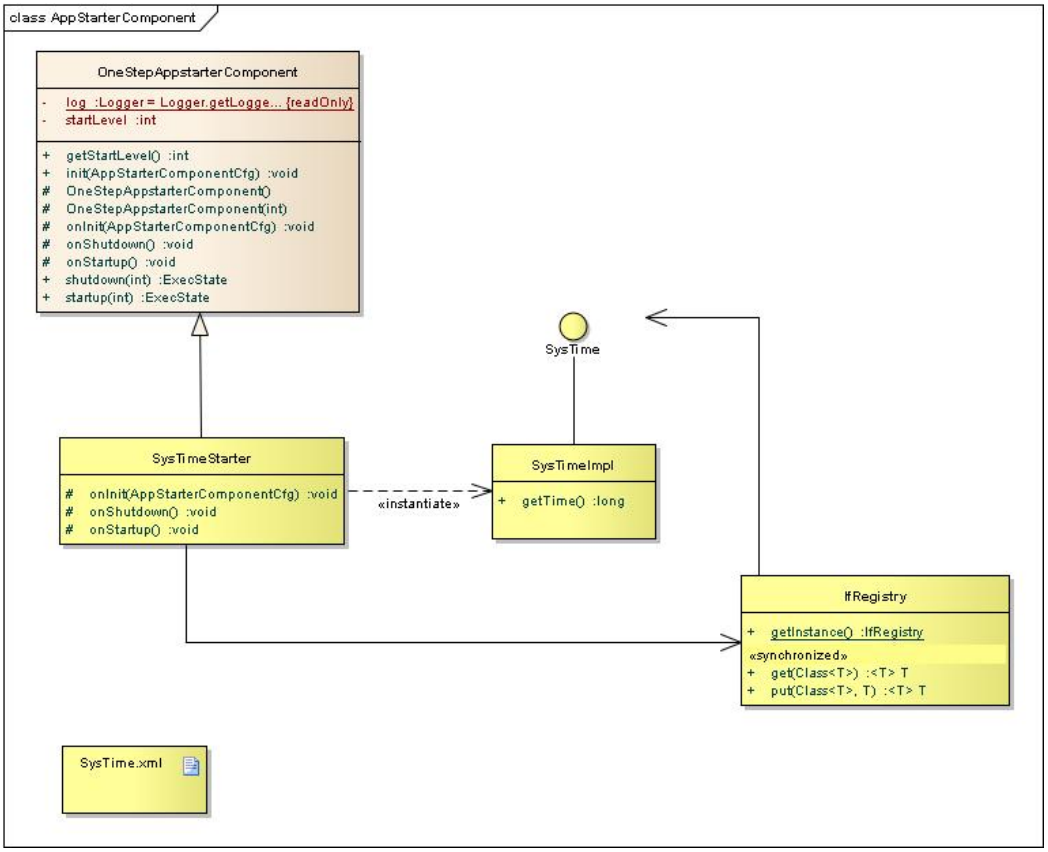


Figure 10: Class Diagram of an example AppStarter Component

The AppStarter component here consists of a simple class called SysTimeImpl implementing a simple interface called SysTime which shall be accessible for all other components in the system. Specifically this example will offer the system time through the following interface and implementation:

```
package com.sitraffic.fd.ic.common;
public interface SysTime
{
  /**
  * returns the current system time. This time is represented
  * in milliseconds since January 1st, 1970.
  * @return long milliseconds since January 1st, 1970.
  */
  long getTime();
}
```

```
package com.sitraffic.systime.impl;
import java.util.Date;
import com.sitraffic.fd.ic.common.SysTime;

public class SysTimeImpl implements SysTime
{
  @Override
  public long getTime()
  {
    Date dt = new Date();
    return dt.getTime();
  }
}
```

Each AppStarter component should have a starter class which extends the OneStepAppstarterComponent class and can therefore override e.g. the methods onInit, onStartup, and onShutdown.

During the onInit the component should be instantiated and its interface should be made accessible to other components by putting it to the central registry IfRegistry. The IfRegistry class contains a hash table where you can register the public interface of components which can be then used by other components depending on that interface. Since the components are implemented as AppStarter components, they have to register their public interfaces at the initialization step of the AppStarter process startup.

```
package com.sitraffic.systime.starter;

import org.apache.log4j.Logger;
import com.siemens.sitraffic.csc.gen.appstarter.AppStarterComponentCfg;
import com.siemens.sitraffic.csc.gen.appstarter.AppStarterComponentException;
import com.siemens.sitraffic.csc.gen.appstarter.OneStepAppstarterComponent;
import com.sitraffic.component1.impl.AppStarterComponent1;
import com.sitraffic.common.IfRegistry;
import com.sitraffic.systime.impl.SysTimeImpl;

public class SysTimeStarter extends OneStepAppstarterComponent
{
```

```
  private static Logger LOG = Logger.getLogger(Component1Starter.class.getName());

  private SysTimeImpl sysTime;

  @Override
  protected void onInit(AppStarterComponentCfg cfg)
    throws AppStarterComponentException
  {
    // create a new component instance and put the service interface to the
registry
    this.sysTime = new SysTimeImpl();
    IfRegistry.getInstance().put(SysTimeImpl.class, this.sysTime);
  }

  @Override
  protected void onStartup()throws AppStarterComponentException
  {
    // do some further startup steps
  }

  @Override
  protected void onShutdown() throws AppStarterComponentException
  {
    // do some shut down steps
  }
}
```

Suppose another component named ComponentX depends on the Interface SysTime. The right time to get the implementing class of this interface is on the trigger onStartup of its starter class. Suppose the ComponentX has starter class ComponentXStarter, then getting the SysTime implementation looks like following:

```
public class ComponentXStarter extends OneStepAppstarterComponent
{
  ...

  private ComponentX componentX;

  @Override
  protected void onInit(AppStarterComponentCfg cfg)
    throws AppStarterComponentException
  {
    this.componentX = new ComponentX();
  }

  @Override
  protected void onStartup() throws AppStarterComponentException
  {
    SysTime sysTime = IfRegistry.getInstance().get(SysTime.class);
    this.componentX.setSysTime(sysTime);
  }

  ...
}
```

The following describes an example of the SysTime AppStarter configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<appStarter xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:noNamespaceSchemaLocation="AppStarterInclude.xsd">
  <componentList>
    <component>
      <name>SysTime</name>
      <className>com.sitraffic.systime.starter.SysTimeStarter</className>
      <paramMap>
        <param>
          <paramKey>START_LEVEL</paramKey>
          <paramValue>1</paramValue>
        </param>
      </paramMap>
    </component>
  </componentList>
</appStarter>
```

5. Use Cases

This chapter describes some main use cases to support the understanding of the example implementations of the traffic actuation and control center process.

5.1. Setup and Preconditions

Basically all the use cases follow the same pattern: first an action is performed using the JMX GUI of the observed traffic actuation or control center process and then the corresponding response is observed on the Controller GUI of the virtualized Sitraffic sX controller.

The following preconditions are relevant for all use cases:

- The virtualized Sitraffic sX controller is started and running properly.
- The virtualized Sitraffic sX controller is correctly configured.
- The controller service GUI can be opened and shows the log-on page. This is indication of successful connection to the virtualized controller.
- The JMX GUI can be opened and shows the entry page of the process you have connected to. This is indication of successful connection.
- The example process (control center or traffic actuation) can successfully be launched from the Eclipse environment.
- The communication between the example java process with the virtualized Sitraffic sX controller was established successfully. Here especially the firewall settings are crucial.

For more detailed information and instructions to problem solving please refer to the installation Guidelines [1].

5.1.1. JMX GUI: Monitoring and Operating the Example Applications

For the administration, monitoring and operation of the example traffic actuation and traffic control center processes you can use the implemented JMX (Java Management Extensions) GUI. With the usage of the Managed Beans (MBean) technology, it is possible to access the example processes

via http-based protocol. Please note that the example process has to be running in order to establish communication (see 3.3.4).

It is possible to access the example processes with a standard web browser or from the JConsole (Java Console) by showing the MBean. In order to access an example process using a web browser you have to insert the right hostname and port into the URL (= uniform resource locator, the specific character string that constitutes the reference to the MBean resource). The syntax is like follows:

<hostname>:<port>

If the example process is started directly from Eclipse and therefore running locally on your host machine, you have to insert the following URL:

- Control Center Example: localhost:8022
- Traffic Actuation Example: localhost:8012

Otherwise, if the example process is started directly on the target machine, respectively the virtualized Sitraffic sX controller, then you have to use the IP address of the controller. After inserting the URL, your web browser should show the registered MBeans, which we call here the JMX GUI.

Both examples provide MBeans showing specific information.

Control Center Example

As soon as you open the ControlCenter MBean by selecting "C10.ControlCenterClient", the following information from the controller configuration is displayed:

- The IP address of the virtualized Sitraffic sX controller and if it is reachable
- Field device name
- Number of control center and field device
- List of configured sub-intersections, identified by number
- List of configured signal programs, identified by number.

Traffic Actuation Example

Opening the MBean of the traffic actuation process by selecting "C10.SharedValueStorage" reveals to following information:

- Actual and Requested status
- Signal indications and detector data
- Target Stage

5.1.2. JMX GUI: Actuation of Detectors

Throughout the workflow of the traffic actuation example you need to trigger a detector. As no real loop detectors are available only a simulated actuation is also possible. You need to start the JMX GUI of CBC-Sim (CBC-Simulation, see [4]) which provides access to simulated detector operations. Nevertheless, the example java processes will not notice the difference. For simulated as well as for real actuation, detector data is provided via the C-Control interface. A more detailed discussion of the differences between the real- and the virtualized Sitraffic sX controller, please refer to [4].

Start the web browser and address CBC-Sim by its URL using hostname and port. For connecting to the virtualized Sitraffic sX controller please use:

- CBC Simulation: http://192.168.237.231:8032

If the connection has been established successfully, the registered MBeans are available. For simulated actuation of detectors select:

- "C10.CbcSim name=DetStateViewMXBean"

If you cannot find the MXBean please check whether a valid detector configuration was loaded successfully.

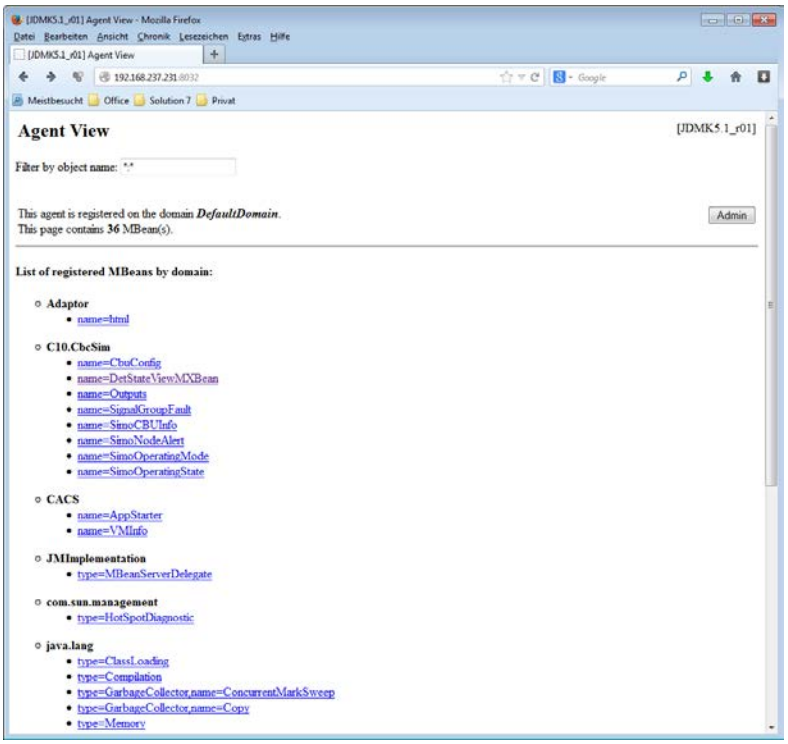


Figure 11: JMX Web GUI showing the registered MBeans

If you connected successfully to CBC-Sim, the available MBean attributes are:

- ActualDetectorFaultStates: Describes the simulated error state of all available detectors. Please note that only detectors configured correctly can be seen.
- PossibleErrorCodes: Here you see all available enum-types, corresponding to simulated error states.

The following operations can be performed:

- setDetectorFaultState: That function is used for setting the simulated error state of an detector. The detector number must be one listed by “ActualDetectorFaultStates”.
- setDetectorOccupation: This function is used for simulating detector occupation. The occupation starts immediately after submitting the command for the duration set at “occupation time[ms]”. The detector number must be one listed by “ActualDetectorFaultStates”.

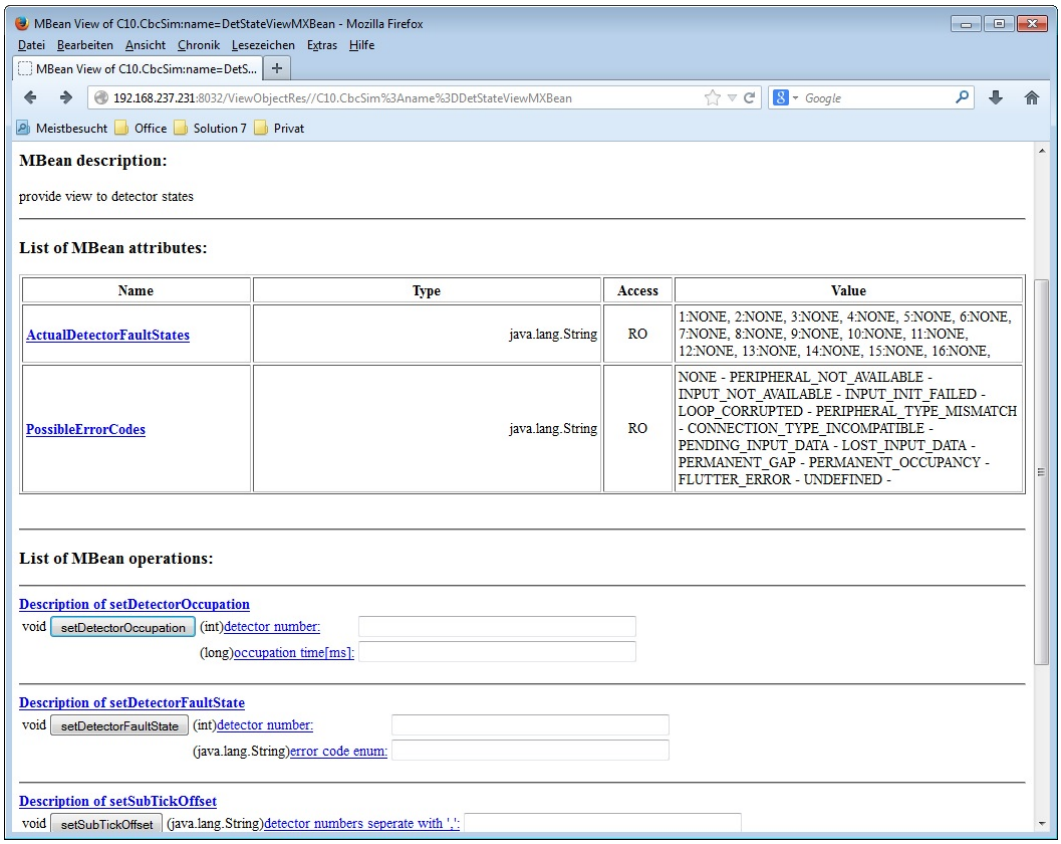


Figure 12: JMX Web GUI showing the MBeans attributes

A detector occupation can be simulated by selecting "List of MBEAN operations -> setDetectorOccupation". For a addressing a detector you have to enter its number and the duration of occupancy. The very actuation itself takes place at the moment you push the ®setDetectorOccupation-Button (see Fig. 11).

5.1.3. Controller GUI: Loading Configuration

One important precondition for running the example applications is that the virtualized Sitraffic sX controller is configured correctly. A suitable configuration file comes with the DWS installation and will be addressed later on.

For editing and generating such files of type *.c10 yourself, the configuration tool Sitraffic SmartCore [3] is part of this delivery. A detailed description how to install and use Sitraffic SmartCore please refer to [3].

In any case you have to import the configuration file into the virtualized controller using the Controller Service GUI. For a more information how to use it, please refer to [2]. Here we will focus just on the basic features necessary throughout the described workflow.

For the operating the Sitraffic sX controller (real or virtualized) you can use its web based Service GUI. A HTML 5 capable web browser can access that GUI via http. In the context of this development environment (DWS), please use the following IP address:

- Sitraffic sX Service GUI: http://192.168.237.231

After successfully connecting to the virtualized Sitraffic sX, completing the log-in is next. For loading a new configuration it is necessary to chose either the “Service” or “Developer” role.

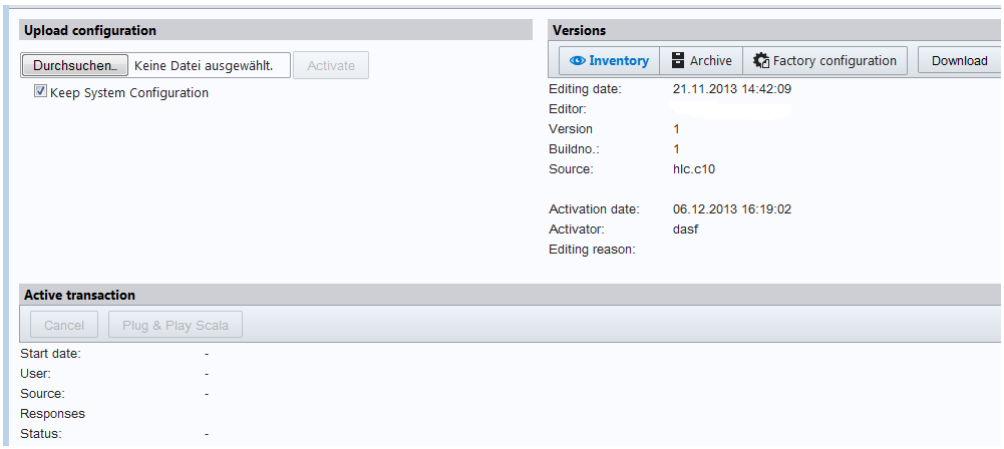


Figure 13: Sitraffic sX Controller Service GUI, main window for changing configuration.

Following the successful log-in the main Service GUI is displayed. In the first line of buttons, please select “Configuration” and push the picture-button “Configurations”. Now the window for importing configuration files is displayed (see Figure 13).

Please choose “Browse” at “Upload configurations”. If you have chosen the default installation path, you find the example configuration file at: “C:\dws\configurations\config001\hlc.c10”. Upload and activate the configuration.

In some cases a manual restart of the controller is necessary. After successfully staring up and logging in to the virtualized Sitraffic sX controller its LINUX shell is available (see Figure 14). A full restart can be

triggered using the "reboot" command. For a less comprehensive reboot on path "/etc/init.d/" the script "./application restart" is available.

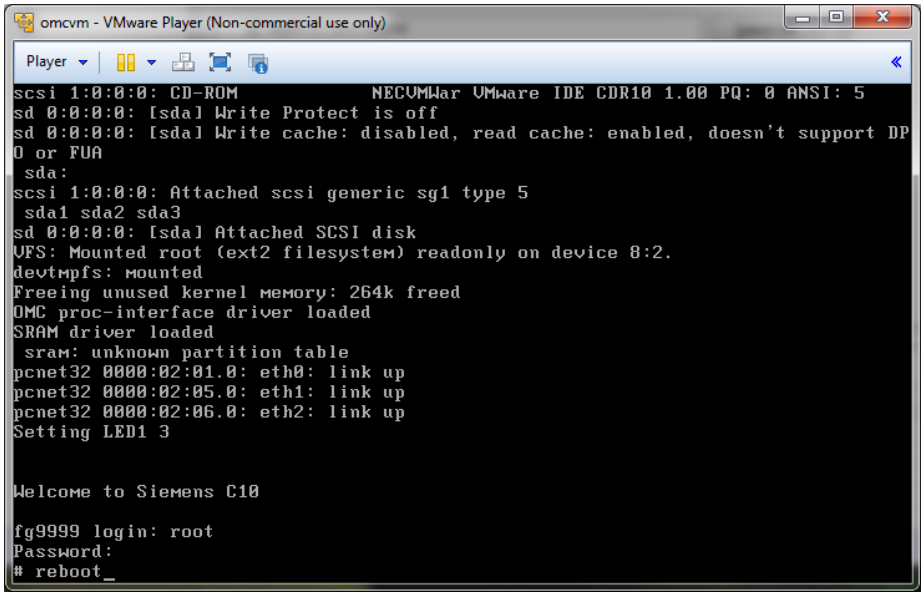


Figure 14: Sitraffic sX VM Linux Shell.

5.1.4. Controller GUI: Monitoring and Operating

Besides loading configuration the Service GUI is used for various monitoring and operating tasks. For a detailed description, please refer to [2]. For the purpose of our examples it is just necessary to change the operating state. Navigate to “Operation -> Operate TSS” for changing the operational state (see Figure 15). The most relevant states in the present context are:

- Selection of Signalplan
- Activate/Deactivate Main Intersection
- Activate/Deactivate Traffic Actuation
- Activate/Deactivate Individual Traffic and/or Public Transport

The impact of the different operating states on the controllers behavior will be illustrated by the use case examples. For a more detailed description of the underlying mechanism, say the control logic of the whole Sitraffic sX platform please refer to [4].

Handling

When setting the operating state, it is worth paying some attention to practical handling. This screen shot shows the main window for setting the operating state of the controller (see Figure 15). Please note that besides the state itself (e.g. TA: ON/OFF) the object whose state should be modified (e.g. TA) has to be selected by check box. Pushing the “Operate” Button finally activates the change in operating state.

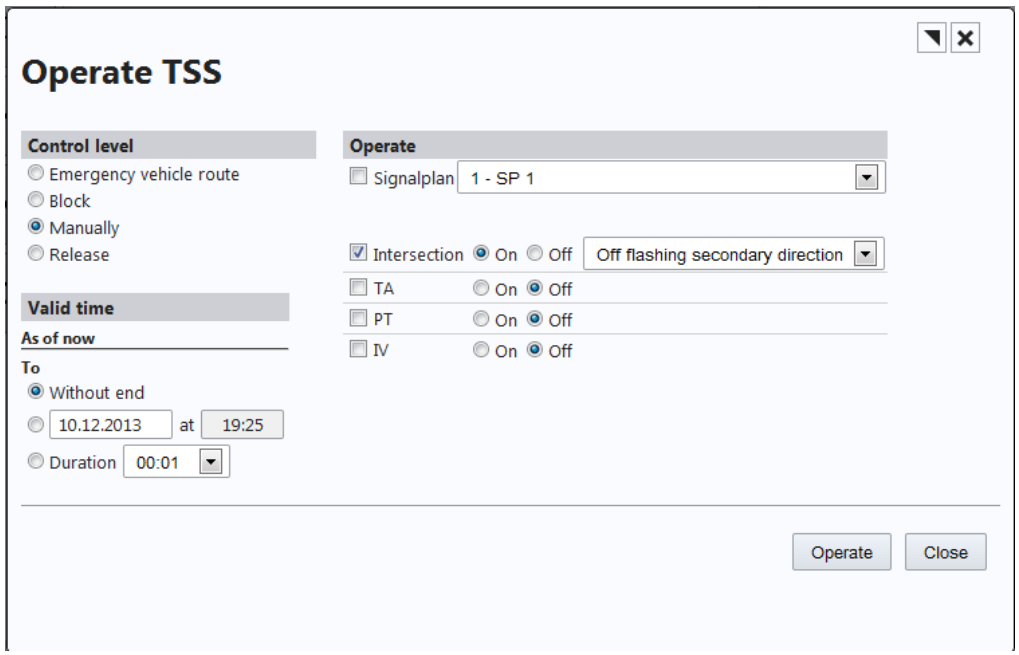


Figure 15: Controller Service GUI, window for setting the operation state.

5.2. Use Cases for the Control Center Process

5.2.1. Use Case: Show the Actual Status

Description

The ActualStatus delivers the current operational state and a collection of detected malfunctions of the virtualized Sitraffic sX controller. The traffic control center processes can explicitly request the ActualStatus or receive it via events if it registers himself as a listener.

Preconditions

The preconditions of chapter 5.1 also apply here.

Steps to be executed

Select the MBean “name=ControlCenter” and request the actual status by clicking the button „showActualStatus“ on the JMX GUI. If you cannot find the Control Center Bean, please check the configuration file “StartControlCenterExtern.xml”. If you have chosen the standard installation path, the file is located at “C:\dws\ControlCenterIfProcessExample\jappl\cfg” and refer to the DWS Installation Guide [1] for configuring IP addresses.

Expected result

The ActualStatus is requested and received successfully from the virtualized Sitraffic sX controller and is shown as simple text on the JMX GUI of the control center process.

Other states and information which can be requested are target state ("showTargetStage"), requested state ("showRequestedState"), detector data ("showDetData") and signal indications ("showSignalIndications").

5.2.2. Use Case: Control Center Switching Request

An operator of a control center can request several types of switching operations:

- Switch on/off the whole intersection
- Switch on/off partial intersections

- Select a signal program
- Switch on/off special interventions

Each switching operation serves with a transaction key, time duration and a specified priority of the process.

A switching operation can be generated through different sources like the traffic actuation process, the control center process and also different sources like detectors, hand panel or scheduler for the annual automatic. A switching operation of a specific source has therefore a specific priority and the one with the highest priority will be taken into account.

Competing switching operations, e.g. two switching operation which should be activated during the same period are evaluated in correspondence to their priority. A control center operator e.g. has a lower priority then a manual operator using the hand panel. Therefore, the manual switching operation from the hand panel blocks the one from the operator of the control center.

An already sent switching operation can be cancelled by using 0 as the time duration.

5.2.3. Use Case: Switch on whole intersection from initial state

Description

The aim of this use case is to switch on the whole intersection and to observe how a signal program is activated if the virtualized controller is started the first time, where no signal program is selected.

There are several off states for a whole or a part of an intersection:

- Off Flash (value OFF_FLASH_SECONDARY_DIRECTION)
- Off Flash All (value OFF_FLASH_AGG)
- Off Dark (value OFF_DARK)

Preconditions

The following preconditions have to be met for this use case and can be checked via the ActualStatus (see chapter 5.2.1):

- The main intersection is in one of the Off-States, e.g. Off Flash All (OFF_FLASH_ALL).
- The sub-intersection is in one of the Off-States, e.g. Off Flash All (OFF_FLASH_ALL).
- No signal program number is selected yet for activating any program. This corresponds to signal program number 0.
- Additionally you should observe “off” on the display of the virtual hand panel of the Controller GUI.

Another important precondition is that no other higher prior switching operation is active in the same time. This you can check in the control levels view of the Controller GUI.

Steps to be executed

The following actions shall be done in the JMX GUI of the traffic control center process:

1. Select signal program number
   You have to enter a valid signal program number and a time duration in seconds into the corresponding fields of the “switchSignalPlan” action and then perform it.
2. Switch on the sub intersection
   You have to switch on a valid (normally only one is available) sub intersection by entering the correct sub intersection number and the ON (integer value 1) to the corresponding fields of the „switchSubIntersection“ action and then perform it.
3. Switch on main intersection
   In order to switch on the main intersection you have to enter the ON state (integer value 1) to the corresponding field of the „switchIntersection“ action and then perform it.

Expected result

If all actions are performed successfully, the JMX GUI should display the return code RC_OK. The switch operation should be visible now in the control levels view of the Controller GUI and the text “Central Signal Program” should be displayed in the first line and a running “TX” in the second line on the virtual hand panel.

The individual states of the selected signal programs, the main intersection and the sub intersection can be observed by requesting the ActualStatus from the JMX GUI (see chapter 5.2.1).

5.2.4. Use Case: Select a specific signal program

Description

Selecting a new signal program from a Control Center preserves the on/off states of the intersection and partial intersection.

Precondition

Even if a signal program change request preserves the other states, this example use case shall highlight the behavior of the controller when the whole intersection and the first partial intersection is in the on state. This means a dedicated signal program is already active and running.

Steps to be executed

The following action shall be done in the JMX GUI of the traffic control center process:

1. First check from the ActualStatus in order to find out the actual activated signal program number (see use case in chapter 5.2.1)

2. Select new signal program number
   After going back to the main page of the control center JMX GUI you have to enter a new valid signal program number and a time duration in seconds into the corresponding fields of the “switchSignalPlan” action and then perform it.

Expected result

If all actions are performed successfully, the JMX GUI should display the return code RC_OK.

The new signal program will not be activated and running immediately after requesting it because the actual running signal program can’t be stopped right away when the request arrives. This could lead e.g. that the signaling color would jump suddenly from red to green or vice versa. The controller avoids this by waiting until the actual running signaling program is in a special state, where all signaling states of both signaling programs

are similar (called “best switching point”). You can observe this behavior from the Visu STP (visualization signaling time plan) view of the Controller GUI.

After the new signal program is active and running, you can check this by requesting the ActualStatus from the JMX GUI (see chapter 5.2.1).

5.2.5. Use Case: Switch off whole intersection

Description

It is also possible to switch off a controlled intersection.

Precondition

This use case makes sense when the intersection and the partial intersection is in the on state and if a signal program is running, so that you can see the behavior of the controller.

Steps to be executed

The following action shall be done in the JMX GUI of the traffic control center process:

1. First check the preconditions from the ActualStatus

2. Switch off main intersection
   In order to switch off the main intersection you have to enter one of the off states (e.g. OFF_DEFAULT = integer value 2) to the corresponding field of the „switchIntersection“ action and then perform it.

Expected result

If the action is performed successfully, the JMX GUI should display the return code RC_OK.

The main intersection will not switch off immediately. It can switch off if the running signaling program is in a harmless state, which can take a while, the so-called “deactivation point of time”. You can observe this behavior from the Visu STP view of the Controller GUI.

5.2.6. Use Cases: Device Variables respectively AP-Values

Description

Device variables (also called AP-Values or in German language "Anwenderprogrammwerte") are internal dynamic variables of a running Sitraffic sX controller. A control center should be able to subscribe to device variables in order to receive and visualize their values and archive them for viewing them later. Currently the Sitraffic sX firmware and the traffic actuation process define device variables like "DigitalOut" for configured digital outputs of the Sitraffic sX controller or the "VAState" for the current state of the traffic actuation process.

The type of device variable values is a 4-byte number, in Java represented as the int primitive type. Device variables are addressed by their OITD number, which is defined by its member type and object type. The member type stands for the producer specific device variables and is specified in the OCIT standard. Siemens e.g. has the producer ID number 60. The object type, which is also just an ID number, defines each device variable. Device variables can be defined as a single variable or as a group (list). If they are defined as a group then each item in the group is addressed by an index, like the position in a list. The minimum and maximum indexes define the borders of the list. If the device variable is defined as a single variable then the index is either 0 or 1 (depending on the definition) and therefore the minimum and maximum indexes are equal.

Preconditions

All device variable use cases require that the system is up and running (see preconditions of chapter 5.1)

5.2.6.1. Use Case: Ask for device variable definitions

Description

The device variable definitions include the name, description and OITD number of available device variables in the Sitraffic sX controller. The device variables are sorted by their provider like the traffic actuation process or the firmware process itself.

Steps to be executed

Call the method “showDeviceValuesMetaData” in the JMX GUI of the traffic control center process in order to ask for the device variable definitions.

Expected result

If the call performed successfully, the JMX GUI should display a list of device variable definitions available in the Sitraffic sX controller.

5.2.6.2. Use Case: Ask for the availability and existence of a single device variables

Description

The traffic control center process can ask for the availability respectively existence of a single device variable before a subscribe or unsubscribe is called.

Steps to be executed

Select the method "checkIfOitdExists" in the JMX GUI of the traffic control center process and type in the member type, object type and index of a device variable of interest to ask about its existence in the virtualized Sitraffic sX controller.

Expected result

If the call performed successfully, the JMX GUI should display a text, which says if the device variable exists or not.

5.2.6.3. Use Case: Subscribe and unsubscribe to device variables

Description

Device variable values can be received only if they are subscribed. In order to avoid unnecessary loading of the network just subscribe the necessary device variables and unsubscribe not needed ones.

Steps to be executed

To subscribe a device variable select the method "subscribe4DeviceVariable" in the JMX GUI and type in the member type, object type and index of the corresponding device variable.

To unsubscribe a device variable select the method "unsubscribe4DeviceVariable" with the same parameters as for "subscribe4DeviceVariable".

Expected result

If the call is performed successfully, the JMX GUI should display the return code RC_OK. The method does not return with an error if the specified device variable does not exist. Therefore, you can ensure the existence by the use case above (chapter 5.2.6.2).

5.2.6.4. Use Case: Receive device variable values

Description

Values of subscribed device variables can be received.

Steps to be executed

By clicking the button "showYoungestDeviceValues" on the JMX GUI, the youngest device variable values are requested.

Expected result

The youngest values of subscribed device variables are requested and received successfully from the virtualized Sitraffic sX controller and are shown as simple text on the JMX GUI of the control center process.

5.3. Use Cases for the Traffic Actuation Process

5.3.1. Use Case: Test of a fully adaptive stage-oriented control logic

Description

The aim of this use case is to illustrate the testing of a fully adaptive stage oriented traffic actuated control logic within the provided development framework.

A toy example is part of the delivery. This basic example of a fully traffic actuated control logic uses two stages. Without any detector actuation the so-called "Idle Stage" is signalized. Upon detector actuation a stage transition to the so-called "Occupied Stage" is executed. That stage is left again after expiration of its predefined duration.

The schematic control logic reads:

- The so-called Idle Stage (e.g. Stage 2) is running permanently.
- If the pre-defined Triggering-Detector is actuated (e.g. Detector 1), and the minimum duration of the Idle-Stage has expired the stage transition into the so-called Occupied Stage (e.g. Stage 3) is started.
- After expiration of the pre-defined duration of the Occupied stage, the stage transition back into the Idle Stage is performed.
- For a schematic illustration please see Figure 16.

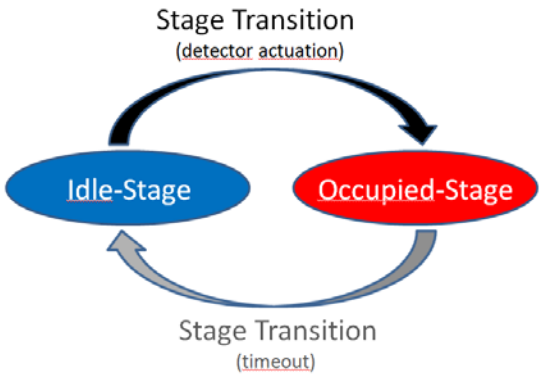


Figure 16: Illustration of control logic example.

The supported way for providing parameters to the traffic actuation process is a configuration file “StartTACExtern.xml ”. All the necessary parameters like stages, detectors or durations used for the example are

defined there. You find that file at its default location "C:\dws\TrafficActuationControlExample\jappl\cfg".

Please note that the parameters provided by the XML configuration file are technically independent from the basic controller configuration files (*.c10). The traffic actuation parameters are therefore not accessible via the C-Control interface and there are no preexisting check mechanisms. If you provide parameters to the traffic actuation process in that way, you need to pay special attention to consistency. In the context of this example this means that you can only switch between stages that are defined in the basic controller configuration and selected for use by the control logic in the external parameter file.

The given example is quite generic. Currently it makes no use already existing technology such as Sitraffic Language or any specific control method for local traffic actuation like PDM or SL. Its main purpose is merely to illustrate the technical principles of the Sitraffic sX controller platform. But of course the platform is intended to stimulate further development.

Preconditions

For running the traffic example process, you have to gather some configuration data. Overall the following preconditions have to be met:

- Configuration data containing at least one stage-oriented signal plan have to be loaded into the controller (e.g. on the default installation path C:\dws\configurations\config001\hlc.c10, see Figure 13).

- The stage-oriented signal plan used for testing local traffic actuation must at least provide two stages and the corresponding stage-transition (e.g. SP5 of config001). The Controller Service GUI [2] provides a visualization of the configured signal plans. In case of stage-oriented plans, the configured stages and stage transitions are indicated by background color and annotation (see Figure 17).

- The configuration data needs to provide at least one detector (e.g. D1_1 of config001).

Please note that at the current stage of development the basic controller configuration data (*.c10) just needs to provide the latter objects. There are currently no special attributes for marking a detector as Triggering Detector, or defining stages to be Idle- or Occupied Stage. That assignment is done in the additional configuration file (*.xml).

- The java project “TrafficActuationControlExample” was imported into the Eclipse IDE (see chapter Importing the Projects into Eclipse 3.3.1).
- Make sure that communication between the VMware Player and the outside world has been established.

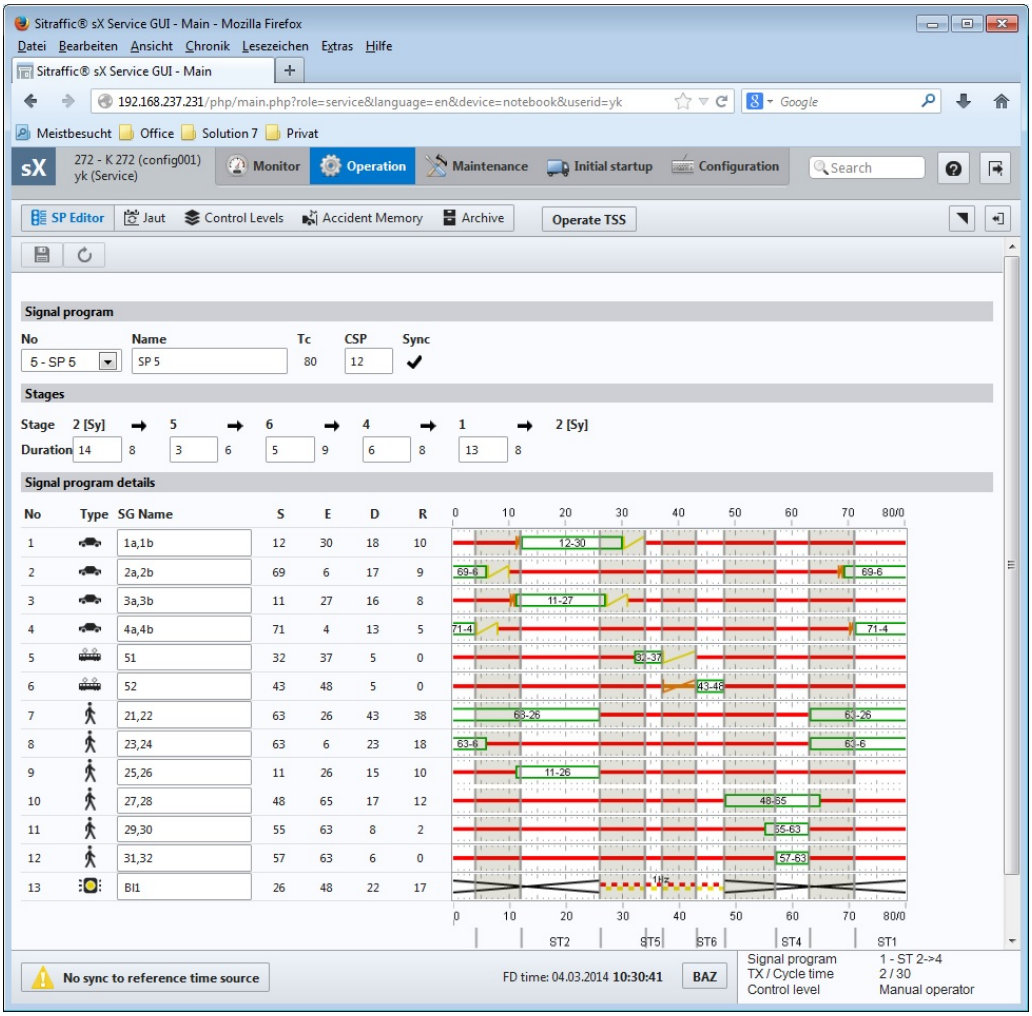


Figure 17: Controller Web GUI showing the signal program editor.

The whole use case of a full functional roundtrip is divided into three smaller and hence more manageable sub cases. Especially the Starting- and Stopping of the traffic actuation process shows general features of the Sitraffic sX controller regarding the interplay of operational state and switching requests. Running and triggering the traffic actuation gives an idea how such a control logic could be implemented in Java.

5.3.1.1. Starting the Traffic Actuation

Steps to be executed

For testing the traffic actuation process the following steps need to be executed:

1. Service GUI -> Operation -> Operate TSS: Activate the stage-oriented signal plan, the intersection and the partial-intersections and push the "Operate" button (see Figure 15).

2. Service GUI -> Monitor -> Visu STP: Check whether the stage-oriented signal-plan is signalized. This is indicated by the change of stages throughout the signalization cycle and the corresponding change of signal indication of signalgroups (see Figure 18).

3. Eclipse IDE: Launch the TrafficActuationControlExample as java application using the corresponding configuration (see 3.3.4).

4. Eclipse IDE: Check on the console and log files (on default path C:\dws \TrafficActuationControlExample\jappl\jAppl\log\tac) whether there are any errors or warnings. If the java process is up and running this is indicated by the red button “Terminate” on the consoles frame.

5. Service GUI -> Operation -> Operate TSS: Activate the Traffic Actuation (TA) and at least one actuation type, Individual (IV) or Public Transport (PT) (see Fig. 14).

Expected result

Service GUI -> Monitor -> Visu STP: Check whether one Idle-Stage (e.g. Stage 2) is signalized permanently.

5.3.1.2. Testing the Traffic Actuation Logic

Steps to be executed

1. JMX GUI of CBC Sim -> DetStateViewMXBean: Use the command “setDetectorOccupation” (e.g. DetNo. 1) with an non-zero occupation time (e.g. 5000ms) for simulating detector occupation.

Expected result

Service GUI -> Monitor -> Visu STP: Check whether the occupation of the detector has been perceived by the controller (see Figure 18).

Service GUI -> Monitor -> Visu STP: Check whether the detector occupation has triggered the immediate stage transition into the “Occupied-Stage”.

Service GUI -> Monitor -> Visu STP: Check whether there is a stage transition back into the “Idle-Stage” after expiration of the configured occupiedStageActiveTime (see 5.3.2.2).

5.3.1.3. Stopping the Traffic Actuation

The traffic actuated control can be stopped in two different ways. In the regular way the operational state is used. More natural to the Eclipse environment is the other way, where the traffic actuation process is terminated directly. Nevertheless in a real deployment scenario this corresponds to an irregular deactivation. This might happen when for example the traffic actuation process crashes due to internal malfunction.

Generally the traffic actuation switching request is refreshed periodically. When signalizing the “Idle Stage” this is according to “IdleStageRefreshTime” (see TacExample.xml). On detector actuation the stage transition as well as the “Occupied Stage” are send as one single switching request. Both variants of terminating the impact of the traffic actuation process differ in the way the switching requests are handled.

5.3.1.3.1. Irregular Deactivation

This case corresponds to an immediate termination of the traffic actuation java process.

Steps to be executed

1. Eclipse IDE: Stop the Traffic Actuation Example process by pushing the red “Terminate” Button on the consoles frame.

Expected result

Service GUI -> Monitor -> Visu STP: Check whether after the expiration of “IdleStageRefreshTime” the selected fixed time signal plan is ran.

As the java process is simply terminated there is no refreshing of the traffic actuations switching request which just time-out. The very time span equals the period for which signal indications are defined by the last switching request. In case of our example this means that there are

different time-out durations depending on the stage that is currently running. For more details please refer the discussion of the java implementation (see 5.3.2.3).

5.3.1.3.2. Regular Deactivation

The shut-down is triggered by an external command. Unlike in the latter case, the traffic actuation process plays an active role in the shut-down process.

Steps to be executed

1. Service GUI -> Operation -> Operate TSS: Deactivate the Traffic Actuation (TA) and at least one actuation type, Individual (IV) or Public Transport (PT).

Expected result

Service GUI -> Monitor -> Visu STP: Check whether the stage-transition into the currently selected fixed time signal plan is started.

This way of deactivation is performed in a defined way. Although not part of the present example, the traffic actuation could provide signal indications for making a smooth transition to fixed-time control.

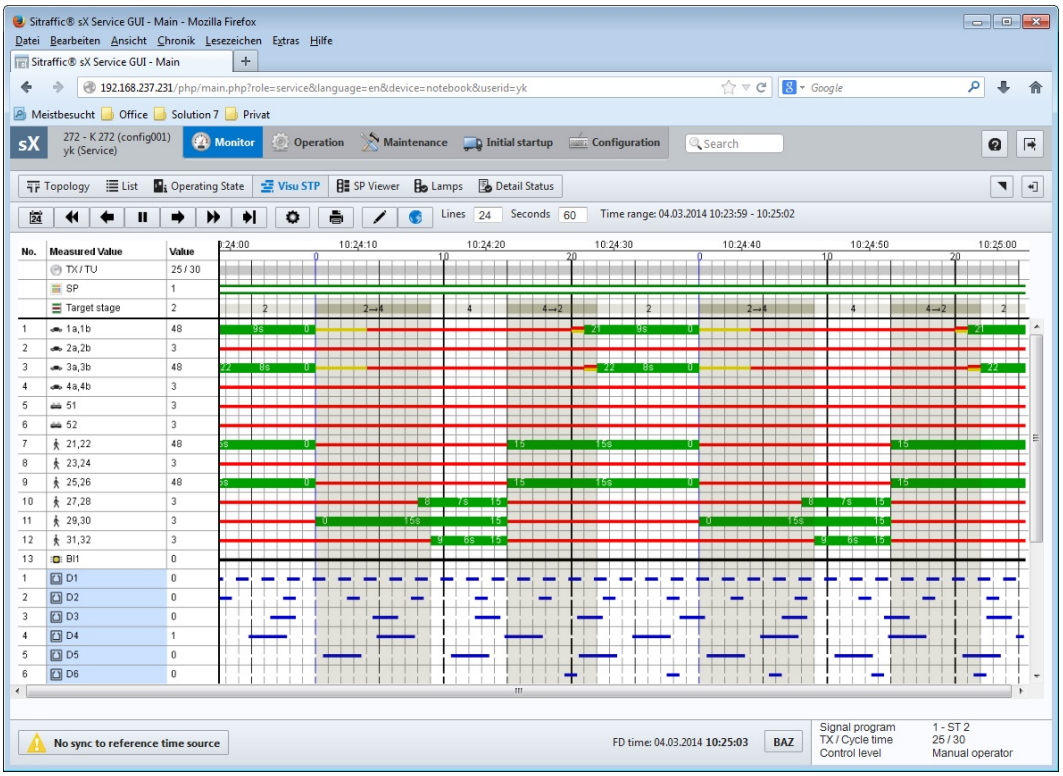


Figure 18: Controller Service GUI showing the online signal program visualization.

5.3.2. Use Case: Definition of a fully adaptive stage-oriented control logic

Description

The previous example provided the basic functionality of a fully adaptive actuated control logic. Part of that example is the java implementation of the control logic itself. Customizing that example, say editing the java implementation, is the focus of this use case.

Readers familiar with existing traffic actuation logics might wonder about specific control methods (e.g. PDM or SL) or customized user-defined logics (e.g. Sitraffic Language). Currently none of these are part of the example project. The whole traffic actuated control functionality is implemented by the two source files of the example java application. Nevertheless the Sitraffic sX development environment basically enables the implementation of software packages with functionality like for example Sitraffic Language or PDM.

The whole example application is based on two source- and one external parameter file. If you have choose the default installation, the root path of the example is: C:\dws\TrafficActuationControlExample

Following the root path further to “\src\com\sitraffic\fd\ic\tacExample\” the three basic files are:

- StartTACExtern.xml, is the additional external configuration file for the java application. It is located at “\jappl\cfg\”.
- TacExampleStarter.java, reads the external configuration file and initializes the process.
- TacExampleImplementation.java, defines the main program flow of the example.

Each of the source files, except for the *.xml file, corresponds to a java class. The subsequent sections provide a brief introduction to the classes and their functionality.

```
<?xml version="1.0" encoding="UTF-8"?>
- <appStarter xsi:noNamespaceSchemaLocation="\csc\XML\AppStarter.xsd" xmlns:xsi="http:/
    <maxInitSteps>30</maxInitSteps>
    <name>TrafficActuationControlExample</name>
    <vmID>TACE</vmID>
  - <componentList>
    + <component modes="normal,stop">
    + <component>
    + <component>
    + <component modes="normal,stop">
    + <component>
    + <component>
    - <component>
        <name>TacExampleService</name>
        <className>com.sitraffic.fd.ic.tacExample.TacExampleStarter</className>
      - <paramMap>
        - <param>
            <paramKey>START_LEVEL</paramKey>
            <paramValue>5</paramValue>
          </param>
        - <param>
            <paramKey>occupiedStageNr</paramKey>
            <paramValue>4</paramValue>
          </param>
        - <param>
            <paramKey>idleStageNr</paramKey>
            <paramValue>2</paramValue>
          </param>
        - <param>
            <paramKey>triggeringDetectorNr</paramKey>
            <paramValue>1</paramValue>
          </param>
        </paramMap>
      </component>
    </componentList>
</appStarter>
```

Figure 19: Parameter file StartTACExtern.xml.

5.3.2.1. The StartTACExample Class

The main function of this class is the initialization and process handling for the TacExampleImplementation Class. It inherits some methods from the Application-Starter (AppStarter) framework.

For our example the relevant ones are:

- onInit(AppStarterComponentCfg cfg)
  - performs the construction of the TacExampleImplementation object
  - initializes the TacExampleImplementation object with parameters from StartTACExtern.xml (via the AppStarterComponentCfg cfg)
- onStartup()
  - calls to the start() method of the TacExampleImplementation object
- onShutdown()
  - calls to the shutdown() method of the TacExampleImplementation object

From a functional point of view the latter methods are triggered from the outside by the App Starter framework. When launching the example process, the methods “onInit” and “onStartup” are executed in sequence. The “onShutdown()” method is currently not used but nevertheless available.

With respect to the present example there are some functional implications. The TacExampleImplementation object is initialized only once by calling its constructor in the onInit method. Changing the values of some parameters in StartTACExtern.xm file will not show any effect. The java process has to be restarted, again executing the oninit method, for new parameters to be read. More detailed information about the App Starter framework is given in chapter 4. With respect to already existing traffic actuation methods, the basic one-time initialization of internal parameters might be performed here.

5.3.2.2. The external Parameter File StartTACExtern.xml

As already discussed, the TacExampleImplementation is initialized with external parameters originating from StartTACExtern.xml. The underlying data model, say the *.xsd, is part of the App Starter framework. Please note, that this is model is different from the basic Sitraffic sX configuration data model. The external *.xml file is a completely separate way to provide configuration. There are no automatic checks for consistency.

For the purpose if this example implementation we just need some basic knowledge about the *.xml structure. Opening “StartTACExtern.xml” in Eclipse essentially shows a list of key-value pairs. The IP-Addresses and the parameters for the control logic are the most important ones. The Occupied- and Idle-Stage as well as the Triggering Detector are defined by their numbers.

Utilizing that xml-structure, parameters are simply addressed by their key. An example is the onInit method of the StartTACExample Class. The parameter <paramValue> is address by its <paramKey>, using the getParam method on the AppStarterComponentCfg object. Obviously two of the parameters of the present example are not defined in the parameter *.xml file:

- occupiedStageActiveTime, default value = 65000 [ms]
- idleStageRefreshTime, default value = 60000 [ms]

As a result <paramValue> is not retrieved by the getParam method. For this case the method offers the possibility of defining a default value that is returned instead.

The case that <paramKey> is inconsistent with the basic configuration, e.g. addressing an unknown detector is more complex. Here the implementation needs to provide some suitable checkup and fallback mechanisms.

5.3.2.3. The TacExampleImplementation Class

With respect to program flow up to here the TacExampleImplementation has been constructed and initialized. It is started by the onStartup method in TacExampleStarter. Like with initialization, the execution of that method is triggered externally by the App Starter, finally calling the start method of TacExampleImplementation. That method comprises three callback-methods that are triggered externally.

- detDataStoreContainer -> processNewDetRawValues
- requestedStatusContainer -> statusChanged
- actualStatusContainer -> statusChanged

The source code example basically follows a Publish-Subscribe pattern.

5.3.2.3.1. Detector Data

Although the implementation example generically follows an event-based approach, the underlying controller architecture employs some clocking. The detector data for example is updated every 100 milliseconds. By construction this imposes the same 100ms clocking interval on the calls to the processNewDetRawValues method. With respect to already existing say established traffic actuation methods, the main control logic would probably be placed right here.

Inspecting processNewDetRawValues one finds most of the functionality of the previous Use Case (see 5.3.1). The basic one is that the stage-transition to the Occupied stage is only executed on detector actuation. If there is no actuation of the triggering detector, the switching request for Idle Stage is just refreshed when necessary (see 5.3.2.3).

5.3.2.3.2. Operational State

Unlike the detector data, the operational state is provided purely event-based. In the present example a change in requested- or actual status triggers the method statusChanged. The method itself mainly consists of checkups and choosing the internal status TacOperationalState accordingly. Depending on the outcome of the very checkup, the method setTaActualStatus is used for communicating an internal status change of the traffic actuation process back to the virtualized Sitraffic sX controller.

With respect to the functionality of the example process, most preconditions for starting the traffic actuation (see 5.3.1) are reflected in the checkup conditions. The intersection, traffic actuation as well as at least one control method have to be activated. Also a stage-oriented signal plan has to running. The shut-down functionality is also found which corresponds to the regular way (see 5.3.1.3.2) already discussed. Here the traffic actuations switching requests are erased by sending a request of duration zero.

5.3.3. Use Case: providing device values

A C-Control client that wishes to provide device values shall provide:

- a device value provider name and version
- device value meta definitions
- The values itself

5.3.3.1. Providing device value provider name and version

In general a c-control client has to register as device value provider see [6] registerDeviceVariableProvider_8(..)

Using the java CControlClientLib the DataReceiver registers the device value provider. The name and version are written into the AppStarter configuration StartTacExtern.xml as is with TacExample for documentation see [6] DataReceiverStarter.

5.3.3.2. Providing device value meta definitions

At c-control interface a device value provider has to return its device value meta definitions with [6] getDeviceValuesMetaData_8().

Using the java CControlClientLib you have to provide your implementation of Interface:

```
com.sitraffic.fd.ic.ccontrolClient.DeviceValueForwarder.DeviceValueTypeFactory
```

within your AppStarter configuration StartTacExtern.xml as is with TacExample for documentation see [6] DataReceiverStarter.

With the device value meta definitions a c-control client may also provide nls property files at directory <application.rootpath>/data/nls file name: <dvpName>V<dvpVersion>_<2 char country code>.properties.
For the tacExample the german nls definitions are looked up at: /opt/jappl/data/nls/TacExampleV1_de.properties.

5.3.3.3. Providing the device value itself

At c-control interface a device value provider sends its device values with an invocation of

```
com.sitraffic.fd.ic.ccontrol.rpc.CControl_CcontrolService_CltIF.setDeviceValues_8(
DeviceValuesCctrl dvs)
```

Using the java CControlClientLib you may use a DeviceValueForwarder interface that is obtained by:

```
DeviceValueForwarder dvf =
IfRegistry.getInstance().get(DeviceValueForwarder.class);
DeviceValueForwarder.Oitd myOitd= new Oitd(myMemberNo, myOtypeNo, myInstanceNo);
..
Int value =42;
dvf.update(myOitd, value);
```

5.3.4. Use Case: provide status

At c-control interface a client may provide status information by invoking [6] setCltStatus_8(java.lang.String host, int port, CltStatusMsgsCctrl status);

Using the java CControlClientLib you may use the com.sitraffic.fd.ic.ccontrolClient.CltStatusForwarder interface. At application startup the DataReceiver registers an CltStatusForwarder implementation at IfRegistry. This implementation may be obtained by IfRegistry.getInstance().get(CltStatusForwarder.class);

Each time the client application invokes setCltStatus_8() it shall pass all of its pending status information. Passing an empty array to setCltStatus will remove all status information for this client.

The format strings used with CfgMessages passed to setCltStatus are NLS translatable if the application has registered as device value provider (see 5.3.3). The key inside the nls-property file is derived in general by replacing ' ' by '_' and '%' by 'P'. if there is no nls key found, the passed format string, framed by '#' is used.

6. Index

D

V

Further information
is provided by:

Siemens AG
Infrastructure and Cities Sector
Mobility and Logistics Division
Road and City Mobility

Otto - Hahn - Ring 6
D-81739 München

The information in this manual contains descriptions and features which can change due to the development of products. The desired features are only binding if they were agreed upon conclusion of the contract.

Order no. A24730-A890-B003

Siemens Aktiengesellschaft

www.siemens.de/traffic