SEVENTH FRAMEWORK PROGRAMME

AsTeRICS Deliverable 4.5

Final Prototype of the AsTeRICS Runtime System

FHTW, UCY

Document Information

Issue Date	**30 June 2012**
Deliverable Number	**D4.5**
WP Number	**WP4 Software**
Status	**Final document**
Dissemination Level	**CO** **PU Public** **PP Restricted to other programme participants (including the Commission Services)** **RE Restricted to a group specified by the consortium (including the Commission Services)** **CO Confidential, only for members of the consortium (including the Commission Services)**

AsTeRICS – Assistive Technology Rapid Integration & Construction Set
Grant Agreement No.247730
ICT-2009.7.2 - Accessible and Assistive ICT
Small or medium-scale focused research project

Disclaimer

Version History

Version	Date	Changed	Author(s)
0.1	June 2nd, 2012	First draft	Chris Veigl (FHTW)
0.2	June 16th, 2012	ARE PT2 changes	Konstantinos Kakousis (UCY)
0.3	June 19th, 2012	CIM description	Christoph Weiss (FHTW)
0.4	June 21st, 2012	Added peer review comments and changes	Chris Veigl (FHTW)
0.5	June 25th, 2012	Integrated comments from peer review	Konstantinos Kakousis (UCY)

Table of Contents

1 Introduction

The purpose of this document is to provide a detailed description of the final version of the implemented AsTeRICS Runtime Environment (ARE). The ARE provides the necessary middleware and infrastructure to deploy AsTeRICS models for Assistive Technology applications. ARE models are serializable manifest files, describing the composition of sensor-, processor-, and actuator software plugins into complex Assistive Technology applications. Furthermore, the ARE manages remote connections to the graphical AsTeRICS Configuration Suite (ACS, D4.4 [1]) and provides necessary services for plugin developers. Some example services are connection management from/to Communication Interfaces Modules (CIMs), a logging and exception handling, data synchronization, GUI helper methods, etc.

In the current document, we provide a thorough overview of the ARE advances implemented since the release of the first prototype. The ARE has been extended to support more advanced operations and ease the rapid prototyping and execution of assistive applications. Based on the user evaluation results as well as feedback from plugin developers, the ARE has been modified to provide optimized performance, better support in displaying plugins' GUI elements, improved communication with the ACS and smoother communication between plugins and the AsTeRICS middleware.

The rest of this document is structured as follows: section 2 presents the new ARE GUI with a description of the API methods for plugin developers, section 3 presents the new ARE services and utilities that have been added since the first prototype. Section 4 concludes the deliverable.

1.1 Relationship to other AsTeRICS deliverables

This deliverable is related to the following AsTeRICS deliverables:

- D2.2 (Updated System Specification and Architecture) [2]: This document describes in detail the updates in hardware and software requirements for the AsTeRICS system as they have been depicted after the user tests and developers' feedback on PT1.
- D4.4 (Final Prototype of the AsTeRICS Configuration Suite) [1]: This document describes updates on the AsTeRICS configuration suite, some of these updates affect directly the runtime environment.
- Developer's Manual: the updated version of the developer's manual for the final AsTeRICS prototype contains further information about the architecture and the AsTeRICS SW-framework, including plugin development examples and how-to, ASAPI / Thrift updates, as well as updates on naming conventions.

2 The new Graphical User Interface

Although the ACS serves as the main graphical user interface unit in the AsTeRICS system, the ARE also provides a graphical user interface in order to allow end-users to interact directly with the runtime environment and monitor execution of running applications. In prototype 1 we presented a minimal 'control panel' for simulating ASAPI functions, mainly used for development and debugging purposes as an easy way for testing the deployment of various components and their reaction on various ASAPI commands.

The image below shows a screenshot from the PT1 ARE GUI. The old GUI was nothing more than a basic menu providing easy access to ASAPI commands. Although this was a handy way to test the ARE-ACS interconnection, we have decided to extend the ARE GUI for the second prototype.

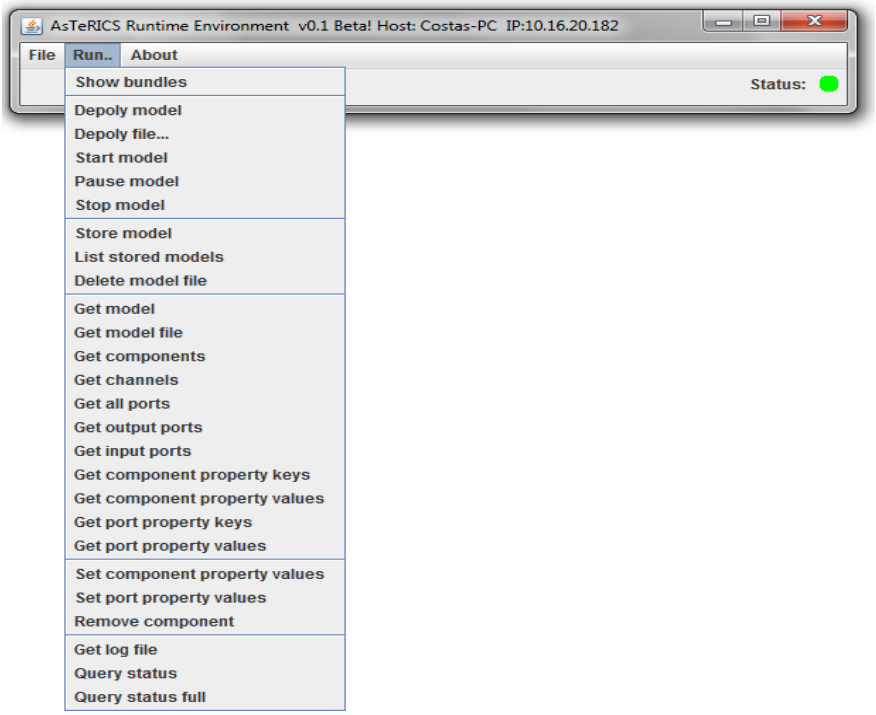

Figure 1: The ARE GUI in PT1

The ARE GUI for the second prototype has been evolved to a useful monitoring tool that can be used for better visualizing the status of the model - including the display of live signals and allows a GUI-based interaction with the runtime environment.

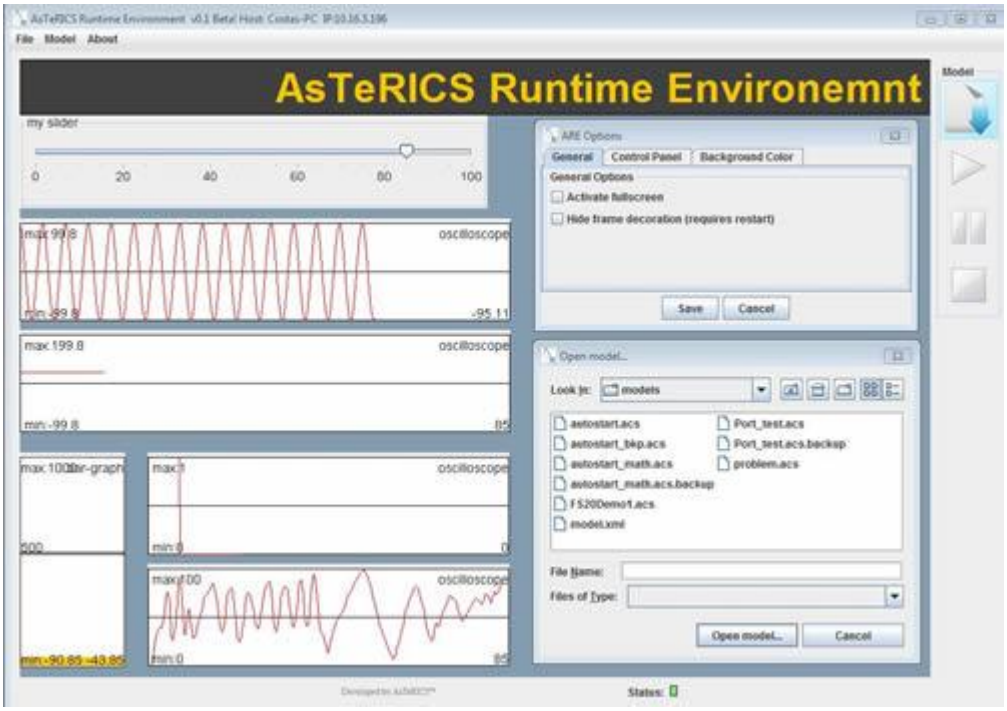

Figure 2: The ARE GUI in PT2

2.1 The main menu

As in PT1, we have a main menu with three items:

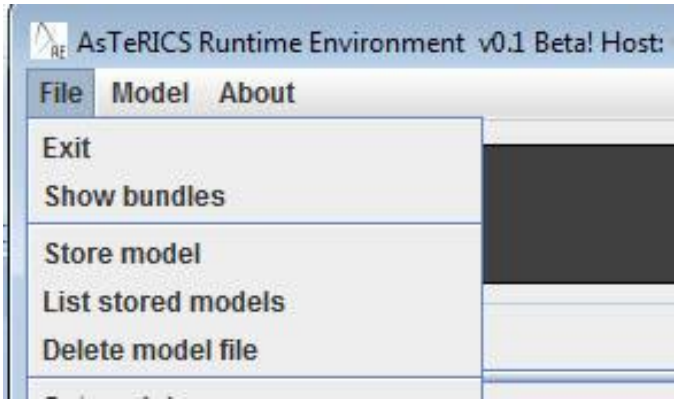

Figure 3: Main menu in the ARE GUI (PT2)

- **File**: provides access to main ASAPI functions as listed below:

Menu Item	ASAPI function	Description
Show bundles	–	Displays on the console a list of the plugins currently deployed on the ARE.
Store model	storeModel(String modelInXML, String filename)	Stores the currently deployed XML under a default name. A

		confirmation message is printed on the console.
List stored models	listAllStoredModels()	Displays on the console a list of all stored model file names found inside the default models folder.
Delete model file	deleteModelFile (String filename)	Deletes the default xml model. A confirmation message is printed on the console.
Get model	getModel()	Prints on the console the XML representation of the currently deployed model.
Get model file	getModelFromFile(String filename)	Prints on the console the XML representation of the default model.
Get components	getComponents()	Prints on the console the currently deployed component instances (including multiple instances of the same component type).
Get channels	getChannels(String componentID)	Prints on the console the IDs of all the channels that include the given component instance either as a source or target.
Get all ports	getAllPorts(String componentID)	Prints on the console the IDs of all the ports (i.e., includes both input and output ones) of the specified component instance.
Get output ports	getOutputPorts(String componentID)	Prints on the console the IDs of all the output ports of the given component instance.
Get input ports	getInputPorts(String componentID)	Prints on the console the IDs of all the input ports of the given component instance.

Get component property keys	getComponentPropertyKeys(String componentID)	Prints on the console the IDs of all properties set for the specified component.
Get component property values	getComponentProperty(String componentID, String key)	Prints on the console the value of the property with the specified key in the component with the specified ID.
Get port property keys	getPortPropertyKeys(String componentID, String portID)	Prints on the console the IDs of all properties set for the specified port.
Get port property values	getPortProperty(String componentID, String portID, String key)	Prints on the console the value of the property with the specified key of the port with the specified ID in the component with the specified ID.
Set component property values	setComponentProperty(String componentID, String key, String value)	Sets the property with the specified key in the component with the specified ID with the given string representation of the value and prints a confirmation message.
Set port property values	setPortProperty(String componentID, String portID, String key, String value)	Sets the property with the specified key in the port with the specified ID with the given string representation of the value and prints a confirmation message.
Remove component	removeComponent(String componentID)	Deletes the instance of the component that is specified by the given ID and prints a confirmation message.
Get log file	getLogFile()	Prints on the console the logged messages since the ARE instantiation.
Query status	queryStatus(boolean fullList)	Prints on the console errors that may have been reported from deployed components or the runtime

		environment itself. If the given argument is true the complete log history is printed, otherwise only messages that have not been printed in the past.

Table 1: the main ASAPI function, accessible via the File menu.

- **Model**: provides access to the main methods for interacting with models. From this menu item you can *Deploy*, *Start*, *Stop* or *Pause* a model. The deploy submenu item will open a file chooser window for selecting models to deploy from a local folder.

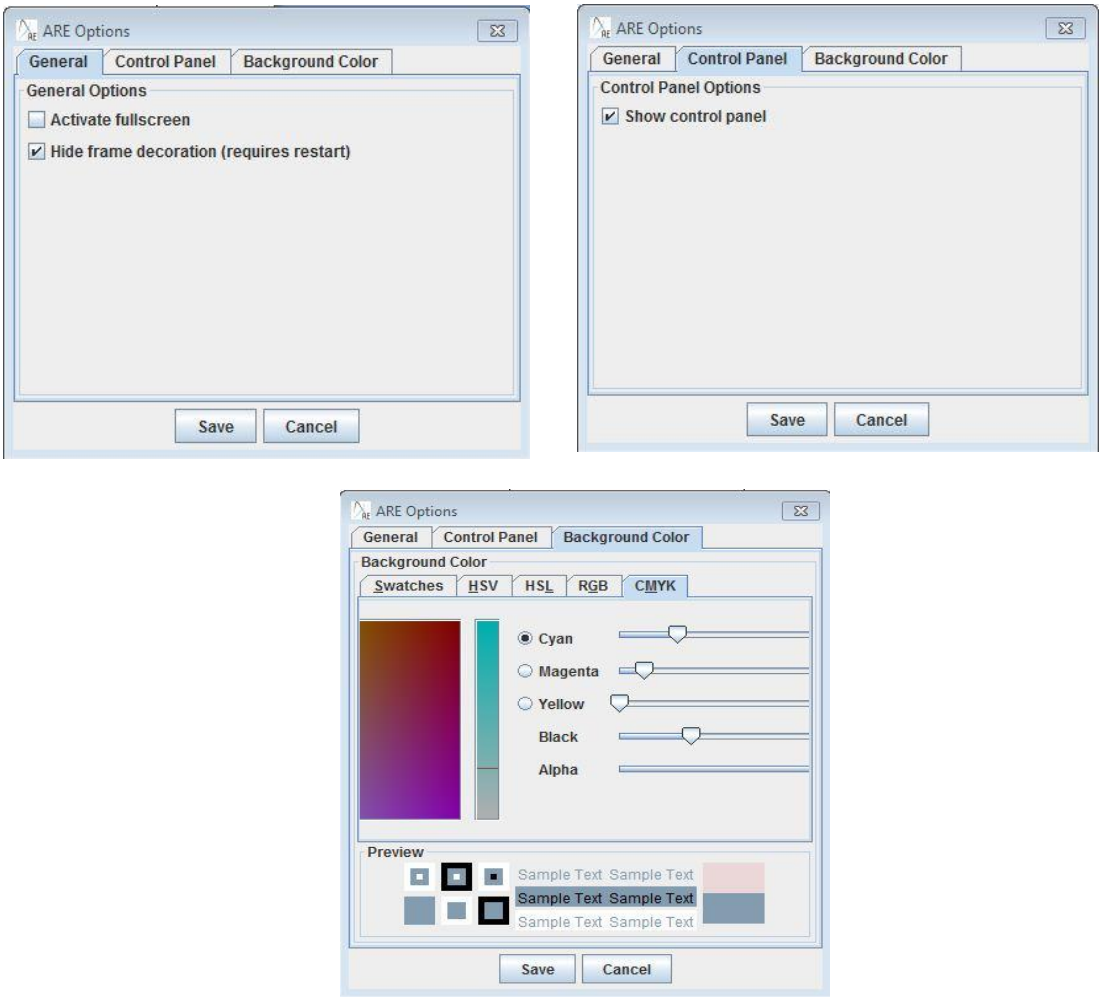

Figure 4: Options window in ARE GUI (PT2)

- **About/Options**: the new ARE GUI supports an options menu where various parameters can be set up depending on the usecase (see Figure 4). These options are particularly useful when we need to maximize available space for displaying plugin GUI elements on the personal device. The options are stored in system files and will be remembered the next time the ARE is deployed. In detail, the available options are:
    - **Activate Fullscreen**: If selected, the ARE widow will be maximized and set to fullscreen the next time we start ARE.
    - **Hide frame decoration**: If selected, window decorations will be hidden. In other words, the extra bar added by the runtime environment for minimizing/maximizing/closing the window will be hidden. This allows us to save some space as well as achieve better mapping between the ACS GUI Designer and the ARE main window (see Section 2.2). In order for this option to take effect, the ARE needs to be restarted since Java does not allow hiding/showing the frames after the application launch.

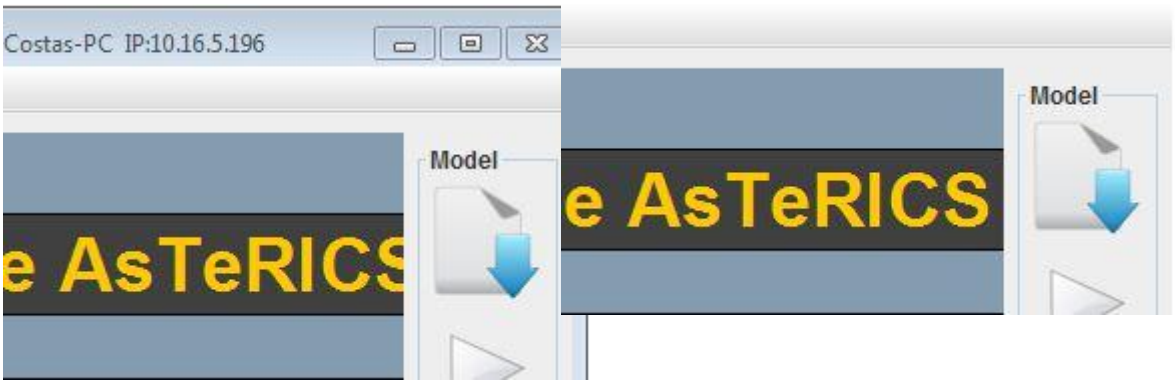

Figure 5: ARE GUI (PT2): with and without frame decoration.

- **Show control panel:** The control panel is a side frame acting as a toolbar for quick access to the main module functionalities, namely: *Deploy, Start, Pause* and *Stop.* Mainly for space reasons this side bar can be shown or hidden at will, using the Show Control panel option.
- **Background Color:** This option allows the end user to specify a color for the main window. It utilizes the built in color picker for selecting any color easily.
- **Other options:** In addition to the abovementioned options, the ARE will "remember" the last position and dimensions of the ARE window and will use these values next time the ARE is started.

Figure 6: ARE GUI (PT2): Control Panel.

2.2 The ARE Desktop

We refer to the ARE main window as ARE Desktop which is used for displaying GUI elements from plugins participating in a running model. A great advancement of the second prototype is the ability of "designing" the placement and size of the GUI elements directly in the ACS. The latter has been enriched with a drag n' drop editor where the designer can specify the position and size of any GUI element. Figure 7 shows an example of how the designed placement has been transmitted from the ACS GUI Designer to the ARE Desktop.

Figure 6: GUI composition using the ACS GUI designer

In order to achieve this, we have amended the deployment models with a gui element per Component which specifies the position and size as a percentage of available space (screen size). This means that we can perform changes on the way that windows are positioned "on the fly" i.e., without the need for starting/stopping ARE.

```
<xs:complexType name="guiType">
    <xs:sequence>
        <xs:element name="posX" minOccurs="1" maxOccurs="1" type="xs:integer"/>
        <xs:element name="posY" minOccurs="1" maxOccurs="1" type="xs:integer"/>
        <xs:element name="width" minOccurs="1" maxOccurs="1" type="xs:integer"/>
        <xs:element name="height" minOccurs="1" maxOccurs="1" type="xs:integer"/>
    </xs:sequence>
</xs:complexType>
```

Table 2: the gui element addition to the deployment model schema.

2.3 Middleware GUI Services

The AsTeRICS middleware provides some services to the plugin developers in order to allow them displaying their GUI element onto the ARE Desktop. The middleware services encapsulate the complexity of dealing with positioning and allow displaying all GUI elements onto the same container: the ARE Desktop.

All GUI services are defined in eu.asterics.mw.services.AREServices so developers need to import this class in order to get access to the following methods:

- *void displayPanel (JPanel panel, IRuntimeComponentInstance componentInstance, boolean display)*

  This method is used for displaying (or hiding) a plugin's panel at/from the ARE desktop. Developers need to pass

  - the panel they want to be displayed (or removed)
  - the plugin object, in order to help the middleware finding the desired position and dimensions from the deployment model
  - a boolean argument specifying if they wish to hide or show the given panel.

- *Dimension getAvailableSpace(IRuntimeComponentInstance componentInstance)*

  The space that each plugin will occupy on the ARE desktop is defined by the designer on the ACS and passed to the ARE via ASAPI. Plugin developers can get the available space for their graphical elements by calling the getAvailableSpace method which will return the space occupied for the plugin object passed as argument.

- *Point getComponentPosition (IRuntimeComponentInstance componentInstance)*

  The positioning of plugin's GUI elements is defined by the designer on the ACS and passed to the ARE via ASAPI. Plugin developers can get the position of their graphical elements by calling the getComponentPosition which will return the position on screen for the plugin object passed as argument.

- *void adjustFonts(JPanel panel, int maxFontSize, int minFontSize,int offset)*

  This service can be used by plugin developers interested in auto-adjusting the fonts of their GUI components depending on the space occupied for their plugins on the ARE desktop. They need to pass

  - a panel to which all the internal fonts will be auto-adjusted
  - the maximum font size (in case there is more space available than needed)
  - the minimum font size, in case there is too little space which causes the text to become non-readable. Finally, the offset argument is used in case we want to occupy a percentage of the available space

3 New ARE Services and Utilities

Deliverable D4.2 [3] already lists available ARE services that may be used by plugin developers in order to interact with the middleware. Section 2.3 already lists available GUI services. This section describes other services added since the first prototype.

3.1 ARE core events notification service

The ARE core events notification service allows plugins to register/unregister to the ARE middleware in order to receive notifications of ARE core events.

- *void registerAREEventListener(IAREEventListener clazz)*

  It is sometimes necessary that plugins can be notified of various ARE events so they can react as needed. This method can be called by component instances that wish to be notified of such ARE events. Currently, the core events supported are:

  - *preDeployModel:* registered ARE event listeners will be notified just before the deployment of a model.
  - *postDeployModel:* registered ARE event listeners will be notified immediately after the deployment of a model.
  - *preStartModel:* registered ARE event listeners will be notified just before the currently deployed model is started.
  - *postStopModel:* registered ARE event listeners will be notified immediately after the deployed model has been stopped.

- *void unregisterAREEventListener(IAREEventListener clazz)*

  Plugins already registered for receiving ARE core events can un-register using this method.

3.2 Data Synchronization

After the release of the first prototype, we received requests from plugin developers for a data synchronization service that will reduce the complexity of synchronizing incoming data at the input ports of plugins, by providing an abstraction for this synchronization in the middleware.

Some plugins need data of multiple ports to be available before they can start processing. Due to the threaded nature of our input-ports, it could happen that one input port of a plugin receives multiple values before another port gets one value, although both signal channels deliver values at the same sampling rate. Therefore, plugin developers were expected to provide a buffering mechanism at plugin level that will allow them to synchronize incoming data.

The synchronization service provides a buffering mechanism at the middleware level that can be utilized by plugin developers in order to make sure that incoming data of selected input ports arrives synchronized.

In prototype 2, plugin developers are expected to extend the *DefaultRuntimeInputPort* instead of implementing the *IRuntimeInputPort.* Basically, *DefaultRuntimeInputPort* provides a default implementation for the necessary buffering methods, as shown in the table below.

```
public abstract class DefaultRuntimeInputPort implements IRuntimeInputPort {
```

```
        private boolean buffering;
         public void receiveData(final byte [] data) {
                ;
         }
        public void startBuffering (AbstractRuntimeComponentInstance c,
                     String portID) {
              this.buffering = true;
        }
        public void stopBuffering (AbstractRuntimeComponentInstance c,
                     String portID) {
              this.buffering = false;
        }
        public boolean isBuffered () {return this.buffering;}
}
```

Table 3: the DefaultRuntimeInputPort abstract class.

The designer can define that a plugin's input port should be synchronized with some other input ports via the ACS. This will cause an argument change of the inputPort element on the deployment model file (e.g., <inputPort portTypeID="inB" **sync="true"**>). As soon as a model is deployed on the ARE, the middleware collects per component every port noted as synchronized port. When the model is successfully deployed and started, the ARE will buffer data which enters synchronized input ports until data on all synchronized ports has arrived. At that point, the ARE will call a new *AbstractRuntimeComponentInstance* callback method.

Developers that wish to support data synchronization need to implement the following method at their component instances.

public void syncedValuesReceived(HashMap<String, byte[]> dataRow)

Where dataRow is a HashMap between Input Port ID and byte[]. For synchronized input ports, instead of implementing the regular *void receiveData(byte[] data)* method which delivers incoming data of a single port, developers need to implement the *syncedValuesReceived* method which will be called from the ARE with synchronized data from all the input ports that have been selected.

3.3 Dynamic Properties

Another new feature introduced in the second prototype are dynamic properties. In some applications, developers needed a way to communicate to the ACS some “live” properties. A typical example is the wave file player plugin which plays sound files when an event occurs. Therefore, we needed a way to display available wave files in the ACS properties window. Apart from the wave file player plugin, this feature is particularly useful for plugins that are hardware dependent (selecting e.g. a soundcard or a midi player), or depend on the file system.

If a plugin is implementing a dynamic property, the values will be requested from the ARE, as soon as the ACS is synchronized with the ARE, via a new ASAPI function: *List<String> getRuntimePropertyList(String componentID, String key).*

The ARE middleware will forward the request for valid property values to the component instance with the given ID. For the second prototype we added the *List<String> getRuntimePropertyList(String key)* method to the *AbstractRuntimeComponentInstance* class which every AsTeRICS component extends. Therefore, any plugin that wishes to pass a

string list with dynamic property values (e.g. theavailable wave files) has to implement the *getRuntimePropertyList* method. Finally, as soon as the targeted component returns the string list to the middleware, the latter forwards the string list to the ACS via ASAPI. ACS will dynamically update the propertiy list in the properties window [1].

3.4 Secure Dynamic Plugin Loading

For the first prototype, the ARE by default was starting every plugin that was available at the AsTeRICS binary folder. This approach had several disadvantages: As new plugins continue to arrive and the number of bundles to be started on ARE startup is increased, unnecessary workload and performance delays occur. Furthermore, any dysfunctional plugin was preventing the ARE middleware bundle to start.

For the second prototype, we shifted to a dynamic plugin loading where only bundles participating in the deployed model are started. This change had significant performance improvements and allows us to avoid starting malfunctioning plugins that are not even necessary for a deployed model. The dynamic, programmatic plugin loading also allows us to prevent unexpected exceptions with non-working plugins. With the current approach, if a dysfunctional plugin is started, the ARE will detect the error and prompt a warning message to the user, letting him know what plugin is preventing the normal execution of the runtime environment. Finally, when a model is stopped, the ARE will also stop the participating plugins (or OSGi bundles) - thus taking full advantage of the dynamic bundle loading offered by the underlying OSGi framework.

3.5 Improved Thread Pooling

As described in [3], the ARE middleware makes use of the AsTeRICS Thread Pooling system. In particular, the **AstericsThreadPool** class is based on the newCachedThreadPool method of the java.util.concurrent.Executors class. The newCachedThreadPool method creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available. While this thread pooling approach has significant performance advantages, it was causing some synchronization problems on our thread-based data transition between ports. The synchronization problem was caused due to the fact that we do not have access to the number of threads created and the sequence of their execution using the *cachedThreadPool*. Therefore, pending tasks (waiting threads) were executed in a non-FIFO order which caused troubles in data processing.

For the second prototype, we utilize an additional thread pool only for sending data between ports. The new thread pool is based on the fixedThreadPool java method which is not multi-threaded and it is used for performing the data transportation in a sequential manner. The old thread pool is still used for any other thread execution in AsTeRICS.

3.6 Enhanced CIM Communication Services

For the final prototype, the integration of Bluetooth and Zigbee communication required certain changes in the CIM communication management. For a preparation to support different kinds of devices that use a COM port for communication, a generalized communication controller class, hiding the fact whether a common CIM or a CIM connected via wireless media, was extracted.

The listener interface for incoming data was changed in such a way that the ARE is not aware of whether a component is a peripheral adhering to the CIM protocol or any other module using its own protocol.

All incoming data is passed through the interface method:

```
public void handlePacketReceived(CIMEvent e);
```

The CIMEvent object contains the packet that has been transferred. This can be either a CIM protocol packet or a generic byte stream. The plugin receiving the notification should process the data correspondingly.

The use of serial input data for control applications proved that the approach using a decoupling of the monitor thread for incoming data and the notification of received data via a blocking queue introduced too much delay on the data and made control impossible. Thus for higher data rates, a further extension was needed.

3.6.1 Interface Unification

All access to the different types of serial communication is done via the class CIMPortManager. The class provides three types of connections:

```
public CIMPortController getConnection(short cimId)

public CIMPortController getRawConnection(String portName, int baudRate,
boolean highSpeed)
```

The method getConnection() returns a port controller for a requested CIM Id if this CIM has been detected on the platform. GetRawConnection returns either a raw port controller handling incoming data just like described above, or - if highSpeed is set - a high speed raw port controller where the decoupling between RX monitoring and processing thread has been removed and an input/output stream has been implemented.

The final prototype's hardware updates reduced the amount of digital inputs and outputs per CIM significantly. Thus it might be necessary to use two CIMs of the same type on one platform. During the first prototype, a way to uniquely identify a CIM was devised but it was not implemented in ARE. The method to identify a CIM is via the id tuple consisting of the CIM's type id and a unique serial number which is unique for every CIM of a specific type.

3.6.2 Uniquely Identifiable CIMs

For the final prototype, the CIM communication service was extended to support the use of two or more CIMs of the same type simultaneously. In the CIM identification phase, the CIM's unique number is saved with the CIM type and all the available CIM's are placed in a list. The plugin developer now has two options to access these CIMs:

```
public CIMPortController getConnection(short cimId)

public CIMPortController getConnection(short cimId, long uniqueNumber)
```

Calling the method getConnection() with only the CIM Id as parameter will return the first CIM of a certain type that was found. If a unique number is given, only the corresponding CIM connection will be returned or an error is raised if this uniquely identified CIM is not connected.

Thus it is necessary for the plugin developer to provide means to define how the CIM's unique number is to be passed to communication service methods if he deems it necessary to work with multiple CIMs. The CIM communication services provide a method to list all the attached unique numbers of CIMs of a certain type in order to allow the developer to work with dynamic properties for the CIM selection.

3.6.3 Bluetooth Problems

During the work on the first prototype, the automatic CIM detection occasionally caused long delays or hang ups of the ARE. Investigations led to the result that this was related to Bluetooth devices using the Microsoft Bluetooth stack. This stack creates two virtual COM ports, one for incoming data, one for outgoing.

Automatic detection of CIMs works by iterating through all COM ports and sending a CIM detection packet over the serial interface. Opening the Bluetooth ports caused numerous problems.

On the one hand it occurs that one port cannot send data, which will cause problems when trying to close the port as the event listener cannot be removed because the send operation is still pending. Opening the other port caused the PC to not recognize the Bluetooth ports anymore until the devices have been disconnected and paired again.

Analysis of the source code of RXTX (version 2.1.7) revealed that the first problem was caused by the detection of writeable COM ports. This is done by enumerating the COM ports via the Windows API CreateFile() function which hung upon opening the sender COM port. This could be remedied by using a different method to enumerate the serial ports, in this case the function QueryDosDevice() which returns a list of all the devices attached to a Windows installation. From within this list, all the COM ports can be extracted.

The second problem was caused by the use of overlapped (or asynchronous) I/O operation without timeout in the serial write function of the RXTX library. The fact that the receiver Bluetooth port did not allow GetOverlappedResult() to finish if no timeout was set, caused the serial port object to never finish the write operation and in the end locking the native code portions when the port was to be closed.

Introduction of a timeout to GetOverlappedResult with a repeat counter allowed to eliminate the problem and serial_write to fail silently. After these changes to the native code of RXTX, the automatic detection of CIMs attached via Bluetooth worked the same way as if the connection was established by wire.

4 Conclusions

Since the release of the first AsTeRICS prototype, we have been extending and stabilizing the runtime environment based on new requirements defined in [2] and feedback from our developers. Currently we have a stable runtime environment capable of satisfying the most needs of Assistive Technology developers and flexible enough to support new features.

In this document we have presented the main improvements of the ARE since the release of the first prototype. Although no drastic changes have emerged at the underlying architecture or core functionality, we have significantly improved the ARE GUI and improved the middleware services and utilities which are provided to the developers.

All major goals for the architectural framework implementations, which were derived from D2.2, have been accomplished. We will continue the ARE stabilization and fine tuning during WP5 integration tests.

References

1 AsTeRICS Deliverable D4.4 “Final Prototype of ACS”.

2 AsTeRICS Deliverable D2.2 “Updated System Specification and Architecture”.

3 AsTeRICS Deliverable D4.2 “Prototype 1 of the AsTeRICS Runtime System”.