Writing Correlator Plug-ins

5.2.0

August 2014

This document applies to Apama 5.2.0 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: PAM-Writing_Correlator_Plug-ins-5.2.0-20140808@233876

Table of Contents

Preface

About this documentation

Although the correlator's native programming language, the Apama Event Processing Language (EPL), has most of the functionality of modern programming languages, its primary purpose is enabling the detection of, correlation across, and triggering on complex event patterns. (Apama Event Processing Language is the new name for MonitorScript.)

In most cases existing code could be ported and rewritten in EPL, but in practice this might not be feasible. For example, an application might need to carry out advanced arithmetic operations and a significant programming library of such functions might already be available. Porting such complex code to EPL would be a lengthy, expensive and error prone task, and is unnecessary.

In order to incorporate existing specialized functionality, developers can write what is termed an *event correlator plug-in*. A correlator plug-in consists of an appropriately formatted library of C or C++ functions that can be called from within EPL code. In the case of a plug-in written in Java, the Java classes that are called from an application's EPL code are contained in a jar file. The event correlator does not need to be modified to enable or to integrate with a plug-in, as the plug-in loading process is transparent and occurs dynamically when required.

Custom correlator plug-ins can be developed using Apama's EPL Plug-in APIs for C, C++, and Java. Once a plug-in is developed, a developer can call the functions it contains directly from EPL code, passing EPL variables and literals as parameters, and getting return values that can be manipulated.

This document describes Apama's EPL Plug-in APIs version 5.2 and illustrates how to use them.

Preface

Documentation roadmap

On Windows platforms, the specific set of documentation provided with Apama depends on whether you choose the Developer, Server, or User installation option. On UNIX platforms, only the Server option is available.

Apama provides documentation in three formats:

- HTML viewable in a Web browser
- PDF
- Eclipse Help (if you select the Apama Developer installation option)

On Windows, to access the documentation, select Start > All Programs > Software AG > Apama 5.2 > Apama Documentation . On UNIX, display the `index.html` file, which is in the `doc` directory of your Apama installation directory.

The following table describes the PDF documents that are available when you install the Apama Developer option. A subset of these documents is provided with the Server and User options.

Title	Contents
What's New in Apama	Describes new features and changes since the previous release.
Installing Apama	Instructions for installing the Developer, Server, or User Apama installation options.
Introduction to Apama	Introduction to developing Apama applications, discussions of Apama architecture and concepts, and pointers to sources of information outside the documentation set.
Using Apama Studio	Instructions for using Apama Studio to create and test Apama projects; write, profile, and debug EPL programs; write JMon programs; develop custom blocks; and store, retrieve and playback data.
Developing Apama Applications in Event Modeler	Instructions for using Apama Studio's Event Modeler editor to develop scenarios. Includes information about using standard functions, standard blocks, and blocks generated from scenarios.
Developing Apama Applications in EPL	Introduces Apama's Event Processing Language (EPL) and provides user guide type information for how to write EPL programs. EPL is the native interface to the correlator. This document also provides information for using the standard correlator plug-ins.
Apama EPL Reference	Reference information for EPL: lexical elements, syntax, types, variables, event definitions, expressions, statements.
Developing Apama Applications in Java	Introduces the Apama in-process API for Java, referred to as JMon, and provides user guide type information for how to write Java programs that run on the correlator. Reference information in Javadoc format is also available.
Building Dashboards	Describes how to create dashboards, which are the end-user interfaces to running scenario instances and data view items.
Dashboard Property Reference	Reference information on the properties of the visualization objects that you can include in your dashboards.
Dashboard Function Reference	Reference information on dashboard functions, which allow you to operate on correlator data before you attach it to visualization objects.

Title	Contents
Developing Adapters	Describes how to create adapters, which are components that translate events from non-Apama format to Apama format.
Developing Clients	Describes how to develop C, C++, Java, or .NET clients that can communicate with and interact with the correlator.
Writing Correlator Plug-ins	Describes how to develop formatted libraries of C, C++ or Java functions that can be called from EPL.
Deploying and Managing Apama Applications	Describes how to: • Use the Management & Monitoring console to configure, start, stop, and monitor the correlator and adapters across multiple hosts. • Deploy dashboards over wide area networks, including the internet, and provide dashboards with effective authorization and authentication. • Improve Apama application performance by using multiple correlators, and saving and reusing a snapshot of a correlator's state. • Use the Apama ADBC adapter to store and retrieve data in JDBC, ODBC, and Apama Sim databases. • Use the Apama Web Services Client adapter to invoke Web Services. • Use correlator-integrated messaging for JMS to reliably send and receive JMS messages in Apama applications. • Use Universal Messaging to connect correlators.
Using the Dashboard Viewer	Describes how to view and interact with dashboards that are receiving run-time data from the correlator.

Preface

Contacting customer support

You may open Apama Support Incidents online via the eService section of Empower at http://empower.softwareag.com. If you are new to Empower, send an email to `empower@softwareag.com` with your name, company, and company email address to request an account.

If you have any questions, you can find a local or toll-free number for your country in our Global Support Directory at https://empower.softwareag.com/public_directory.asp and give us a call.

Preface

Chapter 1: Introduction to Correlator Plug-ins

Although the correlator's native programming language, the Apama Event Processing Language (EPL), has most of the functionality of modern programming languages, its primary purpose is enabling the detection of, correlation across, and triggering on complex event patterns. (Apama Event Processing Language is the newer name for MonitorScript.)

In most cases, existing code could be ported and rewritten in EPL, but in practice this might not be feasible. For example, an application might need to carry out advanced arithmetic operations and a significant programming library of such functions might already be available. Porting such complex code to EPL would be a lengthy, expensive and error prone task, and is unnecessary.

In order to incorporate existing specialized functionality, developers can write what is termed an *event correlator plug-in*. A correlator plug-in consists of an appropriately formatted library of C or C++ functions, which can be called from within EPL code. In the case of a plug-in written in Java, the Java classes that are called from an application's EPL code are contained in a `jar` file. The event correlator does not need to be modified to enable or to integrate with a plug-in, as the plug-in loading process is transparent and occurs dynamically when required.

Custom correlator plug-ins can be developed using Apama's EPL Plug-in APIs for C, C++, and Java. Once a plug-in is developed, a developer can call the functions it contains directly from EPL code, passing EPL variables and literals as parameters, and getting return values that can be manipulated.

Note: It is very important that strict plug-in development guidelines are followed when developing a plug-in. The functions provided must be adequately debugged prior to their integration within a plug-in. This is because when the event correlator loads a plug-in it is dynamically linked with the correlator's runtime process. If any code within the plug-in causes a runtime error the correlator might fail and terminate.

For this reason, Apama customers who experience problems with correlator stability while using plug-ins will be asked by Apama Technical Support to remove the plug-in and reproduce the problem prior to being offered further technical help. Apama Technical Support will only lift this restriction if the plug-ins have had prior certification by Apama.

"Writing a Plug-in in C or C++" on page 8 illustrates the plug-in development process through exploration of a simple example. "Advanced Plug-in Functionality in C++ and C" on page 18 takes this further with a more comprehensive example, while "The EPL Plug-in APIs for C and C++" on page 38 provides a more complete overview of the functionality of the EPL Plug-in C API and the EPL Plug-in C++ API. "Writing Correlator Plug-ins in Java" on page 54 describes the EPL Plug-in for Java.

Chapter 2: Writing a Plug-in in C or C++

The Apama EPL Plug-in APIs for C and a C++ make it possible for developers to write event correlator plug-ins either in C or in C++.

As long as certain conventions are followed, writing a plug-in is very straightforward. In essence, a plug-in consists of a set of static C or C++ functions (representing the functionality that the developer wishes to invoke from within EPL code) and some necessary initialization functions.

C++ compilers vary extensively in their support for the ISO C++ standard and in how they support linking. For this reason Apama supports the writing of C++ plug-ins only with specific compilers. For a list of the supported C++ compilers, see Software AG's Knowledge Center in Empower at https://empower.softwareag.com.

On the other-hand, C has been standardized for many years, and for this reason the C API should work with the majority of modern C/C++ compilers on all platforms.

The EPL Plug-in APIs are versioned. For a correlator plug-in to be compatible with an event correlator they both need to support the same plug-in interface version. Plug-ins built with earlier versions of the APIs need to be re-compiled and re-linked. Note that the API version number is separate from the product version number and only increases when the plug-in ABI is changed.

The APIs comprise the relevant header files, `correlator_plugin.hpp` and `correlator_plugin.h`, and are accompanied by a set of sample applications.

To configure the build for a correlator plug-in:

- On Linux, copying and customizing an Apama makefile from a sample application is the easiest method.
- On Windows, you might find it easiest to copy an Apama sample project. If you prefer to use a project you already have, be sure to add `$(APAMA_HOME)\include` as an include directory. To do this in Visual Studio, select your project and then select Project Properties > C/C++ > General > Additional Include Directories.

A simple plug-in in C++

As an example, this topic describes the development of a simple plug-in called, appropriately, `simple_plugin`. It has only one function, called `test`, which takes a string as its sole parameter, makes some alterations to it, prints it out, and passes back another string as the result.

Let us first consider a C++ example using the C++ API. The first requirement is to include the header file `correlator_plugin.hpp`. This header file contains the definitions for the C++ API. The file is located in the Apama installation's `include` directory.

The C++ method that implements this plug-in function must be defined as follows:

```
class SimplePlugin {
  public:
  static void AP_PLUGIN_CALL test(
    const AP_Context& ctx,
    const AP_TypeList& args,
    AP_Type& rval,
    AP_TypeDiscriminator rtype);
}
```

The definition for all plug-in functions must be as for `SimplePlugin::test` above. In essence only the method name and the enclosing class name should vary as far as the definition is concerned. This is important, since it is the Correlator's Plug-in Support Mechanism that will be calling this C++ method and filling in its parameters.

- The `ctx` parameter is known as the execution context and is used internally by the event correlator to make the call to the plug-in function. The developer normally need not be concerned with it in the function's implementation.
- `args` is an array of parameters; in effect the parameters that the EPL writer will have to supply when calling `test`.
- `rval` denotes the return value. Plug-in function implementations must pass out any return value through this parameter, although as will be shown, in EPL the function will appear to return a result in the traditional way. The return value can be a `float`, `boolean`, `string`, `integer` or `chunk`, and the expected return type is indicated by `rval.discriminator()`. It is not possible to return sequences or other correlator types.
- `rtype` is the expected return type, identical to the result of calling `rval.discriminator()` and is retained for backwards compatability.

The next important step is to define exactly what type of parameters the above plug-in function should expect and accept, what it should return, and under what name it should appear within EPL.

```
/** Parameter types for the 'test' function */
static const char8* testParamTypes[1] = {"string"};
/** Declare functions provided by this plugin */
static AP_Function Functions[1] = {
  {"test", &SimplePlugin::test, 1, &testParamTypes[0], "string"}
};
```

The static array of `AP_Functions` structures needs to be defined in every plug-in to describe which functions that plug-in is exporting to the event correlator. In this case the C++ method `SimplePlugin::test` has been mapped to appear as the external plug-in function "test", to take a single parameter, with the latter being of the EPL type `string` (as defined within `testParamTypes`), and return a value of EPL type `string`. If a particular function returns nothing, the return type should be specified as `void`.

All that is left is to implement the "C" plug-in initialization method:

```
AP_PLUGIN_DLL_SYM AP_ErrorCode AP_PLUGIN_CALL
  AP_INIT_FUNCTION_NAME {
    const AP_Context& ctx,
    uint32& version,
    uint32& nFunctions,
    AP_Function*& functions
);
```

the "C" plug-in shutdown method:

```
AP_PLUGIN_DLL_SYM AP_ErrorCode AP_PLUGIN_CALL
  AP_SHUTDOWN_FUNCTION_NAME (const AP_Context& ctx);
```

and the "C" plug-in library version check:

```
AP_PLUGIN_DLL_SYM AP_ErrorCode AP_PLUGIN_CALL
  AP_LIBRARY_VERSION_FUNCTION_NAME(const AP_Context& ctx,
 uint32& version);
```

The names of the functions are macros defined in `correlator_plugin.hpp`.

Linking limitations require that these three functions be defined as "C" functions. Both should at least implement the code as indicated in the full source code example below. For most situations it is recommended that the developer re-deploy the initialization and shutdown methods provided unchanged, although more complex plug-ins may include plug-in-specific startup and shutdown code in these functions. Note that the initialization and shutdown functions are invoked each time the library is loaded or unloaded, so these functions must be re-entrant and able to be safely invoked multiple times.

Going back to the implementation of the test method, through use of the extensive library of helper functions available on the `AP_Type` class, the developer can manipulate the values passed through by the EPL code.

For example, this code displays an integer argument passed to a function:

```
cout << args[0].integerValue();
```

while this call increments the second element of a `sequence` argument:

```
AP_Type &element = args[0][2];
element.integerValue(element.integerValue()+1);
```

Note that this is relevant since `sequence`s and `chunk`s are passed by reference. So, if the EPL code calling it was:

```
sequence<integer> mySeq := [0,10,20,30];
myPlugin.exampleFunction(mySeq);
print mySeq;
```

then after the call `mySeq` is [0,10,21,30].

Note that this is modifying the `sequencemySeq` refers to, not altering the value of `mySeq` itself. A plug-in function cannot do the equivalent of: `mySeq := otherSeq;`

Similarly, it is not possible to modify primitives passed to a plug-in as arguments. Strings, while strictly speaking a reference type, are immutable and so cannot be modified either.

The complete code base of this simple example is as follows, and may be found in the Apama installation's `samples\correlator_plugin\cpp\simple_plugin.cpp` directory. A 'makefile' (for use with GNU Make) and batch file (for Microsoft's Visual Studio .NET) are provided in this folder to assist with compiling plug-ins on UNIX and Windows platforms respectively. A `README.txt` file in the directory describes how to build the example.

```
/**
 * simple_plugin.cpp
 *
 * A very simple example plugin.
 *
 * $Copyright(c) 2013 Software AG, Darmstadt, Germany and/or its licensors$
 * $Revision: 188575 $ $Date: 2012-07-17 20:22:08 +0100 (Tue, 17 Jul 2012) $
 */

#include <correlator_plugin.hpp>
#include <iostream>
#include <string>
#include <cstring>

// Used in the test function below
#define TEST_STRING "Hello, World"
```

```
using namespace std;

/**
 * Plugin implementation class.  Contains *static* members matching the
 * AP_FunctionPtr typedef in "correlator_plugin.hpp".  These could also be
 * provided as static top-level functions with C++ linkage.
 */
class SimplePlugin {

public:
 /**
  * Plugin function.  Takes a single string parameter and returns
  * another string.  The input string will also be modified (pass by
  * reference for non-primitive types).
  *
  * @param ctx Execution context for this invocation of the function.
  * @param args Function parameters, as declared below.
  * @param rval Function return value, to be filled in by plugin.
  *
  * @throw AP_PluginException If anything goes wrong.
  */
 static void AP_PLUGIN_CALL test(
        const AP_Context& ctx,
        const AP_TypeList& args,
        AP_Type& rval,
        AP_TypeDiscriminator rtype) {
  cout << "simple_plugin function test called" << endl;
  cout << "args[0] = " << args[0].stringValue() << endl;
  // This demonstrates the use of the new string allocation
  // functions in AP_Context.  Note that string assignment copies
  // the string, so we must free our local copy to avoid a memory
  // leak.
  char8* newStr = ctx.char8alloc(strlen(TEST_STRING) + 1);
         // +1 for the terminator!
  strcpy(newStr, TEST_STRING);
  rval = newStr;
  ctx.char8free(newStr);
  cout << "return value = " << rval.stringValue() << endl;
  cout.flush();
 }
};

/** Parameter types for the 'test' function */
static const char8* testParamTypes[1] = { "string" };

/** Declare functions provided by this plugin */
static AP_Function Functions[1] = {
 { "test", &SimplePlugin::test, 1, &testParamTypes[0], "string" }
};

/**
 * Initialisation and shutdown functions.  Must be declared extern "C" so that
 * the plugin loader can look them up by name.  Signatures must match the
 * AP_InitFunctionPtr and AP_ShutdownFunctionPtr typedefs, respectively.
 * Likewise, the function names must use the AP_INIT_FUNCTION_NAME and
 * AP_SHUTDOWN_FUNCTION_NAME macros.
 */
extern "C" {

/**
 * Plugin initialisation function.  Called each time the plugin library is
 * loaded, so must expect to be called more than once.
 *
 * @param ctx Execution context for the function call.
 * @param version Active plugin API version.  The plugin should verify it
 * is compatible with this version (and return AP_VERSION_MISMATCH_ERROR
 * if not) and update version to indicate the API version the plugin was
 * compiled against.
 * @param nFunctions The plugin should set this to the number of functions
```

```
 * it exports.
 * @param functions The plugin should set this to the address of an array
 * of AP_Functions structures, of length nFunctions, describing the
 * functions exported by the plugin.  This data will be read and copied by
 * the plugin API.
 *
 * @return An AP_ErrorCode indicating the success or otherwise of the
 * library initialisation.
 */
AP_PLUGIN_DLL_SYM AP_ErrorCode AP_PLUGIN_CALL AP_INIT_FUNCTION_NAME(
     const AP_Context& ctx,
     uint32& version,
     uint32& nFunctions,
     AP_Function*& functions) {
 // Export the plugin's version
 version = AP_PLUGIN_VERSION;
 // Export function declarations
 nFunctions = 1;
 functions = &Functions[0];
 return AP_NO_ERROR;
}

/**
 * Plugin shutdown function.  Called each time the plugin library is
 * unloaded, so must expect to be called more than once.
 *
 * @param ctx Execution context for the function call.
 *
 * @return An AP_ErrorCode indicating the success or otherwise of the
 * library initialisation.
 */
AP_PLUGIN_DLL_SYM AP_ErrorCode AP_PLUGIN_CALL AP_SHUTDOWN_FUNCTION_NAME(
    const AP_Context& ctx) {
 // Nothing to do -- just indicate success
 return AP_NO_ERROR;
}

/**
 * Plugin library version function.
 *
 * @param ctx Execution context for the function call.
 *
 * @param version The plugin should fill this in with the version of the
 * API it was compiled against.
 *
 * @return An AP_ErrorCode indicating the success or otherwise of the
 * function call.
 */
AP_PLUGIN_DLL_SYM AP_ErrorCode AP_PLUGIN_CALL AP_LIBRARY_VERSION_FUNCTION_NAME(
       const AP_Context& ctx,
       uint32& version) {
 // Just return the version number
 version = AP_PLUGIN_VERSION;
 return AP_NO_ERROR;
}

/**
 * Plugin capabilities function.
 *
 * @param ctx Execution context for the function call.
 *
 * @return An AP_Capabilities bitset describing the capabilities of this
 * plugin.
 */
AP_PLUGIN_DLL_SYM AP_Capabilities AP_PLUGIN_CALL AP_PLUGIN_GET_CAPABILITIES_FUNCTION_NAME(
          const AP_Context& ctx) {
 // This plugin promises no special capabilities. But if you wanted to
 // declare any, you could return them, or-ed together.
 return AP_CAPABILITIES_NONE;
}
```

```
}  // extern "C"
```

Writing a Plug-in in C or C++

Calling the test function from EPL

Compiling `simple_plugin.cpp` produces the plug-in file `libsimple_plugin.so` (on UNIX) or `simple_plugin.dll` (on Windows).

The plug-in needs to be placed in a location where it can be picked up by the event correlator.

This means that on Windows you either need to copy the `.dll` into the `bin` folder of the Apama installation, or else place it somewhere which is on your 'path', that is a location that is referenced by the `PATH` environment variable.

On Linux or Solaris you either need to copy the `.so` into the `lib` directory of your Apama installation, or else place it somewhere which is on your 'library path', that is a directory that is referenced by the `LD_LIBRARY_PATH` environment variable.

The next step is to write some EPL code that imports the `simple_plugin` plug-in and calls the method `test`.

Some example EPL code to achieve this is as follows:

```
monitor SimplePluginTest {

  // Load the plugin
  import "simple_plugin" as simple;

  // To hold the return value
  string ret;
  string arg;
  action onload {

    // Call plugin function
    arg := "Hello, Simple Plugin";
    ret := simple.test(arg);

    // Print out return value
    log "simple.test = " + ret;
    log "arg = " + arg;
  }
}
```

Firstly, `simple_plugin` must effectively be located and loaded. This is the first purpose of the `import` statement. Secondly it must be assigned an alias name, in this case `simple`.

This then allows the plug-in's test method to be invoked as `simple.test()`, taking an EPL `string` as parameter, and returning an EPL code `string` as its result.

The above EPL code is provided as `simple_plugin.mon` in the Apama installation's `samples\correlator_plugin\cpp` directory.

Writing a Plug-in in C or C++

A simple C plug-in

The C version of the above example, `simple_plugin.c`, is very similar. The first difference is the use of the C version of the API, which is `correlator_plugin.h`. This can be located in the `include` directory of the Apama installation.

As before, there is only one function, called `test`, which takes a string as its sole parameter, makes some alterations to it, prints it out, and passes back another string as the result.

The C method that implements this plug-in function must be defined as follows:

```
static void AP_PLUGIN_CALL simplePluginTest(
  const AP_PluginContext* ctx,
  const AP_PluginTypeList* args,
  AP_PluginType* rval,
  AP_TypeDiscriminator rtype)
```

The rest of the example is very similar to the C++ example. The complete code base is as follows, and may be found in the Apama installation's `samples\correlator_plugin\c\simple_plugin.c` directory. A 'makefile' (for use with GNU Make) and batch file (for Microsoft's Visual Studio .NET) are provided in this folder to assist with compiling plug-ins on Unix and Windows platforms respectively. A `README.txt` file in the directory describes how to build the plug-in.

```
 /**
 * simple_plugin.c
 *
 * A very simple example plugin.
 *
 * $Copyright(c) 2013 Software AG, Darmstadt, Germany and/or its licensors$
 *
 * $Revision: 188575 $ $Date: 2012-07-17 20:22:08 +0100 (Tue, 17 Jul 2012) $
 */
#include <correlator_plugin.h>

#include <stdio.h>

/* Used in the test function below */
#define TEST_STRING "Hello, World"

/**
 * Plugin function.  Takes a single string parameter and returns
 * another string.  The input string will also be modified (pass by
 * reference for non-primitive types).
 *
 * @param ctx Execution context for this invocation of the function.
 * @param args Function parameters, as declared below.
 * @param rval Function return value, to be filled in by plugin.
 */
static void AP_PLUGIN_CALL simplePluginTest(
              const AP_PluginContext* ctx, const AP_PluginTypeList* args,
              AP_PluginType* rval, AP_TypeDiscriminator rtype)
{
 AP_PluginType* arg;
 printf("simple_plugin function test called\n");
 arg = args->functions->getElement(args,0);
 printf("args[0] = %s\n",arg->functions->getStringValue(arg));
 rval->functions->setStringValue(rval,TEST_STRING);
 printf("return value = %s\n",rval->functions->getStringValue(rval));
 fflush(stdout);
}

/** Parameter types for the 'test' function */
static const AP_char8* testParamTypes[1] = { "string" };

/** Declare functions provided by this plugin */
static AP_PluginFunction Functions[1] = {
 { "test", &simplePluginTest, 1, &testParamTypes[0], "string" }
};
```

```
/**
 * Initialisation and shutdown functions. Signatures must match the
 * AP_PluginInitFunctionPtr and AP_PluginShutdownFunctionPtr typedefs,
 * respectively. Likewise, the function names must use the
 * AP_INIT_FUNCTION_NAME and AP_SHUTDOWN_FUNCTION_NAME macros.
 */

/**
 * Plugin initialisation function.  Called each time the plugin library is
 * loaded, so must expect to be called more than once.
 *
 * @param ctx Execution context for the function call.
 *
 * @param version Active plugin API version.  The plugin should verify it
 * is compatible with this version (and return AP_VERSION_MISMATCH_ERROR
 * if not) and update version to indicate the API version the plugin was
 * compiled against.
 *
 * @param nFunctions The plugin should set this to the number of functions
 * it exports.
 *
 * @param functions The plugin should set this to the address of an array
 * of AP_Functions structures, of length nFunctions, describing the
 * functions exported by the plugin.  This data will be read and copied by
 * the plugin API.
 *
 * @return An AP_ErrorCode indicating the success or otherwise of the
 * library initialisation.
 */
AP_PLUGIN_DLL_SYM AP_ErrorCode AP_PLUGIN_CALL AP_INIT_FUNCTION_NAME(
               const AP_PluginContext* ctx, AP_uint32* version,
               AP_uint32* nFunctions, AP_PluginFunction** functions)
{
 /* Export the plugin's version */
 *version = AP_CORRELATOR_PLUGIN_VERSION;
 /* Export function declarations */
 *nFunctions = 1;
 *functions = &Functions[0];
 return AP_NO_ERROR;
}

/**
 * Plugin shutdown function.  Called each time the plugin library is
 * unloaded, so must expect to be called more than once.
 *
 * @param ctx Execution context for the function call.
 *
 * @return An AP_ErrorCode indicating the success or otherwise of the
 * library initialisation.
 */
AP_PLUGIN_DLL_SYM AP_ErrorCode AP_PLUGIN_CALL AP_SHUTDOWN_FUNCTION_NAME(
           const AP_PluginContext* ctx) {
 /* Nothing to do -- just indicate success */
 return AP_NO_ERROR;
}

/**
 * Plugin library version function.
 *
 * @param ctx Execution context for the function call.
 *
 * @param version The plugin should fill this in with the version of the
 * API it was compiled against.
 *
 * @return An AP_ErrorCode indicating the success or otherwise of the
 * function call.
 */
AP_PLUGIN_DLL_SYM AP_ErrorCode AP_PLUGIN_CALL AP_PLUGIN_VERSION_FUNCTION_NAME(
           const AP_PluginContext* ctx, AP_uint32* version) {
 /* Just return the version number */
```

```
 *version = AP_CORRELATOR_PLUGIN_VERSION;
 return AP_NO_ERROR;
}

/**
 * Plugin capabilities function.
 *
 * @param ctx Execution context for the function call.
 *
 * @return An AP_Capabilities bitset describing the capabilities of this
 * plugin.
 */
AP_PLUGIN_DLL_SYM AP_Capabilities AP_PLUGIN_CALL AP_PLUGIN_GET_CAPABILITIES_FUNCTION_NAME(
            const AP_PluginContext* ctx) {
 // This plugin promises no special capabilities. But if you wanted to
 // declare any, you could return them, or-ed together.
    return AP_CAPABILITIES_NONE;
}
```

Writing a Plug-in in C or C++

Parameter-less plug-in functions

Occasionally it is useful to invoke a function or method within a plug-in which requires, and returns, no parameters. This is simply achieved by having the function/method ignore the function/method parameters and defining a function which takes no parameters and returns `void` in the function table. For example:

C++

In C++ the method is defined as:

```
void AP_PLUGIN_CALL Analytic::SilentInitialisation (
  const AP_Context& ctx,
  const AP_TypeList& args,
  AP_Type& rval,
  AP_TypeDiscriminator rtype) {
  // Custom Code Here
  // Ignoring the args, rval and rtype parameters
}
```

C

And, in C as:

```
static void AP_PLUGIN_CALL SilentInitialisation (
  const AP_PluginContext* ctx,
  const AP_PluginTypeList* args,
  AP_PluginType* rval,
  AP_TypeDiscriminator rtype)  {
    // Custom Code Here
    // Ignoring the args, rval and rtype parameters
}
```

Then the function table would appear thus, in C++:

```
Static AP_Function Functions[1] = {
  {
    "SilentInit", &Analytics:: SilentInitialisation, 0, NULL,
    "void" },
};
```

And as below in C:

```
Static AP_Function Functions[1] = {
  { "SilentInit", &SilentInitialisation, 0, NULL, "void" },
};
```

EPL

In EPL, the plug-in function/method is then invoked as:

```
import "analytics" as a;
action onload {
  a.SilentInit();
  // Custom Code Here
}
```

Returning multiple values

Each call to a plug-in function returns a single value. Occasionally it is necessary for an operation to return multiple values; there are various techniques that can be used to achieve this:

- Provide multiple functions which are called in turn, each of which returns one of the values.
- Return a `chunk` expressing the composite value, and provide functions that interrogate the `chunk` to extract each individual value.
- Return a `string` that can be parsed as an event that expresses the composite value.
- Enqueue an event that expresses the composite value.
- Pass the function a `string` and modify the elements.

See "The chunk type" on page 19 for details of how chunks are used and "Asynchronous plug-ins" on page 31 for how to enqueue an event from a plug-in.

Writing a Plug-in in C or C++

Chapter 3: Advanced Plug-in Functionality in C++ and C

This topic uses the `simple_plugin` example described in "Writing a Plug-in in C or C++" on page 8. This section extends the example and illustrates more advanced use of the APIs.

Introducing complex_plugin

Appropriately, this extended C++ example is called `complex_plugin.cpp`, and it is also available in the `samples\correlator_plugin\cpp` directory of the Apama installation. A `README.txt` file in the directory describes how to build the example plug-in.

This time the C++ example has three plug-in C++ methods defined:

```
class ComplexPlugin {
  public:

    static void AP_PLUGIN_CALL test1(
      const AP_Context& ctx,
      const AP_TypeList& args,
      AP_Type& rval,
      AP_TypeDiscriminator);

    static void AP_PLUGIN_CALL test3(
      const AP_Context& ctx,
      const AP_TypeList& args,
      AP_Type& rval,
      AP_TypeDiscriminator);

    static void AP_PLUGIN_CALL test4(
      const AP_Context& ctx,
      const AP_TypeList& args,
      AP_Type& rval,
      AP_TypeDiscriminator);
}
```

`ComplexPlugin::test1` dynamically decodes and displays its arguments, and then modifies the contents of any sequences that are passed to it. `ComplexPlugin::test3` allocates and returns an `ExampleChunk` *opaque object*. Opaque objects will be discussed shortly. `ComplexPlugin::test4` uses an `ExampleChunk` object as created by `ComplexPlugin::test3`, modifies and prints its contents, and then returns it.

You may have noticed that no `test2` was defined. This is intentional and the reason will become evident shortly.

In order to map the above C++ methods to plug-in functions one must define the `Functions` static array. This time this looks as follows:

```
static const char8* test1ParamTypes[4] =
  {"integer", "float", "boolean", "string"};

static const char8* test2ParamTypes[4] =
  {"sequence<integer>", "sequence<float>", "sequence<boolean>",
   "sequence<string>"};

static const char8* test3ParamTypes[1] = {"integer"};
static const char8* test4ParamTypes[1] = {"chunk"};

static AP_Function Functions[4] = {
  {"test1",&ComplexPlugin::test1,4,&test1ParamTypes[0],"string"},
  {"test2",&ComplexPlugin::test1,4,&test2ParamTypes[0],"float"},
  {"test3",&ComplexPlugin::test3,1,&test3ParamTypes[0],"chunk"},
  {"test4",&ComplexPlugin::test4,1,&test4ParamTypes[0],"void"}
};
```

This definition highlights some of the powerful capabilities available to plug-in developers.

- First of all it maps `ComplexPlugin::test1` to the plug-in method `test1`, indicates that it takes four EPL parameters, sets these to be an `integer`, a `float`, a `boolean` and a `string` respectively, and sets the return type to be a `string`.
- It then maps `ComplexPlugin::test1` (again!) to the plug-in method `test2`, this time indicating that it will take four EPL parameters, and sets these to be a `sequence` of `integer`, a `sequence` of `float`, a `sequence` of `boolean` and a `sequence` of `string`, respectively. It then sets the return type to be a `float`. It is important to note that this multiple mapping of the same C++ method can only be carried out if the method is written with no assumptions regarding the type of its parameters or result. In fact, if you examine the full source code for this example, as provided below, you will see that this method examines the parameters' types before manipulating them.
- `ComplexPlugin::test3` is mapped to `test3` and set to take a single `integer`. Interestingly though, it is set to return a `chunk` type. This is a special purpose *opaque type*. For an explanation of this type see the next section.
- `ComplexPlugin::test4` is mapped to `test4`, and accepts a `chunk` type. Its implementation is designed to work on the `chunk` result produced by `ComplexPlugin::test3`. It does not return a value.

Advanced Plug-in Functionality in C++ and C

The chunk type

Apama's Plug-in Support Mechanism assumes that the functions called are stateless, that is they do not retain state between calls. However, it is recognized that in some circumstances a developer might need to retain complex state in between function calls and in order to assist in this the opaque type `chunk` is provided. Furthermore, the `chunk` type allows data to be referenced from EPL that has no equivalent EPL type.

It is not possible to perform operations on data of type `chunk` from EPL code directly; it exists purely to allow 'pass-through' of data output by one external plug-in function to another function. The event correlator does not modify the internal structure of `chunk` values in any way, so as long as a receiving function expects the same type as that output by the original function, any complex data structure can be passed around using this mechanism.

Note: Chunks cannot be routed, emitted or enqueued. Also note that passing a chunk created by one plug-in to a second plug-in in the same monitor is not permitted. If one plug-in returns a chunk and a second plug-in tries to read it, a C++ exception will be thrown within the second plug-in and, unless it is caught, the exception will terminate the correlator instance.

To use `chunk`s with plug-ins first requires declaring a variable of type `chunk`. It can then be assigned the return value from an external function or used as a parameter in the function call.

The following example illustrates this. Monitor `printTime` prints out the current time when it is loaded. To generate `timeString` the monitor uses an external time plug-in. In this plug-in, the `time()` function returns a `float` representing the time in seconds; `localtime()` returns a structure containing year, month, day and time data which the `asctime()` function formats into a `string` of the form: "`Friday February 1 15:00:07 GMT 2002`".

```
import "apama_time" as time;

monitor printTime {
  float millis;
  chunk timeData;
  string timeString;

  action onload {
    millis := time.time();
    timeData := time.localtime(millis);
    timeString := time.asctime(timeData);
    print "The time is " + timeString;
  }
}
```

It can be seen that the `timeDatachunk` is used to store output from `localtime()` and pass it to `asctime()`; the value is not inspected from EPL code directly.

Although the `chunk` type was designed to support unknown data types, it is also a useful mechanism to improve performance. Where data returned by external library functions does not need to be accessed from the EPL code, using a `chunk` can cut down on unnecessary type conversion. For example, in the above example the output of `localtime()` is actually a 9-element array of `float`. The fact that the value is never accessed by the EPL code means that it can be declared as a `chunk` and an unnecessary conversion from native array to an EPL `sequence` and back again is removed.

Advanced Plug-in Functionality in C++ and C

Working with chunk in C++

A `chunk` object points to an instance of a class derived from the class `AP_Chunk`.

In this example `ComplexPlugin::test3` and `ComplexPlugin::test4` communicate state through the use of the same `chunk`, with this being of the type `ExampleChunk`. The `ExampleChunk` class is defined as follows:

```
/**
 * Simple 'chunk' class demonstrating how opaque, plugin-private data
 * may be passed between plugin functions by MonitorScript. Note that
 * every chunk class must be derived from AP_Chunk.
 */
```

```
class ExampleChunk : public AP_Chunk {
public:
 /**
  * Construct an ExampleChunk containing the specified number of
  * floating-point values.
  */
 explicit ExampleChunk(size_t size=2048);

 /**
  * Note that we can rely on the default copy constructor and
  * destructor
  */

 /**
  * Copy method creates a new ExampleChunk that is an exact duplicate
  * of the current object. This method must be provided by every chunk
  * class, so that the Engine can assign to and from chunk objects.
  */
 AP_Chunk* copy(const AP_Context& ctx) const;

 /**
  * Print out the contents of the chunk
  */
 void print() const;

 /** The contents of this chunk */
 std::vector<float64> data;
};
```

For every chunk sub-class, you need to define the copy (or cloning) method `copy()`. When the chunk is no longer needed by the correlator, it is deleted. As for any other C++ class, it is important to ensure that the destructor releases any resources or memory owned by the instance, though best practice is for the class's members to manage their own resources, as with `std::vector<float64> data` in `ExampleChunk`. In the `correlator_plugin.hpp` header file these are defined as:

```
/**
 * Pure virtual destructor.  It is *essential* that every AP_Chunk
 * derived class implements this method to free any resources
 * allocated by the derived class.  The Engine will arrange for the
 * destructor to be called when the associated MonitorScript chunk
 * object is deleted.
 *
 * The correlator interface may not be used from a chunk's destructor.
 *
 * Also, correlator interface calls on another thread may
 * block until the chunk's destructor returns. Chunk
 * destructors that can block should therefore be careful to
 * avoid deadlocking against such a thread.
 */
 virtual ~AP_Chunk() {}

/**
 * Chunk cloning method (typically calls a copy constructor in the
 * derived class).  As with the destructor, it is essential for
 * AP_Chunk derived classes to implement this method, to ensure that
 * MonitorScript assignments to/from chunk objects work correctly.
 *
 * @param ctx Execution context for the copy operation.
 *
 * @return Pointer to a copy of this AP_Chunk object.
 */
virtual AP_Chunk* copy(const AP_Context& ctx) const = 0;
```

The contents of these methods depend on what the `chunk` is intended to contain. In this example the `chunk` is intended to store `data`, a vector of `float` values. Because `std::vector`, and therefore also `ExampleChunk`, is copy-constructible, returning `new ExampleChunk(*this);` suffices. Only if the type is not copy-constructible will `copy()` need to do anything more elaborate. The destructor needs to adequately de-allocate the memory assigned to this structure. Both methods are used implicitly by

EPL: the event correlator will invoke the destructor when the `chunk` object is no longer accessible to the EPL code, and will call the `copy()` method as necessary to handle EPL assignments.

It is important to note that if plug-in function code invokes the `chunk` destructor itself, it should first call `chunkValue(NULL)` on the associated `AP_Type` object, to prevent the event correlator from attempting to delete the same object again.

The `size` member is then being used to keep track of the size of the `data` member. Two constructors and a utility `print()` method have also been provided in this case.

Advanced Plug-in Functionality in C++ and C

Working with chunk in C

In C, working with `chunk`s is similar. Functions that carry out the functionality that would otherwise be defined within the class methods need to be implemented. There are two specific rules that must be followed.

First a callback function table must be supplied with every user `chunk` that is created. Here's an example;

```
const struct AP_PluginChunk_Callbacks exampleChunkCallbacks = {
  exampleChunkFreeUserData,
  exampleChunkCopyUserData,
};
```

This specifies the functions that represent the 'destroy' and 'copy' functions described in the above C ++ sections.

The other rule is that the user must implement a `chunk` 'constructor' like method. Its contents or name do not matter, but it must return a specific structure that is obtained through calling the `createChunk` function. Here's an example constructor;

```
AP_PluginChunk* exampleChunkConstructor(
  const AP_PluginContext *ctx, unsigned size)
{
  struct ExampleChunk* data;

  data = (struct ExampleChunk*)malloc(
    sizeof(struct ExampleChunk));
  data->size = size;
  data->data = (double *)malloc(sizeof(double) * size);
  printf(
    "ExampleChunk constructor called with size = %u\n",size);
  return ctx->functions
    ->createChunk(ctx,&exampleChunkCallbacks,data);
}
```

Advanced Plug-in Functionality in C++ and C

Working with sequence

Sequences are the most complex type currently supported in the API. As the `DumpAP_Type` function in the example demonstrates, `AP_Type` functions and operators exist to:

- Get the number of elements in a `sequence`.
- Get the type of the `sequence` elements.

- Extract a single element from the `sequence`, as an `AP_Type`.

```
cout << "sequence type = " << arg.sequenceType() << endl;
cout << "sequence size = " << arg.sequenceLength() << endl;
for (uint32 i = 0; i < arg.sequenceLength(); i++) {
  cout << "sequence element[" << i << "]: ";
  DumpAP_Type(arg[i]);
  ModifyAP_Type(arg[i]);
}
```

It is also possible to map some or all of the `sequence` elements onto traditional C/C++ arrays, consisting either of `AP_Type` objects encapsulating the individual elements of the `sequence`, or of the "native" objects stored in each element. For example, the elements of an EPL `sequence<integer>` object can be mapped onto a native `int64` array like this:

```
int64 * intArray = arg.integerSequenceElements();
for (uint32 i = 0; i < arg.sequenceLength(); i++) {
  cout << intArray[i] << endl;
}
```

Alternatively, a "slice" containing a range of elements from the `sequence` can be mapped. The example below maps elements 20 through 59 of the `sequence` onto a native `int64` array of length 40. Note that an exception will be thrown if the specified slice lies outside the bounds of the `sequence`.

```
int64* intSlice = arg.integerSequenceElements(20, 40);
for (uint32 i = 0; i < 40; i++) {
  cout << intSlice[i] << endl;
}
```

Mapping `sequence` elements in this way may be relatively inefficient. EPL `sequence`s are not necessarily stored as native object arrays internally, so it may be necessary to copy the element data into the native array when performing the mapping. Likewise, the EPL `sequence` must be updated to reflect any changes to the elements made by the plug-in function, before returning to EPL. This latter operation is achieved by the family of `release`*<type>*`SequenceElements()` functions. For the integer `sequence` in the example above, it is necessary to call

```
sequence.releaseIntegerSequenceElements();
```

before returning from the plug-in function. Note that this will immediately invalidate the arrays returned by *all* calls to `integerSequenceElements()` made by the plug-in function, so it should typically only be used once per function invocation, once the function is finished with any mapped `sequence` data.

Any necessary `release`*<type>*`SequenceElements()` calls will in fact be made automatically when a plug-in function terminates. These functions are provided to plug-in writers so that mapped data can be released early if it is necessary to make memory available. It is also possible to access elements of a `sequence` using the visitor idiom by calling `visitSequenceElements` with an appropriate functor. Although less convenient than other `sequence` element accessors, it is more efficient as it entails no memory allocation.

Advanced Plug-in Functionality in C++ and C

The complete example

The following complete listing of `complex_plugin.cpp` contains the implementation of the `ComplexPlugin::` methods and the `ExampleChunk` class. The implementation of `ComplexPlugin::test1` is particularly interesting as it demonstrates how to use the functionality provided by the EPL Plug-in C++ API to examine the type of a parameter and act accordingly.

The equivalent C example is supplied in the installation as `complex_plugin.c` in the `samples/c` folder of the Apama installation.

The Plug-in initialization and shutdown methods are as used within `simple_plugin`.

```
/**
 * complex_plugin.cpp
 *
 * A more interesting example plugin.
 *
 * $Copyright(c) 2013 Software AG, Darmstadt, Germany and/or its licensors$
 *
 * $RCSfile$ $Revision: 188571 $ $Date: 2012-07-17 19:13:07 +0100 (Tue, 17 Jul 2012)$
 */

/**
 ** Please read and understand simple_plugin.cpp before using this file!
 **/

#include <correlator_plugin.hpp>
#include <iostream>
#include <string>
#include <vector>
#include <math.h>

using std::cout;
using std::endl;

/**
 * Plugin implementation class.
 */
class ComplexPlugin {

public:
 /**
  * Plugin function that dynamically decodes and displays its
  * arguments, and modifies them for return to the caller where
  * possible.
  */
 static void AP_PLUGIN_CALL test1(
                 const AP_Context& ctx, const AP_TypeList& args,
                 AP_Type& rval, AP_TypeDiscriminator);

 /**
  * Plugin function that allocates and returns a new ExampleChunk
  * opaque object.
  */
 static void AP_PLUGIN_CALL test3(
                 const AP_Context& ctx, const AP_TypeList& args,
                 AP_Type& rval, AP_TypeDiscriminator);

 /**
  * Plugin function that uses an ExampleChunk created by test3().
  */
 static void AP_PLUGIN_CALL test4(
                 const AP_Context& ctx, const AP_TypeList& args,
                 AP_Type& rval, AP_TypeDiscriminator);

 /**
  * Helper method to display the contents of an AP_Type and, if it
  * is a sequence, modify the elements
  */
 static void DumpAP_Type(const AP_Type& arg);

 /**
  * Helper method to modify the contents of an AP_Type
  */
 static void ModifyAP_Type(AP_Type& arg);
};
```

```
/**
 * Simple 'chunk' class demonstrating how opaque, plugin-private data
 * may be passed between plugin functions by MonitorScript. Note that
 * every chunk class must be derived from AP_Chunk.
 */
class ExampleChunk : public AP_Chunk {
public:
 /**
  * Construct an ExampleChunk containing the specified number of
  * floating-point values.
  */
 explicit ExampleChunk(size_t size=2048);

 /**
  * Note that we can rely on the default copy constructor and
  * destructor
  */

 /**
  * Copy method creates a new ExampleChunk that is an exact duplicate
  * of the current object. This method must be provided by every chunk
  * class, so that the Engine can assign to and from chunk objects.
  */
 AP_Chunk* copy(const AP_Context& ctx) const;

 /**
  * Print out the contents of the chunk
  */
 void print() const;

 /** The contents of this chunk */
 std::vector<float64> data;
};

/**
 * Implementation of test1 and test2 plugin functions.
 */
void AP_PLUGIN_CALL ComplexPlugin::test1(
                 const AP_Context& ctx, const AP_TypeList& args,
                 AP_Type& rval, AP_TypeDiscriminator) {
 // Display all the input parameters
 cout << "test1(): arg list length = " << args.size() << endl;
 for (uint32 i = 0; i < args.size(); i++) {
  DumpAP_Type(args[i]);
 }

 // Set the return value appropriately for its type. Note the use
 // of overloaded assignment operators to store values of various
 // types in an AP_Type object.
 switch (rval.discriminator()) {
 case AP_NULL_TYPE:
  // 'void' return; no value needed
  break;
 case AP_CHUNK_TYPE: {
  // Create a new chunk
  rval = new ExampleChunk;
  break;
 }
 case AP_INTEGER_TYPE:
  // Some compilers require this cast to distinguish between
  // integer and boolean assignment.
  rval = (int64)42;
  break;
 case AP_FLOAT_TYPE:
  // Make sure that float literals include a decimal point (or use
  // scientific notation) so that float, rather than integer or
  // boolean, assignment is used here.
  rval = 2.71828;
  break;
```

```
 case AP_BOOLEAN_TYPE:
  rval = false;
  break;
 case AP_STRING_TYPE:
  // String value will be copied by the assignment
  rval = "Hello, World";
  break;
 case AP_SEQUENCE_TYPE:
  // Sequence returns are not yet supported
  throw AP_UnimplementedException("test1(): Attempt to return sequence from plugin");
  break;
 default:
  // Unknown return type
  throw AP_TypeException("test1(): Unknown return type in plugin");
 }
}

/**
 * Implementation of test3 plugin function.
 */
void AP_PLUGIN_CALL ComplexPlugin::test3(
                 const AP_Context& ctx, const AP_TypeList& args,
                 AP_Type& rval, AP_TypeDiscriminator) {
 // args[0] contains desired size for chunk.
 ExampleChunk* chunk = new ExampleChunk(args[0].integerValue());

 // Put known values in the chunk data
 for (uint32 i = 0; i < chunk->data.size(); i++) {
  chunk->data[i] = i;
 }

 // Print chunk contents
 chunk->print();

 // Return the new chunk object
 rval = chunk;
}

/**
 * Implementation of test4 plugin function.
 */
void AP_PLUGIN_CALL ComplexPlugin::test4(
                 const AP_Context& ctx, const AP_TypeList& args,
                 AP_Type& rval, AP_TypeDiscriminator) {
 // Extract the chunk object from args[0], making sure to cast it to the
 // correct derived type (the plugin must know what this should be)
 ExampleChunk* chunk = (ExampleChunk*)(args[0].chunkValue());

 // Do some computation on the chunk data
 for (uint32 i = 0; i < chunk->data.size(); i++) {
  chunk->data[i] = sqrt(chunk->data[i]);
 }

 // Print chunk contents
 chunk->print();
}

/**
 * Dump the contents of arg to cout. Sequences will be recursively
 * expanded and modified.
 */
void ComplexPlugin::DumpAP_Type(const AP_Type& arg) {
 cout << "discriminator = " << arg.discriminator() << endl;
 switch (arg.discriminator()) {
 case AP_NULL_TYPE: {
  cout << "null type has no value" << endl;
  break;
 }
 case AP_CHUNK_TYPE: {
  cout << "chunk type is opaque" << endl;
```

```
  break;
 }
 case AP_INTEGER_TYPE: {
  cout << "integer value = " << arg.integerValue() << endl;
  break;
 }
 case AP_FLOAT_TYPE: {
  cout << "float value = " << arg.floatValue() << endl;
  break;
 }
 case AP_BOOLEAN_TYPE: {
  cout << "boolean value = " << arg.booleanValue() << endl;
  break;
 }
 case AP_STRING_TYPE: {
  cout << "string value = " << arg.stringValue() << endl;
  break;
 }
 case AP_SEQUENCE_TYPE: {
  cout << "sequence type = " << arg.sequenceType() << endl;
  cout << "sequence size = " << arg.sequenceLength() << endl;
  for (uint32 i = 0; i < arg.sequenceLength(); i++) {
   cout << "sequence element[" << i << "]: ";
   DumpAP_Type(arg[i]);
   ModifyAP_Type(arg[i]);
  }
  break;
 }
 default: {
  cout << "unknown discriminator value" << endl;
 }
 }
}

/**
 * Modify a value, if possible
 *
 * Note that if the argument is a sequence, it doesn't modify the
 * elements - that's handled by DumpAP_Type.
 */
void ComplexPlugin::ModifyAP_Type(AP_Type& arg) {
 switch (arg.discriminator()) {
 case AP_INTEGER_TYPE:
  arg = arg.integerValue() + 1;
  break;
 case AP_FLOAT_TYPE:
  arg = arg.floatValue() * 2.0;
  break;
 case AP_BOOLEAN_TYPE:
  arg = !arg.booleanValue();
  break;
 case AP_STRING_TYPE:
  arg = "How long is a piece of string?";
  break;
 default:
  break;
 }
}


/**
 * ExampleChunk constructor.
 */
ExampleChunk::ExampleChunk(size_t size) : data(size) {
 cout << "ExampleChunk constructor called with size = " << size << endl;
}

/**
 * ExampleChunk copy method. Implements virtual method declared by AP_Chunk
 * interface.
```

```
 */
AP_Chunk* ExampleChunk::copy(const AP_Context &) const {
 return new ExampleChunk(*this);
}

/**
 * Print the contents of the chunk.
 */
void ExampleChunk::print() const {
 cout << "Chunk size = " << data.size() << endl;
 for (uint32 i = 0; i < data.size(); i++) {
  cout << "Chunk element [" << i << "] = " << data[i] << endl;
 }
}

/** Parameter types for the plugin functions */
static const char8* test1ParamTypes[4] = { "integer", "float",
        "boolean", "string" };
static const char8* test2ParamTypes[4] = { "sequence<integer>",
        "sequence<float>", "sequence<boolean>",
        "sequence<string>" };
static const char8* test3ParamTypes[1] = { "integer" };
static const char8* test4ParamTypes[1] = { "chunk" };

/** Declare functions provided by this plugin */
static AP_Function Functions[4] = {
 { "test1", &ComplexPlugin::test1, 4, &test1ParamTypes[0], "string" },
 { "test2", &ComplexPlugin::test1, 4, &test2ParamTypes[0], "float" },
 { "test3", &ComplexPlugin::test3, 1, &test3ParamTypes[0], "chunk" },
 { "test4", &ComplexPlugin::test4, 1, &test4ParamTypes[0], "void" }
};

/**
 * Initialisation and shutdown functions. Must be declared extern "C" so that
 * the plugin loader can look them up by name. Signatures must match the
 * AP_InitFunctionPtr and AP_ShutdownFunctionPtr typedefs, respectively.
 * Likewise, the function names must use the AP_INIT_FUNCTION_NAME and
 * AP_SHUTDOWN_FUNCTION_NAME macros.
 */
extern "C" {

/**
 * Plugin initialisation function. Called each time the plugin library is
 * loaded, so must expect to be called more than once.
 *
 * @param ctx Execution context for the function call.
 * @param version Active plugin API version. The plugin should verify it
 * is compatible with this version (and return AP_VERSION_MISMATCH_ERROR
 * if not) and update version to indicate the API version the plugin was
 * compiled against.
 * @param nFunctions The plugin should set this to the number of functions
 * it exports.
 * @param functions The plugin should set this to the address of an array
 * of AP_Functions structures, of length nFunctions, describing the
 * functions exported by the plugin. This data will be read and copied by
 * the plugin API.
 *
 * @return An AP_ErrorCode indicating the success or otherwise of the
 * library initialisation.
 */
AP_PLUGIN_DLL_SYM AP_ErrorCode AP_PLUGIN_CALL AP_INIT_FUNCTION_NAME(
                 const AP_Context& ctx, uint32& version, uint32&
                 nFunctions, AP_Function*& functions) {
 version = AP_PLUGIN_VERSION;
 nFunctions = 4;
 functions = &Functions[0];
 return AP_NO_ERROR;
}

/**
```

```
 * Plugin shutdown function. Called each time the plugin library is
 * unloaded, so must expect to be called more than once.
 *
 * @param ctx Execution context for the function call.
 *
 * @return An AP_ErrorCode indicating the success or otherwise of the
 * library initialisation.
 */
AP_PLUGIN_DLL_SYM AP_ErrorCode AP_PLUGIN_CALL AP_SHUTDOWN_FUNCTION_NAME(
                  const AP_Context& ctx) {
 return AP_NO_ERROR;
}

/**
 * Plugin library version function.
 *
 * @param ctx Execution context for the function call.
 *
 * @param version The plugin should fill this in with the version of the
 * API it was compiled against.
 *
 * @return An AP_ErrorCode indicating the success or otherwise of the
 * function call.
 */
AP_PLUGIN_DLL_SYM AP_ErrorCode AP_PLUGIN_CALL AP_LIBRARY_VERSION_FUNCTION_NAME(
                  const AP_Context& ctx, uint32& version) {
 // Just return the version number
 version = AP_PLUGIN_VERSION;
 return AP_NO_ERROR;
}

/**
 * Plugin capabilities function.
 *
 * @param ctx Execution context for the function call.
 *
 * @return An AP_Capabilities bitset describing the capabilities of this
 * plugin.
 */
AP_PLUGIN_DLL_SYM AP_Capabilities AP_PLUGIN_CALL
       AP_PLUGIN_GET_CAPABILITIES_FUNCTION_NAME(
                  const AP_Context& ctx) {
 // This plugin promises no special capabilities. But if you wanted to
 // declare any, you could return them, or-ed together.
 return AP_CAPABILITIES_NONE;
}

} // extern "C"
```

Advanced Plug-in Functionality in C++ and C

Using complex_plugin from the event correlator

Example EPL code that imports this plug-in and uses its functionality is provided below. The file, `complex_plugin.mon`, is located in the `samples\correlator_plugin\cpp` folder of the Apama installation.

```
//
// complex_plugin.mon
//
// MonitorScript program to test complex example plugin
//
monitor ComplexPluginTest {

  // Load the plugin
  import "complex_plugin" as complex;
  // To hold the return values
  string str1;
```

```
  string ret1;
  float ret2;

  // Opaque chunk value
  chunk myChunk;

  // Loop counter
  integer i;

  // Sequences for test2
  sequence<integer> intSeq;
  sequence<float> floatSeq;
  sequence<boolean> boolSeq;
  sequence<string> stringSeq;

  action onload {
    // Call test1 function
    str1 := "Hello, Complex Plugin";
    ret1 := complex.test1(42, 3.14159, true, str1);
    log "complex.test1 = " + ret1 at INFO;
    log "str1 = " + str1 at INFO;

    // Initialise sequences
    intSeq.setSize(10);
    floatSeq.setSize(10);
    boolSeq.setSize(10);
    stringSeq.setSize(10);
    i := 0;
    while (i < 10) {
      intSeq[i] := i;
      floatSeq[i] := i.toFloat();
      boolSeq[i] := false;
      stringSeq[i] := i.toString();
      i := i + 1;
    }

    // Call test2 function
    ret2 := complex.test2(intSeq, floatSeq, boolSeq, stringSeq);
    log "complex.test2 = " + ret2.toString() at INFO;
    i := 0;
    while (i < 10) {
      log "intSeq[" + i.toString() + "] = " +
        intSeq[i].toString() at INFO;
      log "floatSeq[" + i.toString() + "] = " +
        floatSeq[i].toString() at INFO;
      log "boolSeq[" + i.toString() + "] = " +
        boolSeq[i].toString() at INFO;
      log "stringSeq[" + i.toString() + "] = " +
        stringSeq[i].toString() at INFO;
      i:= i + 1;
    }

    // Generate a new chunk
    myChunk := complex.test3(20);
    // Do some computation on the chunk
    complex.test4(myChunk);
  }
}
```

Once more the monitor starts by importing `complex_plugin`, this time mapping it to the alias `complex`.

After defining a number of variables, it calls `complex.test1`. This function displays the number of arguments and then displays them. It also returns the `string` value "`Hello, World`", which is then stored in `ret1`.

The call to `complex.test2` requires setting up the `sequences` it takes as parameters. As the implementation of `test2` within `complex_plugin` is effectively the same as `test1`, this does the same; it

displays the number of arguments and then displays each one, in this case printing out the contents of every `sequence`. The `float` value `2.71828` is returned instead.

For `complex.test3` the monitor is creating a `chunk`. The `test3` method will create a `chunk` with a numeric array of the specified size 20, which it initializes with the numbers 1 to 20. It then prints the contents out and returns the `chunk` to the event correlator for retaining in `myChunk`.

The event correlator cannot examine or manipulate `myChunk`, but `myChunk` can be passed in to other plug-in methods that expect a `chunk` of the same type. Note that the type of a `chunk`, in this case the C++ class `ExampleChunk`, is not visible in EPL, so it is up to the developer to ensure the `chunks` are compatible across plug-in methods. This broadly applies to all plug-in methods irrespective of parameter and return types. The developer must ensure that the parameters passed are of the correct types as otherwise failure might occur.

`Thecomplex.test4` method is called with `myChunk`. This traverses the array of floating point numbers contained within and takes the square root of each one. It then prints out the revised numbers. It does not return anything.

Advanced Plug-in Functionality in C++ and C

Asynchronous plug-ins

It is possible to write a plug-in that can send events asynchronously to the event correlator. This is not a recommended technique as multiple correlator processes or external processes connected via the client API are preferred approaches to scaling Apama deployments. However, an example of how to implement asynchronous plug-ins is available in the `samples\correlator_plugin\cpp` folder of the Apama installation, and is called `async_plugin.cpp`. Since this is a simple example, only an overview is provided here rather then the complete sample.

This sample uses the `getCorrelator()` method of `AP_Context` to get a reference to an `AP_CorrelatorInterface`.

The single public method of `AP_CorrelatorInterface` is declared as follows:

```
/**
* Send an event to the correlator
*
* @param event the event to send.  The event is represented as a string
* using the format described in Deploying and Managing Apama Apps.
* See the correlator utilities section, Event File Format.
*/
virtual void sendEvent(const char* event) const = 0;
```

The event correlator implements this method by using the same event queuing and asynchronous processing mechanism as is used for the EPL `enqueue` keyword.

In this sample, the plug-in has one function exposed that is also called `sendEvent`. This function demonstrates the feature by simply sending the data it was given back to the event correlator. A more elaborate use of this mechanism might use its own background processing thread to occasionally send events to the event correlator.

Examples of using the `sendEvent` method include:

```
ctx.getCorrelator()->sendEvent("SimpleCounter(1)");
```

This will dispatch the event of type SimpleCounter with a single integer field set to 1.

Also, in the sample discussed above it is used thus:

```
ctx.getCorrelator()->sendEvent(args[0].stringValue());
```

Here the event provided by the first argument is the complete event to be dispatched.

There is one area where extra caution is required when building asynchronous plug-ins, which is the lifetime of variables within the plug-in. When a plug-in function is called with an `AP_Context` argument, that context is valid only for the duration of the call (and only on that thread). However, the `AP_CorrelatorInterface` remains valid for the lifetime of the plug-in. References to it may be retained and used at any time on any thread. This information is important to anyone writing a plug-in that may be holding references to an `AP_CorrelatorInterface`, for example, in another thread. The plug-in author must ensure that when the plug-in is shutdown these references are cleaned up, since attempts to use these references after the plug-in has been shutdown may cause instability of the event correlator.

Advanced Plug-in Functionality in C++ and C

Writing correlator plug-ins for parallel processing applications

For a plug-in created before Apama 5.2 to work with Apama 5.2, you must re-compile it.

Beginning with release 5.0, all plug-ins are required to be thread-safe. Beginning with release 4.2, the interface is more accurate with respect to the use of `const`, so minor code changes may be required.

Plug-ins created before Apama 4.2 run in a single operating system thread at a time. The correlator assumes that such plug-ins are not thread-safe. For each call to such a plug-in, the correlator acquires a mutex to ensure that multiple correlator contexts cannot use the plug-in at the same time.

When multiple contexts need to concurrently use a plug-in you must ensure that the plug-in is thread safe. A plug-in can export a function that returns the capabilities of the plug-in. See the `AP_PluginCommon.h` header file for the definition of `AP_PLUGIN_GET_CAPABILITIES_FUNCTION_NAME`. The correlator calls this function before it calls the plug-in's `init()` function. The return value is a bit-wise OR of capabilities, as defined in the `AP_PluginCommon.h` header file. If the return value indicates that the plug-in is thread-safe, multiple contexts can make concurrent calls to the plug-in. When multiple contexts need to concurrently use a plug-in, you must ensure that the plug-in is thread-safe.

A plug-in can use a context's ID to send events to a particular context. Use the `AP_Context.getContextId()` method to obtain the context ID. The correlator passes an `AP_Context` object to each plug-in. This object has a `getCorrelator()` method that returns an interface that defines a `sendEventTo()` method, which has the following signature:

```
sendEventTo(const char *event, AP_uint64 targetContextId, const AP_Context &source)
```

The `sendEventTo()` method takes three arguments:

- `event` — For the event to send, specify a string in the format described in .
- `targetContextId` — Specify the ID of the context you want to send the event to.
- `&source` — Specify the context that this plug-in call is running in. This is the `AP_Context` object that was passed to the plug-in method or event handler method. If this method is called from a background thread then that thread passes an `AP_Context::NoContext()` object to this method. Specify that object as the source context.

  You can obtain the current context ID with a call to `AP_Context.getContextId()`, which might be useful for sending or passing events to other threads. However, you should not use the returned object as the value for the `&source` argument.

The following overloading of the `sendEventTo()` method is deprecated and will be removed in a future release. Use the previously described overloading instead.

```
sendEventTo(const char *event, AP_uint64 targetContextId, AP_uint64 sourceContextId)
```

Note: The class `AP_Context`, which you use for correlator plug-in development, is completely different and unrelated to contexts that you define in EPL for parallel processing.

Advanced Plug-in Functionality in C++ and C

Working with blocking behavior in C++ plug-ins

When the behavior of a C++ plug-in is that it never blocks or does not usually block you can declare the plug-in to be nonblocking. Even if one or more methods defined in a plug-in might block, you can declare the plug-in to be nonblocking and override the nonblocking designation for just the methods that might block. The benefit of declaring a plug-in to be nonblocking is that the correlator refrains from creating unneeded processing threads.

By default, the correlator assumes that any plug-in method or handler it calls might block for an arbitrary amount of time. Consequently, the correlator creates additional threads to continue processing other contexts. If the plug-in method/handler does not block, these extra threads represent an expense that could be avoided.

Declaring a plug-in as nonblocking

To declare a plug-in as nonblocking, define an `AP_PLUGIN_GET_CAPABILITIES_FUNCTION_NAME` function that returns `AP_CAPABILITIES_NON_BLOCKING`. See the `AP_PluginCommon.h` header file in the `include` directory of your Apama installation directory for the definition of `AP_PLUGIN_GET_CAPABILITIES_FUNCTION_NAME`. The correlator calls this function before it calls the plug-in's `init()` function. The return value is an `AP_Capabilities` object that contains a bit-wise `OR` of capabilities, as defined in the `AP_PluginCommon.h` header file.

When you declare a plug-in to be nonblocking the correlator lets plug-in methods and any plug-in event handlers process to completion without spawning new threads.

You must ensure that a plug-in declared to be nonblocking does not block. If a nonblocking plug-in does block it can cause a correlator deadlock. To avoid this, for each plug-in method that might block, be sure to override the nonblocking designation. For example, consider a method that accesses a local cache. Normally, this method would not block. However, if the method uses a remote process when the needed object is not in the local cache then the method might block. You must override the nonblocking designation for a method such as this one.

Overriding the nonblocking designation for particular methods

A plug-in that you declare as nonblocking can have one or more methods that might block. In each method that might block, you must call the `pluginMethodBlocking()` function on either an `AP_Context` object or an `AP_CorrelatorInterface` object. The signature for the `pluginMethodBlocking()` function is as follows for each type of object:

```
void pluginMethodBlocking();
```

Calling `pluginMethodBlocking()` is idempotent. A call to `pluginMethodBlocking()` informs the correlator that the containing method might block and that the correlator should start additional threads to compensate.

Advanced Plug-in Functionality in C++ and C

Working with channels in C++ plug-ins

In a C++ correlator plug-in, you can send an event to a particular channel, subscribe to receive events sent to particular channels, receive events sent on subscribed channels, and unsubscribe from subscribed channels.

There is currently no support for channels in correlator plug-ins written in C.

Sending events to particular channels

To send an event to a particular channel, call the `AP_CorrelatorInterface.sendEventTo()` method:

```
virtual void sendEventTo(const char *event, const char *targetChannel, const AP_Context &source)
```

- `event` — For the event to send, specify a string in the format described in .
- `targetChannel` — Specify the name of the channel you want to send the event to.
- `&source` — Specify the context that this plug-in call is running in. This is the `AP_Context` object that was passed to the plug-in method or event handler method. If this method is called from a background thread then that thread passes an `AP_Context::NoContext()` object to this method. Specify that object as the source context.

An event that is passed to the `sendEventTo()` method is delivered to any contexts, receivers, and plug-in event handlers that are subscribed to the specified channel.

Defining an event handler class for receiving events

To receive events sent to channels, derive an event handler class from `AP_EventHandlerInterface` and implement the `handleEvent()` method:

```
virtual void handleEvent(const AP_BlockingAwareContext &ctx, const char *event, const char *channel)
```

`ctx` — Context in which this execution of the event handler is happening.

`event` — An event being received. The event must be represented as a string in the format described in .

`channel` — The channel on which the event was received.

Store each reference to an event handler instance in the `AP_EventHandlerInterface::ptr_t` smart pointer. When the last reference to a particular event handler is dropped then that instance is deleted.

Subscribing event handlers to channels

After you create an event handler class, you use event handler objects to subscribe to receive events sent on one or more channels. Each event handler object can receive events from multiple channels and you can specify the same event handler in multiple subscriptions. If you subscribe to receive events from the same channel more than once the duplicate subscriptions are ignored. When an event handler is subscribed to one or more channels its `handleEvent()` method is called once for each event that is sent to any subscribed channel.

There are several overloadings of the `AP_CorrelatorInterface.subscribe()` method:

- To use an initializer list to subscribe an event handler object to one or more channels:

  ```
  void subscribe(const AP_EventHandlerInterface::ptr_t &handler, std::initializer_list<const char *> channels);
  ```

This overloading is not supported on SUSE Linux Enterprise Server 11.

This overloading uses smart pointers for reference counting. Use the following format to call it:

```
correlator->subscribe(AP_EventHandlerInterface::ptr_t(new MyHandlerType()), { "channel one", "channel two" });
```

`handler` — Specify the handler to subscribe.

`channels` — Specify one or more channels that you want to receive events from.

- To use an iterator pair or an array of `char*` values to subscribe an event handler object to one or more channels:

```
template<typename ITER>
void subscribe(const AP_EventHandlerInterface::ptr_t &handler, const ITER &start, const ITER &end);
```

This overloading uses smart pointers for reference counting. Use the following format to call it:

```
correlator->subscribe(AP_EventHandlerInterface::ptr_t(new
MyHandlerType()), channels.begin(), channels.end());
```

`handler` — Specify the handler to subscribe.

`start` — The iterator to start from.

`end` — The iterator to end at.

The iterators must resolve to values that can be cast to `const char*` values. Alternatively, you can use an array of `char*` values in place of the iterators.

- To subscribe an event handler object to a single channel:

```
template<typename T>
void unsubscribe(const AP_EventHandlerInterface::ptr_t &handler, const T &channel);
```

This overloading uses smart pointers for reference counting. Use the following format to call this method:

```
correlator->subscribe(AP_EventHandlerInterface::ptr_t(new
MyHandlerType()), "channel one");
```

`handler` — Specify the handler to subscribe.

`channel` — Specify the channel to subscribe to. The value you specify must be a value that can be cast to `char*`.

Unsubscribing event handlers from channels

Several overloadings of the `AP_CorrelatorInterface.unsubscribe()` method let you cancel one, multiple, or all channel subscriptions. If the result of an `unsubscribe()` method is that the event handler has no subscriptions, and if there are no references to that event handler, then the event handler object is deleted.

- To use an initializer list to unsubscribe an event handler object from one or more channels:

```
void unsubscribe(const AP_EventHandlerInterface::ptr_t &handler, std::initializer_list<const char *> channels);
```

This overloading is not supported on SUSE Linux Enterprise Server 11.

This overloading uses smart pointers for reference counting. Use the following format to call it:

```
correlator->unsubscribe(my_handler, { "channel one", "channel two" });
```

`handler` — Specify the handler to unsubscribe.

`channels` — Specify a list of channels for which to cancel subscriptions.

- To use an iterator pair or an array of `char*` values to unsubscribe an event handler object from one or more channels:

  ```
  template<typename ITER>
  void unsubscribe(const AP_EventHandlerInterface::ptr_t &handler, const ITER &start, const ITER &end);
  ```

  This overloading uses smart pointers for reference counting. Use the following format to it:

  ```
  correlator->unsubscribe(my_handler, channels.begin(), channels.end());
  ```

  `handler` — Specify the handler to unsubscribe.

  `start` — The iterator to start from.

  `end` — The iterator to end at.

  The iterators must resolve to values that can be cast to `const char*` values. Alternatively, you can use an array of `char*` values in place of the iterators.

- To unsubscribe an event handler object from a single channel:

  ```
  template<typename T>
  void unsubscribe(const AP_EventHandlerInterface::ptr_t &handler, const T &channel);
  ```

  `handler` — Specify the handler to unsubscribe.

  `channels` — Specify the channel to unsubscribe from. The value you specify must be a value that can be cast to `char*`.

- To unsubscribe an event handler object from all channels it is subscribed to:

  ```
  virtual void unsubscribe(const AP_EventHandlerInterface::ptr_t &handler);
  ```

  `handler` — Specify the handler to unsubscribe. If there are no other references to this event handler, it is deleted.

Notes for writing C++ plug-ins that use channels

- Ordering

  When an event is sent to some contexts and some plug-ins the order the order in which those contexts and plug-ins process the event is unpredictable.

  Events sent on a particular channel maintain their order on the event handler that receives them. However, there is no ordering with regard to other components that might be subscribed to the same channel and so receive and operate on the same events.

  There is no ordering of events sent on different channels and received by the same event handler.

- Blocking

  As with plug-in method calls, methods on event handlers may be blocking or nonblocking. If a plug-in is declared as nonblocking then the correlator will assume that all its event handlers are also nonblocking. You can call the `AP_BlockingAwareContext.pluginMethodBlocking()` method to declare that an event handler is actually blocking, despite the overall plug-in nonblocking setting. Event handlers must not perform any potentially blocking operations if the plug-in is nonblocking without calling `pluginMethodBlocking()`. See "Working with blocking behavior in C++ plug-ins" on page 33.

- Exceptions

  If a handler throws an exception it is reported in the correlator log file and then discarded.

- Plug-in lifetime

If all monitors that reference a plug-in have terminated or have been removed by the `engine_delete` utility, then the plug-in and any event handlers that belong to the plug-in are removed from the correlator. If an event handler callback is in progress, then the delete operation blocks until the event handler has completed. At that point, references to the handler are dropped so that the plug-in can be unloaded.

C++ plug-in samples that use channels

C++ code samples that use channels in plug-ins are in the `subscribe_plugin` file in the *`apama_install_dir`*`\samples\correlator_plugin\cpp` folder.

Advanced Plug-in Functionality in C++ and C

Chapter 4: The EPL Plug-in APIs for C and C++

For reference, this topic provides `correlator_plugin.hpp`, the header file that provides the functionality of the EPL Plug-in C++ API. The file is located in the `include` folder of the Apama installation.

The equivalent header file for the EPL Plug-in C API is `correlator_plugin.h`. This will not explicitly be covered here, as it is broadly identical in functionality to the C++ header file. The only difference is that the functionality presented as class methods in the C++ API is presented as C functions in the C API. This can also be located in the `include` folder.

The C++ header defines several types. First it defines two enumerations:

- `AP_TypeDiscriminator` - Identifies the type of the data item encapsulated by an `AP_Type` "smart union" object. Its values map to EPL types.
- `AP_ErrorCode` – Specifies the error codes that can be returned by plug-in functions that do not throw exceptions. In this release of the API these are just the C-linkage initialization and destructor functions.

Then it defines a number of exceptions. All exception classes inherit from `AP_PluginException`, and they are `AP_TypeException`, `AP_UnimplementedException`, `AP_BoundsException` and `AP_SerialisationException`.

Primary class types

The primary class types follow:

- `AP_Chunk` – This is the base class for all `chunk` values. Plug-ins need to inherit from this class and add suitable data and function members in the derived class to manage their private data structures in memory allocated by the plug-in itself. `AP_Chunk` instances are passed in and out of plug-in functions as the "chunk value" of `AP_Type` objects, and referenced in EPL code via variables of type `chunk`.
- `AP_Type` – This is a type-safe encapsulation of an EPL object for passing arguments and return values into and out of plug-in functions. The implementation of the `AP_Type` member functions is internal to the event correlator. One consequence of this is that plug-ins *cannot* create a useful instance of this class themselves; the only valid `AP_Type` objects are those passed to a plug-in function by the event correlator. `AP_Type` is a "smart union" object; each instance holds a single value of one of the supported types and only allows access to data of that type. Note that `integer`, `float`, and `boolean` values are passed by value, while the "complex" types — sequence and chunk — are passed by reference, so changes made to the contents of these objects by a plug-in will be seen by the invoking EPL code. Strings are treated slightly differently: though EPL string objects themselves are immutable, the plug-in API allows return values and the values in a sequence of strings to be modified. When this is done, a new EPL string object is created containing the specified text. As of version 5.0 it is no longer possible to modify a string argument to a plug-in function.

- `AP_TypeList` – A container class for an ordered list of `AP_Type` objects, typically used to hold the argument list for a plug-in function call.
- `AP_Context` – The execution context for a plug-in function call. Holds per-call correlator-internal data and provides various utility functions to plug-ins. Note that the implementation of the `AP_Context` member functions is internal to the event correlator. One consequence of this is that plug-ins *cannot* create a useful instance of this class themselves; the only valid `AP_Context` objects are those passed in to a plug-in function by the event correlator.
- `AP_Function` – A plug-in function descriptor. The argument and return types in this structure are strings (not `AP_TypeDiscriminator` objects) that use the same syntax as EPL declarations. For example, if one declares a function argument as a `sequence` of `integer`s, the corresponding element of the `paramTypes` array would contain `sequence<integer>`.
- `AP_CorrelatorInterface` – An abstraction of the interface for calling back into the event correlator. There is a single method provided that enables a plug-in to send events to the event correlator. An instance of this class is acquired by requesting it via a method on the `AP_Context`.

The header file also defines pointers to the plug-in initialization and destructor (or shutdown) functions, as well as version checking. Each plug-in must export these two functions with these signatures, named using specific macros and with "C" linkage. The first is called immediately upon loading of the plug-in by the library whereas the other is called immediately before unloading.

If the plug-in's functions can safely be called simultaneously from multiple EPL contexts, a get-capabilities function should be defined that announces the plug-in as thread-safe.

Some plug-ins may need to keep thread-specific data in order to work correctly; in which case a thread-ended function should also be defined. This function will be called on the thread so that resources can be freed if the thread is ending before the plug-in is shut down. When the shutdown function is called it is responsible for freeing resources related to any threads that are still running.

The EPL Plug-in APIs for C and C++

Listing of correlator_plugin.hpp

A listing of `correlator_plugin.hpp` is given here. The file is extensively documented and is recommended as a reference to the functionality available within the definitions of the classes listed above.

```
/**
 * correlator_plugin.hpp
 *
 * Apama Plugin C++ API definitions.  Used by plugin libraries and the plugin
 * support code within the Engine itself.
 *
 * $Copyright(c) 2014 Software AG, Darmstadt, Germany and/or its licensors$
 *
 * $Id$
 */

#ifndef CORRELATOR_PLUGIN_HPP
#define CORRELATOR_PLUGIN_HPP

#include <ApamaTypes.h>
#include <AP_Platform.h>
#include <AP_PluginCommon.h>
#include <exception>
#include <stdexcept>
#include <string>
```

```
#include <vector>
#include <iostream>

/**
 * Plugin API version.  The upper 16 bits identify the major release of the
 * API; the lower 16 bits indicate minor versions within that release.  No
 * changes that break backward binary compatibility will be made within a
 * given major release of the API.  That is, minor releases may add
 * functionality and fix bugs, but will not change or remove any documented
 * behaviour.  AP_PluginCommon.h provides two _VERSION_MASK symbols for
 * convenience in performing fine-grained version checks.
 *
 * Version history:
 * 1.0  First version released in Apama Engine 2.0.3
 *      Win32 support added in Apama Engine 2.0.4
 * 2.0  Add serialisation support for Apama Engine 2.1 release
 *      Unfortunately this breaks back compatibility of chunks
 * 3.0  Added getCorrelator, for internal use only
 * 4.0  Make the result of the getCorrelator() context method into a real
 *      public class - AP_CorrelatorInterface.  Unfortunately v3.0 plugins
 *      will be expecting a different type to be returned by this method,
 *      so a major version bump is required.  However, this does bring the
 *      C++ and C plugin APIs back into sync - future changes can be made
 *      to both APIs and the version numbers incremented together.
 * 5.0  Add getContextId and sendEventTo. Also added the optional
 *      GET_CAPABILITIES and THREAD_ENDED functions that the plugin may
 *      provide. Improved some function prototypes.
 * 6.0  Improved the interface's const-correctness. Increased the
 *      performance of AP_TypeList, made AP_Chunk construction and
 *      destruction cheaper, especially in a multi-threaded
 *      environment. Added AP_Type::visitSequenceElements.
 * 7.0  Remove serialisation support
 *      Prohibit mutation of string arguments to plug-in functions
 *      Improve validation of returned values
 */
#define AP_PLUGIN_VERSION             0x00070000

// suse doesn't have std::initializer_list because it's still on gcc 4.3.
// Snip out those functions on suse
#if (__GNUC__ > 4 || __GNUC_MINOR__ > 3 || !defined(__linux__))
#define _has_std_initializer_list
#endif
/**
 * Base class of all exceptions throw across the correlator/plugin boundary.
 * Unless documented otherwise, plugins should assume that any API function can
 * throw this exception.
 */
class AP_PluginException : public std::runtime_error {
 public:
  explicit AP_PluginException(const std::string& what) : std::runtime_error(what) { }
};

/**
 * Type error.  Typically results from an attempt to read/write a value of the
 * wrong type from/to an AP_Type object.
 */
class AP_TypeException : public AP_PluginException {
 public:
  explicit AP_TypeException(const std::string& what) : AP_PluginException(what) { }
};

/**
 * Thrown by an attempt to use a function that is defined but not implemented in
 * this version of the API.  Can also be thrown by deprecated functions that
 * are still in the API but for which no reasonable implementation exists.
 */
class AP_UnimplementedException : public AP_PluginException {
 public:
  explicit AP_UnimplementedException(const std::string& what) : AP_PluginException(what) { }
};
```

```
/**
 * Boundary checking error.  Thrown by attempted access to non-existent
 * sequence or array elements.
 */
class AP_BoundsException : public AP_PluginException {
 public:
  explicit AP_BoundsException(const std::string& what) :
     AP_PluginException(what) { }
};

/**
 * Thrown by plugin authors to signal an error to any EPL calling this plugin.
 * While all exceptions thrown from a plugin method can be caught within EPL,
 * this class is preferred as it allows the plugin author to determine the
 * type of the EPL exception.
 */
class AP_UserPluginException : public AP_PluginException {
 public:
  /**
   * @param type Calling getType() on the resulting EPL exception will
   * result in "PluginException:<type>"
   * @param what Calling getMessage() on the resulting EPL exception will
   * return this string
   */
  explicit AP_UserPluginException(const std::string type,
     const std::string& what) : AP_PluginException(what), type(type) {}
  const std::string& getType() const {
   return type;
  }
  ~AP_UserPluginException() throw() {}
 private:
  std::string type;
};
/** Forward declarations */
class AP_Chunk;
class AP_CorrelatorInterface;
class AP_BlockingAwareContext;
/**
 * Execution context for a library function call.
 *
 * Note that the plugin 'execution context' is not the same as the
 * concept of 'contexts' in EPL, which are used to provide parallelism,
 * despite the name being the same. However each AP_Context plugin execution
 * context provides a getContextId() method for getting the EPL context from
 * which the plugin is being invoked.
 *
 * AP_Context holds per-call Engine-internal data and provides various
 * utility functions to plugins.
 *
 * Note that the implementation of the AP_Context member functions is internal
 * to the Engine.  One consequence of this is that plugins *cannot* create a
 * useful instance of this class themselves; the only valid AP_Context objects
 * are those passed in to a plugin function by the Engine.
 */
class AP_Context {

 public:
  /** Destructor */
  virtual ~AP_Context() = 0;

  /** Return active plugin API version */
  virtual uint32 version() const = 0;

  /**
   * Allocate storage for an 8-bit string value.  The memory will be
   * allocated by the Engine and must only be freed using the
   * char8free() function.  These functions should be used for any
   * string data that may need to be copied or otherwise manipulated by
   * the Engine; in particular, strings within chunk objects that the
```

```
   * Engine may be required to (de)serialise.
   *
   * @param len The length of the string buffer.  Must be sufficient to
   * cover any terminators or other special characters to be included in
   * the string.
   *
   * @return Pointer to a memory block of at least len bytes.
   */
  virtual char8* char8alloc(size_t len) const = 0;

  /**
   * Duplicate a string value created using char8alloc().  This function
   * allocates a memory block of sufficient size and copies the contents
   * of the source string into it.  The returned string must only be
   * freed using char8free().
   *
   * @param s The source string to be copied.  Must have been created
   * with char8alloc().
   *
   * @return Pointer to a copy of the source string.
   */
  virtual char8* char8dup(const char8* s) const = 0;

  /**
   * Free the memory occupied by a string created using char8alloc() or
   * char8free().  The source string pointer is invalid after this
   * function returns.  This function must not be used to free any
   * object that was not allocated by char8alloc() or char8free().
   *
   * @param s The source string to be freed.
   */
  virtual void char8free(char8* s) const = 0;

  /**
   * Return the correlator implementation.
   */
  virtual AP_CorrelatorInterface* getCorrelator() const = 0;
  /**
   * Return the identifier of the EPL (parallelism) context associated
   * with this plugin execution context.
   */
  virtual AP_uint64 getContextId() const = 0;
};

/**
 * An interface class that should be implemented by event handlers which are
 * registered via AP_CorrelatorInterface::subscribe.
 */
class AP_EventHandlerInterface
{
public:
 /**
  * A smart-pointer type for managing lifetime of event handlers.
  *
  * All event handlers must be stored in an AP_EventHandlerInterface::ptr_t.
  * A reference is held by the correlator for each subscribed event handler.
  * When the last reference is dropped, the event handler will be deleted.
  */
 typedef std::shared_ptr<AP_EventHandlerInterface> ptr_t;
 /**
  * Ensure we have a virtual destructor, so that derived class destructors
  * can be correctly called.
  */
 virtual ~AP_EventHandlerInterface() {}
 /**
  * Called for each event sent to a channel to which this event handler
  * is subscribed.
  *
  * @param ctx Execution context for this invocation of the event handler.
  * @param event the event received.  The event is represented as a string
```

```
  * using the format described in the manual.
  * @param channel The channel on which the event was received.
  */
 virtual void handleEvent(const AP_BlockingAwareContext &ctx,
    const char *event, const char *channel) = 0;
};

/**
 * Asynchronous interface to the correlator.  A per-plugin instance of this
 * class can be obtained by calling AP_Context::getCorrelator().  Methods
 * of this class may be invoked from background threads.
 */
class AP_CorrelatorInterface {

public:
 /** Destructor */
 virtual ~AP_CorrelatorInterface() = 0;

 /**
  * Send an event to the correlator
  *
  * @param event the event to send.  The event is represented as a string
  * using the format described in the manual.
  * See AP_EventWriter.h for utility code to help build valid event strings.
  */
 virtual void sendEvent(const char* event) = 0;

 /**
  * Send an event to a specific context in the correlator
  *
  * @param event the event to send.  The event is represented as a string
  * using the format described in the manual.
  * See AP_EventWriter.h for utility code to help build valid event strings.
  * @param targetContext the target context.
  * @param source the source context. Must be as returned from
  * AP_Context.getContextId(), or 0 if called from a plugin thread.
  */
  virtual void sendEventTo(const char *event, AP_uint64 targetContext,
                const AP_Context &source) = 0;

 /**
  * Send an event to a specific channel in/out of the correlator
  *
  * @param event the event to send.  The event is represented as a string
  * using the format described in the manual.
  * See AP_EventWriter.h for utility code to help build valid event strings.
  * @param targetChannel the target channel name.
  * @param sourceContext the source context. Must be as returned from
  * AP_Context.getContextId(), or 0 if called from a plugin thread.
  */
 virtual void sendEventTo(const char *event, const char *targetChannel,
    const AP_Context &source) = 0;
#ifdef _has_std_initializer_list
 /**
  * Subscribe an event handler to a list of channels.
  *
  * If this event handler is already subscribed to one or more channels,
  * then the channels you specify in this call are added to that subscription
  * quashing duplicates). Otherwise, a new subscription is created for the
  * channels you specify.
  *
  * This method uses smart pointers for reference counting and an initializer
  * list (C++11) so should be called like this:
  *
  * correlator->subscribe(AP_EventHandlerInterface::ptr_t(new MyHandlerType()),
  *   { "channel one", "channel two" });
  *
  * @param handler The event handler to subscribe.
  * @param channels A list of the channels to subscribe to.
  */
```

```
 void subscribe(const AP_EventHandlerInterface::ptr_t &handler,
    std::initializer_list<const char *> channels)
 {
  subscribe_impl(handler, channels.begin(), channels.end());
 }
#endif
 /**
  * Subscribe an event handler to a list of channels.
  *
  * If this event handler is already subscribed to one or more channels,
  * then the channels you specify in this call are added to that subscription
  * quashing duplicates). Otherwise, a new subscription is created for the
  * channels you specify.
  *
  * This method uses smart pointers for reference counting and an iterator pair,
  * so should be called like this:
  *
  * correlator->subscribe(AP_EventHandlerInterface::ptr_t(new MyHandlerType()),
  *   channels.begin(), channels.end());
  *
  * or (with an array of char*s):
  *
  * correlator->subscribe(AP_EventHandlerInterface::ptr_t(new MyHandlerType()),
  *    channels+0, channels+n);
  *
  * equivalent to:
  * for (auto it = start; it != end; ++it) {
  *  subscribe(my_handle, *it);
  * }
  *
  * The iterators must dereference to something that can be cast to a const char*
  *
  * @param handler The event handler to subscribe.
  * @param start The iterator to start from
  * @param end The iterator to end at
  */
 template<typename ITER>
 void subscribe(const AP_EventHandlerInterface::ptr_t &handler,
    const ITER &start, const ITER &end)
 {
  subscribe_impl(handler, start, end);
 }
 /**
  * Subscribe an event handler to a list of channels.
  *
  * If this event handler is already subscribed to one or more channels,
  * then the channels you specify in this call are added to that subscription
  * quashing duplicates). Otherwise, a new subscription is created for the
  * channels you specify.
  *
  * This method uses smart pointers for reference counting so should be called
  * like this:
  *
  * correlator->subscribe(AP_EventHandlerInterface::ptr_t(new MyHandlerType()),
  *    "channel one");
  *
  * @param handler The event handler to subscribe.
  * @param channel The channel to subscribe to (must be castable to char*).
  */
 template<typename T>
 void subscribe(const AP_EventHandlerInterface::ptr_t &handler, const T &channel)
 {
  subscribe_impl(handler, &channel, (&channel)+1);
 }
#ifdef _has_std_initializer_list
 /**
  * Unsubscribe an event handler from a list of channels.
  *
  * If this results in the handler being subscribed to no channels, then removes
  * that subscription entirely (which may
```

```
   * result in deleting the handler if it has no other references).
   *
   * This method uses smart pointers for reference counting and an initializer
   * list (C++11) so should be called like this:
   *
   * correlator->unsubscribe(my_handler, { "channel one", "channel two" });
   *
   * @param handler The event handler to unsubscribe.
   * @param channels A list of the channels to unsubscribe from.
   */
  void unsubscribe(const AP_EventHandlerInterface::ptr_t &handler,
     std::initializer_list<const char *> channels)
  {
   unsubscribe_impl(handler, channels.begin(), channels.end());
  }
#endif
  /**
   * Unsubscribe an event handler from a list of channels.
   *
   * If this results in the handler being subscribed to no channels, then removes
   * that subscription entirely (which may
   * result in deleting the handler if it has no other references).
   *
   * This method uses smart pointers for reference counting and an iterator pair
   * so should be called like this:
   *
   * correlator->unsubscribe(my_handler, channels.begin(), channels.end());
   *
   * or (with an array of char*s):
   *
   * correlator->unsubscribe(my_handler, channels+0, channels+n);
   *
   * equivalent to:
   * for (auto it = start; it != end; ++it) {
   *  unsubscribe(my_handle, *it);
   * }
   *
   * The iterators must dereference to something that can be cast to a const char*
   *
   * @param handler The event handler to unsubscribe.
   * @param start The iterator to start from
   * @param end The iterator to end at
   */
  template<typename ITER>
  void unsubscribe(const AP_EventHandlerInterface::ptr_t &handler,
     const ITER &start, const ITER &end)
  {
   unsubscribe_impl(handler, start, end);
  }
  /**
   * Unsubscribe an event handler from a channel.
   *
   * If this results in the handler being subscribed to no channels,
   * then removes that subscription entirely (which may
   * result in deleting the handler if it has no other references).
   *
   * This method uses smart pointers for reference counting.
   *
   * @param handler The event handler to unsubscribe.
   * @param channel The channel to unsubscribe from (must be castable to a char*)
   */
  template<typename T>
  void unsubscribe(const AP_EventHandlerInterface::ptr_t &handler,
     const T &channel)
  {
   unsubscribe_impl(handler, &channel, (&channel)+1);
  }
  /**
   * Unsubscribe an event handler from all channels.
   *
```

```
  * This may result in deleting the handler if it has no other references.
  *
  * @param handler The event handler to unsubscribe.
  */
 virtual void unsubscribe(const AP_EventHandlerInterface::ptr_t &handler) = 0;
protected:
 virtual void subscribe_impl(const AP_EventHandlerInterface::ptr_t &handler,
    char const *const *start, char const *const *end) = 0;
 virtual void unsubscribe_impl(const AP_EventHandlerInterface::ptr_t &handler,
    char const *const *start, char const *const *end) = 0;
 template<typename ITER>
 void subscribe_impl(const AP_EventHandlerInterface::ptr_t &handler,
    const ITER &start, const ITER &end)
 {
  for (ITER it = start; it != end; ++it) {
   subscribe_impl(handler, (char const *const *) &(*it),
      (char const *const *) &(*it) + 1);
  }
 }
 template<typename ITER>
 void unsubscribe_impl(const AP_EventHandlerInterface::ptr_t &handler,
    const ITER &start, const ITER &end)
 {
  for (ITER it = start; it != end; ++it) {
   unsubscribe_impl(handler, (char const *const *) &(*it),
   (char const *const *) &(*it) + 1);
  }
 }
};
/**
 * Base class for all 'chunk' classes.  Plugins should inherit from this class
 * and add suitable data and function members in the derived class to manage
 * their private data structures, in memory allocated by the plugin itself.
 * AP_Chunk instances are passed in and out of plugin functions as the 'chunk
 * value' of AP_Type objects, and referenced in MontitorScript code as
 * variables of type 'chunk'.  Note that the contents of a chunk are entirely
 * opaque to MonitorScript.
 */
class AP_PLUGIN_API AP_Chunk {

 public:
  /**
   * Pure virtual destructor.  It is *essential* that every AP_Chunk
   * derived class implements this method to free any resources
   * allocated by the derived class.  The Engine will arrange for the
   * destructor to be called when the associated MonitorScript chunk
   * object is deleted.
   *
   * The correlator interface may not be used from a chunk's destructor.
   *
   * Also, correlator interface calls on another thread may
   * block until the chunk's destructor returns. Chunk
   * destructors that can block should therefore be careful to
   * avoid deadlocking against such a thread.
   */
  virtual ~AP_Chunk() {}

  /**
   * Chunk cloning method (typically calls a copy constructor in the
   * derived class).  As with the destructor, it is essential for
   * AP_Chunk derived classes to implement this method, to ensure that
   * MonitorScript assignments to/from chunk objects work correctly.
   *
   * @param ctx Execution context for the copy operation.
   *
   * @return Pointer to a copy of this AP_Chunk object.
   */
  virtual AP_Chunk* copy(const AP_Context& ctx) const = 0;

  /**
```

```
   * Obsolete constructor retained for backward compatability
   */
  explicit AP_Chunk(const AP_Context &) {}

  /**
   * Default constructor declared because there's a non-default one
   */
  AP_Chunk() {}
};


/**
 * Type-safe encapsulation of a MonitorScript object, for passing arguments
 * and return values into and out of plugin functions.  Note that the
 * implementation of the AP_Type member functions is internal to the Engine.
 * One consequence of this is that plugins *cannot* create a useful instance
 * of this class themselves; the only valid AP_Type objects are those passed
 * in to a plugin function by the Engine.
 *
 * AP_Type is a "smart union" object -- each instance holds a single value of
 * one of the supported types and only allows access to data of that type.
 */
class AP_Type {

 public:
  /** Destructor */
  virtual ~AP_Type() = 0;

  /** Get the type of the encapsulated object */
  virtual AP_TypeDiscriminator discriminator() const = 0;

  /** Get the chunk value of the object */
  virtual AP_Chunk* chunkValue() const = 0;

  /**
   * Set the chunk value of the object.
   *
   * The correlator takes ownership of the new AP_Chunk; it should
   * not be deleted by the plugin.
   *
   * The correlator releases ownership of the old AP_Chunk and
   * returns it to the plug-in for disposal.
   *
   * The old pointer, or new, or both, may be null.
   */
  virtual AP_Chunk *chunkValue(AP_Chunk* val) const = 0;

  /** Get the integer value of the object */
  virtual int64 integerValue() const = 0;

  /** Set the integer value of the object */
  virtual void integerValue(int64 val) = 0;

  /** Get the float value of the object */
  virtual float64 floatValue() const = 0;

  /** Set the float value of the object */
  virtual void floatValue(float64 val) = 0;

  /** Get the boolean value of the object */
  virtual bool booleanValue() const = 0;

  /** Set the boolean value of the object */
  virtual void booleanValue(bool val) = 0;

  /** Get the string value of the object */
  virtual const char8* stringValue() const = 0;

  /** Set the string value of the object.
   * Note that this is only possible on sequence elements and
```

```
 * return values, not arguments (which are const)
 */
virtual void stringValue(const char8* val) = 0;

/** Get the number of elements in a sequence object */
virtual size_t sequenceLength() const = 0;

/** Get the element type of a sequence object */
virtual AP_TypeDiscriminator sequenceType() const = 0;

/**
 * Get a reference to a single sequence element. The
 * element AP_Type remains usable until the sequence AP_Type
 * is destroyed.
 */
virtual AP_Type &sequenceElement(size_t index) const = 0;

/**
 * Visit the specified elements of a sequence using the
 * specified functor.
 *
 * Although less convenient than other sequence element
 * accessors, it is more efficient as it entails no memory
 * allocation.
 */
template <typename FN>
void visitSequenceElements(const FN &fn, size_t start = 0,
          size_t length = size_t(-1)) const {
 visitSequenceElementsImpl(Wrap<FN>(fn), start, length);
}

/**
 * Generate an array of pointers to AP_Type objects, encapsulating
 * elements [start..start+length-1] of the sequence object.  This may
 * or may not require copying the original MonitorScript objects.
 */
virtual AP_Type* const * sequenceElements(size_t start = 0,
         size_t length = size_t(-1)) const = 0;

/**
 * Free the resources allocated by *all* preceding calls to
 * sequenceElements() and ensures that any changes made to the
 * sequence elements are reflected in the underlying MonitorScript
 * objects.  Any arrays previously returned by sequenceElements()
 * immediately become invalid.  This function is automatically called
 * when a sequence AP_Type is destroyed, but can be invoked explicitly
 * by a plugin to copy changes back during a long-running computation.
 */
virtual void releaseSequenceElements() const = 0;

/**
 * Generate an array of pointers to chunks encapsulating elements
 * [start..start+length-1] of the sequence object.  This may or may
 * not require copying the original MonitorScript objects.
 *
 * Note that the AP_Chunk pointers cannot be modified via this
 * call. Use sequenceElement(), visitSequenceElements() or
 * sequenceElements() instead, being sure to free any AP_Chunk
 * instances that are replaced.
 */
virtual AP_Chunk* const * chunkSequenceElements(size_t start = 0,
         size_t length = size_t(-1)) const = 0;

/**
 * Free the resources allocated by *all* preceding calls to
 * chunkSequenceElements(). Arrays previously returned by
 * chunkSequenceElements() immediately become invalid. This
 * function is automatically called when a sequence AP_Type is
 * destroyed, but can be invoked explicitly by a plugin during
 * a long-running computation.
```

```
 */
virtual void releaseChunkSequenceElements() const = 0;

/**
 * Generate an array of integers encapsulating elements
 * [start..start+length-1] of the sequence object.  This may or may
 * not require copying the original MonitorScript objects.
 */
virtual int64* integerSequenceElements(size_t start = 0,
          size_t length = size_t(-1)) const = 0;

/**
 * Free the resources allocated by *all* preceding calls to
 * integerSequenceElements() and ensures that any changes made to the
 * sequence elements are reflected in the underlying MonitorScript
 * objects.  Arrays previously returned by integerSequenceElements()
 * immediately become invalid.  This function is automatically called
 * when a sequence AP_Type is destroyed, but can be invoked explicitly
 * by a plugin to copy changes back during a long-running computation.
 */
virtual void releaseIntegerSequenceElements() const = 0;

/**
 * Generate an array of floats encapsulating elements
 * [start..start+length-1] of the sequence object.  This may or may
 * not require copying the original MonitorScript objects.
 */
virtual float64* floatSequenceElements(size_t start = 0,
          size_t length = size_t(-1)) const = 0;

/**
 * Free the resources allocated by *all* preceding calls to
 * floatSequenceElements() and ensures that any changes made to the
 * sequence elements are reflected in the underlying MonitorScript
 * objects.  Arrays previously returned by floatSequenceElements()
 * immediately become invalid.  This function is automatically called
 * when a sequence AP_Type is destroyed, but can be invoked explicitly
 * by a plugin to copy changes back during a long-running computation.
 */
virtual void releaseFloatSequenceElements() const = 0;

/**
 * Generate an array of booleans encapsulating elements
 * [start..start+length-1] of the sequence object.  This may or may
 * not require copying the original MonitorScript objects.
 */
virtual bool* booleanSequenceElements(size_t start = 0,
          size_t length = size_t(-1)) const = 0;

/**
 * Free the resources allocated by *all* preceding calls to
 * booleanSequenceElements() and ensures that any changes made to the
 * sequence elements are reflected in the underlying MonitorScript
 * objects.  Arrays previously returned by booleanSequenceElements()
 * immediately become invalid.  This function is automatically called
 * when a sequence AP_Type is destroyed, but can be invoked explicitly
 * by a plugin to copy changes back during a long-running computation.
 */
virtual void releaseBooleanSequenceElements() const = 0;

/**
 * Generate an array of string pointers encapsulating elements
 * [start..start+length-1] of the sequence object.  This may or may
 * not require copying the original MonitorScript objects. Strings
 * pointed to by this sequence are only valid for the lifetime of a
 * call into the plugin.
 */
virtual const char8** stringSequenceElements(size_t start = 0,
          size_t length = size_t(-1)) const = 0;
```

```
/**
 * Free the resources allocated by *all* preceding calls to
 * stringSequenceElements() and copies any changes made to the
 * sequence elements to the underlying MonitorScript objects.
 * Arrays previously returned by stringSequenceElements()
 * immediately become invalid.  This function is automatically called
 * when a sequence AP_Type is destroyed, but can be invoked explicitly
 * by a plugin to copy changes back during a long-running computation.
 *
 * Note that string sequence element arrays do not own the
 * strings they reference and will never delete a string the
 * plugin has placed in the array - if the plugin has set a
 * value, it must delete that string (and it cannot use
 * the value in the array returned from stringSequenceElement
 * - plugins will need to keep another pointer to any new
 * strings they create.
 */
virtual void releaseStringSequenceElements() const = 0;

/** Operator to get chunk value from object */
operator AP_Chunk*() const {
 return chunkValue();
}

/** Operator to assign chunk value to object */
AP_Type &operator= (AP_Chunk* val) {
 chunkValue(val);
 return *this;
}

/** Operator to get integer value from object */
operator int64() const {
 return integerValue();
}

/** Operator to assign integer value to object */
AP_Type &operator= (int64 val) {
 integerValue(val);
 return *this;
}

/** Operator to get float value from object */
operator float64() const {
 return floatValue();
}

/** Operator to assign float value to object */
AP_Type &operator= (float64 val) {
 floatValue(val);
 return *this;
}

/** Operator to get boolean value from object */
operator bool() const {
 return booleanValue();
}

/** Operator to assign boolean value to object */
AP_Type &operator= (bool val) {
 booleanValue(val);
 return *this;
}

/** Operator to get string value from object */
operator const char8*() const {
 return stringValue();
}

/** Operator to assign string value to object */
AP_Type &operator= (const char8* val) {
```

```
   stringValue(val);
   return *this;
  }

  /** Operator to get sequence element of object */
  AP_Type& operator[] (size_t index) const {
   return sequenceElement(index);
  }

 protected:
  /** Polymorphic functor passed to visitSequenceElementImpl */
  class ElementFn {
  public:
   virtual ~ElementFn() { }
   virtual void operator()(AP_Type &) const =0;
  };

  /** Polymorphic implementation of the generic visitSequenceElements */
  virtual void visitSequenceElementsImpl(const ElementFn &,
            size_t start, size_t length) const =0;

 private:
  template <typename FN>
  class Wrap : public ElementFn {
  public:
   explicit Wrap(const FN &fn) : fn(fn) {}
   void operator()(AP_Type &arg) const { fn(arg); }
  private:
   const FN &fn;
  };
};


/**
 * Container class for an ordered list of AP_Type objects, typically used to
 * hold the argument list for a plugin function call.
 */
class AP_TypeList {
public:
 template <typename T> AP_TypeList(const T *array, size_t n)
  : ptr(reinterpret_cast<const char *>(static_cast<const AP_Type *>(array))),
    n(n), stride(sizeof(T))
 {}

 /** Return the number of objects in the list */
 size_t size() const { return n; }

 /** Return true iff size() == 0 */
 bool empty() const { return n==0; }

 /** Return a reference to an element of the list */
 const AP_Type &operator[] (size_t i) const {
  return *reinterpret_cast<const AP_Type *>(ptr + i*stride);
 }

private:
 const char *ptr;
 size_t n, stride;
};


/**
 * Pointer to a plugin function.  All functions exported by a plugin library
 * must follow this interface.
 *
 * @param ctx Execution context for this invocation of the function.
 * @param args Function parameters, passed by reference.
 * @param rval Function return value, to be filled in by plugin.
 *
 * @throw AP_PluginException If anything goes wrong.
```

```
 */
typedef void (AP_PLUGIN_CALL* AP_FunctionPtr)(
                const AP_Context& ctx, const AP_TypeList& args,
                AP_Type& rval, AP_TypeDiscriminator rtype);


/**
 * Plugin function descriptor.  Note that the argument and return types in
 * this structure are strings (not AP_TypeDiscriminator objects) that use the
 * same syntax as MonitorScript declarations.  For example, the declare a
 * function argument as a sequence of integers, the corresponding element of
 * the paramTypes array would contain "sequence<integer>".
 */
struct AP_Function {

 /** Function name */
 const char8* name;

 /** Pointer to function implementation */
 AP_FunctionPtr fptr;

 /** Argument count */
 size_t nParams;

 /** Argument types.  nParams elements, unterminated */
 const char8** paramTypes;

 /** Return type */
 const char8* returnType;
};


/**
 * Pointer to a plugin library initialisation function.  Each plugin must
 * export a single function with this signature, named using the
 * AP_INIT_FUNCTION_NAME macro, with "C" linkage.  The initialisation function
 * will be called immediately after the library is loaded by the Engine.
 *
 * @param ctx Execution context for the function call.
 *
 * @param version Active plugin API version.  The plugin should verify it is
 * compatible with this version (and return AP_VERSION_MISMATCH_ERROR if not)
 * and update version to indicate the API version the plugin was compiled
 * against.
 *
 * @param nFunctions The plugin should set this to the number of functions it
 * exports.
 *
 * @param functions The plugin should set this to the address of an array of
 * AP_Functions structures, of length nFunctions, describing the functions
 * exported by the plugin.  This data will be read and copied by the plugin
 * API.
 *
 * @return An AP_ErrorCode indicating the success or otherwise of the library
 * initialisation.
 */
typedef AP_PLUGIN_DLL_SYM AP_ErrorCode (
          AP_PLUGIN_CALL* AP_InitFunctionPtr)(const AP_Context& ctx, uint32&
          version, uint32& nFunctions, AP_Function*& functions);

/**
 * Pointer to a plugin library shutdown function.  Each plugin must export a
 * single function with this signature, named using the
 * AP_SHUTDOWN_FUNCTION_NAME macro, with "C" linkage.  The shutdown function
 * will be called immediately before the library is unloaded by the Engine.
 *
 * @param ctx Execution context for the function call.
 *
 * @return An AP_ErrorCode indicating the success or otherwise of the function
 * call.
```

```
 */
typedef AP_PLUGIN_DLL_SYM AP_ErrorCode (
          AP_PLUGIN_CALL* AP_ShutdownFunctionPtr)(const AP_Context& ctx);

/**
 * Pointer to a plugin library version function.  Each plugin must export a
 * single function with this signature, named using the
 * AP_LIBRARY_VERSION_FUNCTION_NAME macro, with "C" linkage.  The version
 * function is called *before* the library initialisation function to
 * determine the version of the plugin API supported by the library without
 * triggering any side-effects that the initialisation function might have.
 * If the function is not present, the plugin API will assume that the library
 * conforms to an earlier version of the API and proceed accordingly.
 *
 * @param ctx Execution context for the function call.
 *
 * @param version The plugin should fill this in with the version of the API
 * it was compiled against.
 *
 * @return An AP_ErrorCode indicating the success or otherwise of the function
 * call.
 */
typedef AP_PLUGIN_DLL_SYM AP_ErrorCode (
          AP_PLUGIN_CALL* AP_LibraryVersionFunctionPtr)(const AP_Context& ctx,
              uint32& version);

/**
 * Pointer to a plugin capability function.  Each plugin must export a
 * single function with this signature, named using the
 * AP_PLUGIN_GET_CAPABILITIES_FUNCTION_NAME macro, with "C" linkage.  The get capabilities
 * function is called after the version function and *before* the library
 * initialisation function to determine the capabilities/ features of the
 * plugin API supported by the library without
 * triggering any side-effects that the initialisation function might have.
 * If the function is not present, the plugin API will assume that the library
 * conforms to an earlier version of the API and proceed accordingly.
 *
 * @param ctx Execution context for the function call.
 *
 * @return An AP_Capabilities indicating the capabilities/ properties of the
 * plugin (multiple values are bitwise-OR'd together)
 */
typedef AP_PLUGIN_DLL_SYM AP_Capabilities (
          AP_PLUGIN_CALL* AP_GetCapabilitiesFunctionPtr)(const AP_Context& ctx);

/**
 * Pointer to a plugin library thread ended function.
 * The function is called by the correlator when a thread ends.
 * The function is optional, but not implementing it may lead to leaks
 * as the plugin will not be told if a  thread has ended.
 *
 *  This function pointer is not guaranteed to be called before AP_ShutdownFunctionPtr
 *  Any plugin using this will have to free resources in AP_ShutdownFunctionPtr as if
 *  AP_ThreadEndedFunctionPtr may have not been called.
 *
 * @param ctx Execution context for the function call.
 *
 * @return An AP_ErrorCode indicating the success or otherwise of the function
 * call.
 */
typedef AP_PLUGIN_DLL_SYM AP_ErrorCode (
          AP_PLUGIN_CALL* AP_ThreadEndedFunctionPtr) (const AP_Context& ctx);

#endif // CORRELATOR_PLUGIN_HPP
```

The EPL Plug-in APIs for C and C++

Chapter 5: Writing Correlator Plug-ins in Java

EPL plug-ins can also be written in Java. Java plug-ins are automatically analyzed by the correlator and any suitable methods exposed as methods that can be called from EPL.

Creating a plug-in using Java

To create a Java class to use as a correlator plug-in:

1. In the Java class used as a plug-in, you need to have one or more public static methods that match the permitted signatures, which are described in "Permitted signatures for methods" on page 55.

   All calls from an Apama application will be made to these static methods from all contexts.

   As the plug-in author you are responsible for any concurrency concerns.

2. Correlator plug-ins in Java are deployed using a JMon application and are packaged in a jar file. You need to create a JMon deployment descriptor file in the application's `META-INF/jmon-jar.xml` file. For the plug-in you need to add a `<plugin>` to the `<application-classes>` element.

   For more information on Apama deployment descriptor files, see "Creating deployment descriptor files" in *Developing Apama Applications in Java*.

   An example plug-in stanza looks like this:

   ```
   <plugin>
     <plugin-name>TestPlugin</plugin-name>
     <plugin-class>test.TestPlugin</plugin-class>
     <description>A test plugin</description>
   </plugin>
   ```

   - `plugin-name` defines the name visible to EPL.
   - `plugin-class` indicates the class to load from the jar for this plugin.
   - `description` is a simple textual description that appears in log messages.

   Instead of writing a deployment descriptor file manually, if you are using Apama Studio to create the plug-in you can annotate the plug-in class and have Studio automatically generate the descriptor file. Here is an example annotation:

   ```
   @com.apama.epl.plugin.annotation.EPLPlugin(name="TestPlugin",
                                              description="A test plugin")
   class testplugin
   {
      ...
   }
   ```

3. Create a jar file for deploying the plug-in and add the Java class file and the deployment descriptor file `META-INF/jmon-jar.xml` to it. In Apama Studio when you create a JMon application, this is done automatically.

For applications that you plan to inject into a correlator, the recommendation is to create separate jar files for:

- Correlator plug-ins written in Java
- JMon applications

Although the mechanism for creating these jars and describing their meta-data is similar, the interactions of these two different uses of injected jars mean that they will often need to be injected into the correlator separately. The creation of separate jar files ensures that you can inject your application components in the correct order, which is typically:

1. Correlator plug-ins written in Java
2. EPL monitors and events
3. JMon applications

Writing Correlator Plug-ins in Java

Permitted signatures for methods

For a method to be exposed to EPL it must be public, must be static and every argument plus the return type must be one of the following:

Java Entry	EPL Type	Notes
`int`	`integer`	Truncated when passed in, for compatibility
`long`	`integer`	
`String`	`string`	Copy in / copy out
`boolean`	`boolean`	
`double`	`float`	
`java.math.BigDecimal`	`decimal`	Passing in either NaN or infinity throws an exception that kills the monitor instance if not caught.
`com.apama.epl. plugin.Context`	`context`	New type defined for plug-ins
`com.apama.epl.plugin.Channel`	`com.apama.Channel`	New type defined for plug-ins
`Object`	`chunk`	Any Java object can be held in EPL via a chunk

Java Entry	EPL Type	Notes
`TYPE[]`	`sequence<TYPE>`	Any above type *except* `int` can be passed in as an arbitrary-depth nested array->sequence. The sequence is strictly copy-in, non-modifiable, but can be returned as copy-out.
`void`	N/A	Permitted as a return type only

Any method not matching this signature is ignored and logged at `DEBUG`.

Oveloaded functions

Any function with multiple overloads is ignored (none of them are exposed) and this is logged once at `WARN` and once per method at `DEBUG`.

Creating a plug-in using Java

Using Java plug-ins

After you create a correlator plug-in in Java, it must be injected into a Java-enabled correlator before it is available for use in Apama applications. Applications that will use the plug-in also need to import the plug-in by name, as is done with correlator plug-ins written in C or C++.

Injecting

The `.jar` file containing the correlator plug-in must be injected into a correlator that has been started with the `--java` option, which enables support for JMon applications. When using the Apama `engine_inject` utility to inject the `.jar` file, you also need to use the `--java` option.

Importing

Once a Java plug-in has been injected it is available for import using the `plugin-name` defined in the deployment descriptor file. The correlator will automatically introspect the class and make available any suitable, public methods that can be called directly from EPL. For example, the following code imports a plug-in named `TestPlugin` and calls its `dosomething` method:

```
monitor m {
   import "TestPlugin" as test;
   action onload
   {
      test.dosomething();
   }
}
```

Note, if the plug-in `.jar` has been incorrectly injected, the correlator will try to load the plug-in as a C/C++ plug-in and may give an error such as `Error opening plug-in library libfoo.so: libfoo.so: cannot open shared object file: No such file or directory`. If this happens and you were trying to load a plug-in written in Java, then check that the `.jar` file was created and injected correctly before your EPL file was injected.

Deleting

A correlator plug-in can be explicitly deleted by calling `engine_delete` with the application name defined in the deployment descriptor, as with JMon applications. Monitors using the plug-in depend on the plug-in type in the normal fashion. The plug-in will not be deleted until the application and all dependent monitors are deleted.

All JMon applications are loaded into their own classloaders. This means that they have no access to any classes loaded in different `.jar` files. If your plug-in requires any other Java libraries they must either be specified on the correlator's global classpath or injected in the same application `.jar` as the plug-in. Once the application has been deleted, the plug-in can be re-injected and it will be loaded into a new classloader.

Interacting with contexts

Correlator plug-ins can be passed context objects using the `com.apama.epl.plugin.Context` type. The `Context` object is defined as:

```
package com.apama.epl.plugin;
public class Context
{
   public String toString();
   public Context();
   public String getName();
   public int hashCode();
   public boolean isPublic();
   public boolean equals(Context other);
   public static native Context getCurrent();
}
```

The `getCurrent` method returns the context that this method was called from.

Interacting with the correlator

Correlator plug-ins can use the `com.apama.epl.plugin.Correlator` class to send an event, subscribe to a channel, or to specify blocking behavior. The `Correlator` class is defined as:

```
package com.apama.epl.plugin;
public class Correlator
{
   public static native void sendTo(String evt, String chan);
   public static native void sendTo(String evt, Context ctx);
   public static native void sendTo(String evt, Context[] ctxs);
   public static native void sendTo(String evt, Channel c);

   public static native void subscribe(EventHandler handler, String[] channels);
   public static native void unsubscribe(EventHandler handler, String[] channels);
   public static native void unsubscribe(EventHandler handler);

   public static native void enqueue(String evt);
   public static native void enqueueTo(String evt, Context c);

   public static native void pluginMethodBlocking();
}
```

The `Correlator` methods are:

- `sendTo(String, String)` – Sends the event represented in the first `String` to the channel specified in the second `String`. Any contexts and external receivers that are subscribed to the specified channel receive the event. If there are no subscribers the event is discarded.
- `sendTo(String, Context)` – Sends the event represented in `String` to the context referred to by the `com.apama.epl.plugin.Context` argument. An exception is thrown if the context reference is invalid.

- `sendTo(String, Context[])` – Sends the event represented in `String` to the array of contexts referred to by the `com.apama.epl.plugin.Context[]` argument. If one context reference is invalid an exception is thrown and the event is not sent to any context.
- `sendTo(String, Channel)` — Sends the event represented in `String`. If the specified `com.apama.epl.plugin.Channel` object contains a string then the event is sent to the channel that has that name. If `Channel` contains a context then the event is sent to that context.
- `subscribe(EventHandler, String[])` – Subscribes the handler object to the channels listed in the string array. If the handler is already subscribed to some channels then the channels listed in the array are added to the list of existing subscriptions. Subscribing to the same channel multiple times results in a single subscription. However, to completely remove a channel subscription that has been added multiple times you must unsubscribe from that channel the same number of times that it was subscribed to.
- `unsubscribe(EventHandler, String[])` – For the channels specified in the string array, this method removes the subscriptions from the specified handler. It is possible for the result of this method to be that the handler is not subscribed to any channels. Unsubscription from a channel that the handler is not subscribed is ignored.
- `unsubscribe(EventHandler)` – Removes all subscriptions from the specified handler. If this handler is not subscribed to any channels the method is ignored.
- `enqueue(String)` – Adds the event represented in `String` to the back of the input queue of all public contexts. This method is expected to be deprecated and then removed in future releases. Use a `sendTo()` method instead.
- `enqueueTo(String, Context)` – Adds the event represented in `String` to the back of the input queue of the specified context. This method is expected to be deprecated and then removed in future releases. Use a `sendTo()` method instead.
- `pluginMethodBlocking()` – Informs the correlator that the plug-in is potentially blocking for the rest of this call and the correlator is free to spin up additional threads on which to run other contexts. See the example at the end of this topic.

For more information on `com.apama.epl.plugin.Context` and `com.apama.epl.plugin.Correlator`, see the Javadoc reference material available at `doc\javadoc\index.html` in your Apama installation directory.

Receiving events from named channels

A Java plug-in can register callbacks to receive events that are sent to named channels. This is similar to the `monitor.subscribe()` method in EPL. Events are delivered in string form by means of a method on a known interface.

To register a callback, the plug-in must define a class that implements the `com.apama.epl.plugin.EventHandler` interface:

```
public interface EventHandler
{
   void handleEvent(String event, String channel);
}
```

The `handleEvent()` method is called once for each event sent to a channel that this handler is subscribed to, with the channel on which it was received. To manage `EventHandler` object channel subscriptions, use the `subscribe()` and `unsubscribe()` methods on `com.apama.epl.plugin.Correlator`. When a handler is unsubscribed from all channels any in-progress callbacks will complete, but no further callbacks will be made to that handler.

Working with Channel objects

Similar to context objects, you can pass EPL `com.apama.Channel` objects into a Java plug-in. The equivalent Java class is `com.apama.epl.plugin.Channel` and you can use objects of this class to send events to channels. Like the EPL `Channel` type, the Java `Channel` class has three constructors:

```
Channel (String name)
Channel (com.apama.epl.plugin.Context c)
Channel ()
```

A `Channel` object can contain a string that is the name of a channel or it can contain a context. The no-argument constructor creates a `Channel` object that contains an empty context. If you try to send an event to an empty context the `sendTo()` method throws an exception.

You can call the `empty()` method on a Java `Channel` object. It returns true only if the object contains an empty context.

Exceptions

If a method throws an exception, that exception is passed up to the calling EPL and can be caught by the calling monitor. If an exception is not caught it will terminate the monitor instance. Details on catching exceptions in EPL can be found in "Catching exceptions" in *Developing Apama Applications in EPL*.

Persistence

No Java plug-ins are persistent and they are not permitted in a persistent monitor, but they are permitted in non-persistent monitors in a persistent correlator.

Load, unload, and shutdown hooks

If a plug-in needs to run anything when it is loaded, you can do this in a static initializer:

```
public class Plugin
{
   static {
      ... // initialization code here
   }
}
```

It is not natively possible for a plug-in to run anything when it is unloaded. If you need this functionality you can declare a method to be called when the plug-in is unloaded using annotations:

```
public class Plugin
{
   @com.apama.epl.plugin.annotation.Callback(
      type=com.apama.epl.plugin.annotation.Callback.CBType.SHUTDOWN)
   public static void shutdown()
   {
      ... // shutdown code here
   }
}
```

The method must be a public static function which takes no arguments and returns void. Currently, Apama does not support callbacks other than `SHUTDOWN`.

Non-blocking plug-ins and methods

In a correlator some threads have the potential to block and others do not. If a thread might block, the correlator starts new threads if it has additional runnable contexts. By default the correlator assumes that a plug-in call may block and will start additional threads on which to run other contexts. In situations where the plug-in call can never block, the additional overhead of starting new

threads when all CPUs are busy is unnecessary. If you know that a plug-in or an individual method is non-blocking, you can improve efficiency by annotating either entire plug-ins or individual methods as non-blocking.

Note, however, if a method declared as non-blocking does block, the correlator can block all threads waiting for them to finish, resulting in a deadlocked correlator. For methods that are normally non-blocking, but may block in predictable situations, see "Sometimes-blocking functions", below.

Annotations

You can apply the annotation `com.apama.epl.plugin.annotation.NoBlock` with no arguments to either a plug-in class, or to a method on a class:

```
@com.apama.epl.plugin.annotation.NoBlock()
public class Plugin
{
  ...
}
```

When applied to a class, the annotation indicates that no method on the plug-in can ever block.

```
public class Plugin
{
  @com.apama.epl.plugin.annotation.NoBlock()
  public static String getValue() { ... }
}
```

When applied to a method, the annotation indicates that this method will never block, but other methods may block.

Sometimes-blocking functions

If you have a function that usually will not block, but under some known conditions may block, then the method can be declared as `NoBlock` as long as it then uses a callback to indicate when it is starting the potentially-blocking behavior. The callback is a static method on `com.apama.epl.plugin.Correlator` called `pluginMethodBlocking`. This function takes no arguments, returns no value and is idempotent. When it is called, the correlator will then assume that the plug-in is potentially blocking for the rest of this call and is free to spin up additional threads on which to run other contexts.

```
public class Plugin
{
   @com.apama.epl.plugin.annotation.NoBlock()
   public static String getValue()
   {
      if (null != localValue) return localValue;
      else {
         com.apama.epl.plugin.Correlator.pluginMethodBlocking();
         localValue = getRemoteValue();
         return localValue;
      }
   }
}
```

Logging

Correlator plug-ins written in Java can log to the correlator's log file. This is done via the `com.apama.util.Logger` class. Each plug-in must create a static instance of the `Logger` using the static `getLogger` method. This instance provides `debug(...)`, `info(...)`, `warn(...)` and `error(...)` methods, which log a string at that log level in the correlator log file. The level is configured either by means of the correlator command line and management commands or using a log4j configuration file.

For more information on using the `Logger` class, including how to override the default log level, see the Javadoc reference material, available starting with `doc\javadoc\index.html` in your Apama installation directory.

The following is an example of logging in a correlator plug-in:

```
package test;
import com.apama.util.Logger;
public class Plugin
{
   private static final Logger logger = Logger.getLogger(Plugin.class);
   public static void foo()
   {
      logger.info("A string that's logged at INFO");
   }
}
```

This will produce entries in the correlator log file like this:

```
2013-06-11 15:14:21.974 INFO  [1167792448:processing] - <test.Plugin> A string that's logged at INFO
```

Writing Correlator Plug-ins in Java

Sample plug-ins in Java

Apama provides sample correlator plug-ins written in Java, located in the `samples\correlator_plugin\java` directory of the Apama installation. The samples are:

- `SimplePlugin` – a basic plug-in with one method that takes a string, and returns another string.
- `ComplexPlugin` – a plug-in that has several methods and handles more complex types.
- `SendPlugin` – a plug-in that demonstrates passing contexts around and sending events.
- `SubscribePlugin` – a plug-in that shows how to subscribe to receive events sent on a particular channel.

The `samples\correlator_plugin\java` directory contains the Java code for the samples, the EPL code for the Apama applications that call each of the plug-ins, the deployment descriptor files, and an Ant `build.xml` file for building all of the samples. The directory also contains a `README.txt` that describes how to build and run the samples as well as text files that depict what the output of the samples should be like.

Writing Correlator Plug-ins in Java

A simple plug-in in Java

The simple plug-in sample is comparable to the similar C and C++ simple plug-in samples.

The Java code for the `simple_plugin` class contains the public static `test` method. (Methods that will be called from EPL code need to be public and static.)

```
public class SimplePlugin
{
  public static final String TEST_STRING = "Hello, World";
  public static String test(String arg)
  {
    System.out.println("SimplePlugin function test called");
    System.out.println("arg = "+arg);
```

```
    System.out.println("return value = "+TEST_STRING);
    return TEST_STRING;
  }
}
```

The deployment descriptor file contains the following `<plugin>` stanza that illustrates how to specify the plug-in.

```
  <application-classes>
    <plugin>
      <plugin-name>SimplePlugin</plugin-name>
      <plugin-class>SimplePlugin</plugin-class>
      <description>A test plugin</description>
    </plugin>
  </application-classes>
```

The EPL code imports the plug-in and calls the `test` method.

```
monitor SimplePluginTest {
  // Load the plugin
  import "SimplePlugin" as simple;
// To hold the return value
  string ret;
  string arg;
  action onload {
    // Call plugin function
    arg := "Hello, Simple Plugin";
    ret := simple.test(arg);
    // Print out return value
    log "simple.test = " + ret at INFO;
    log "arg = " + arg at INFO;
  }
}
```

Sample plug-ins in Java

A more complex plug-in in Java

The complex plug-in sample is comparable to the corresponding C and C++ complex plug-in samples.

The Java code for the `ComplexPlugin` class contains the public static methods: `test1`, `test2`, `test3`, and `test4`. It also contains an object, `ComplexChunk` that represents a complex type.

```
class ComplexChunk
{
  long size;
  double[] data;
  public ComplexChunk(int size)
  {
    this.size = size;
    this.data = new double[size];
    for (int l = 0; l < size; ++l) {
      this.data[l] = l;
    }
    print();
  }
  public void print()
  {
    System.out.println("Chunk size = "+size);
    for (int l = 0; l < size; ++l) {
      System.out.println("Chunk element ["+l+"] = "+data[l]);
    }
  }
}
/**
```

```
  A more complex correlator plugin written in Java with multiple methods, arrays and chunks.
*/
public class ComplexPlugin
{
  /** Prints the arguments and returns a string */
  public static String test1(long l, double f, boolean b, String s)
  {
    System.out.println("integer value = "+l);
    System.out.println("float value = "+f);
    System.out.println("boolean value = "+b);
    System.out.println("string value = "+s);
    return "Hello, World";
  }
  /** Prints the (array) arguments and returns a string */
  public static double test2(long[] ls, double[] fs,
                             boolean[] bs, String[] ss)
  {
    System.out.println("sequence size = "+ls.length);
    for (int i = 0; i < ls.length; ++i) {
      System.out.print("sequence element["+i+"]: ");
      System.out.println("integer value = "+ls[i]);
    }
    System.out.println("sequence size = "+fs.length);
    for (int i = 0; i < fs.length; ++i) {
      System.out.print("sequence element["+i+"]: ");
      System.out.println("float value = "+fs[i]);
    }
    System.out.println("sequence size = "+bs.length);
    for (int i = 0; i < bs.length; ++i) {
      System.out.print("sequence element["+i+"]: ");
      System.out.println("boolean value = "+bs[i]);
    }
    System.out.println("sequence size = "+ss.length);
    for (int i = 0; i < ss.length; ++i) {
      System.out.print("sequence element["+i+"]: ");
      System.out.println("string value = "+ss[i]);
    }
    return 2.71828;
  }
  /** Returns a chunk containing an array of 'l' doubles */
  public static Object test3(int l)
  {
    return new ComplexChunk(l);
  }
  /** Takes a chunk from test3, mutates it and prints it */
  public static void test4(Object o)
  {
    ComplexChunk cc = (ComplexChunk) o;
    for (int i = 0; i < cc.size; ++i) {
      cc.data[i] = Math.sqrt(cc.data[i]);
    }
    cc.print();
  }
}
```

The `complex_plugin.xml` file is the plug-in's deployment descriptor and contains the following `<plugin>` stanza that specifies the name, class, and description for the plug-in.

```
  <application-classes>
    <plugin>
      <plugin-name>ComplexPlugin</plugin-name>
      <plugin-class>ComplexPlugin</plugin-class>
      <description>A test plugin</description>
    </plugin>
  </application-classes>
```

The sample's `ComplexPlugin.mon` file contains the EPL code for the Apama application. It imports the plug-in and calls the various `testx` methods:

```
monitor ComplexPluginTest {
```

```
  // Load the plugin
     import "ComplexPlugin" as complex;
  // To hold the return values
  string str1;
  string ret1;
  float ret2;
  // Opaque chunk value
  chunk myChunk;
  // Loop counter
  integer i;
  // Sequences for test2
  sequence<integer> intSeq;
  sequence<float> floatSeq;
  sequence<boolean> boolSeq;
  sequence<string> stringSeq;
  action onload {
    // Call test1 function
    str1 := "Hello, Complex Plugin";
    ret1 := complex.test1(42, 3.14159, true, str1);
    log "complex.test1 = " + ret1 at INFO;
    log "str1 = " + str1 at INFO;
    // Initialize sequences
    intSeq.setSize(10);
    floatSeq.setSize(10);
    boolSeq.setSize(10);
    stringSeq.setSize(10);
    i := 0;
    while (i < 10) {
      intSeq[i] := i;
      floatSeq[i] := i.toFloat();
      boolSeq[i] := true;
      stringSeq[i] := "How long is a piece of string?";
      i := i + 1;
    }
    // Call test2 function
    ret2 := complex.test2(intSeq, floatSeq, boolSeq, stringSeq);
    log "complex.test2 = " + ret2.toString() at INFO;
    i := 0;
    while (i < 10) {
      log "intSeq[" + i.toString() + "] = " + intSeq[i].toString()
          at INFO;
      log "floatSeq[" + i.toString() + "] = " + floatSeq[i].toString()
          at INFO;
      log "boolSeq[" + i.toString() + "] = " + boolSeq[i].toString()
          at INFO;
      log "stringSeq[" + i.toString() + "] = " + stringSeq[i].toString()
          at INFO;
      i:= i + 1;
    }
    // Generate a new chunk
    myChunk := complex.test3(20);
    // Do some computation on the chunk
    complex.test4(myChunk);
  }
}
```

Sample plug-ins in Java

A plug-in in Java that sends events

This sample plug-in shows how to pass contexts around and how to send events to specific contexts.

The Java class for the plug-in imports `com.apama.epl.plugin.Context` and `com.apama.epl.plugin.Correlator` and it declares a public method that sends an event to a channel and another public method that sends an event to a particular context:

```
import com.apama.epl.plugin.Context;
import com.apama.epl.plugin.Correlator;
/**
  Demonstrates passing contexts around and sending events
*/
public class SendPlugin
{
   public static void sendEventToChannel(String event, String channel)
   {
      // Send the event to the channel
      Correlator.sendTo(event,channel);
   public static void sendEventTo(String event, Context context)
   {
      // Get the current context we're running in
      Context current = Context.getCurrent();
      // For some reason we don't want to let you do this.
      // An exception will terminate the monitor instance
      if (current.equals(context))
      {
         throw new IllegalArgumentException("Please don't use this function
            to send events to the current context!");
      }
      // Send the event to the other context
      Correlator.sendTo(event, context);
   }
}
```

The `SendPlugin.xml` deployment descriptor file contains the name, class, and description of the plug-in in the `<plugin>` stanza:

```
  <application-classes>
    <plugin>
      <plugin-name>SendPlugin</plugin-name>
      <plugin-class>SendPlugin</plugin-class>
      <description>A test plugin</description>
    </plugin>
  </application-classes>
```

The Apama application `SendPlugin.mon` first imports the plug-in and then calls the plug-in's `sendEventToChannel()` method as well as its `sendEventTo()` method with a variety of contexts.

```
event A {
  string str;
}
monitor SendPluginTest {

   // Load the plugin
   import "SendPlugin" as send_plugin;

   action onload {
      context c1 := context("receiver");
      context c2 := context("receiver2");
      context c3 := context("sender");
      context c4 := context("sender2");
      context c5 := context("sender3");
      spawn receiver() to c1; // listen for events
      spawn channelReceiver("SampleChannel") to c2; // listen for events
      spawn sendto(c1) to c3; // this will work
      spawn sendto(c4) to c4; // this won't work because both contexts are the same
      spawn sendtoChannel("SampleChannel") to c5;
   }

   action channelReceiver(string chan)
      {
         monitor.subscribe(chan);
         A a;
         on all A(): a {
            print "channel receiver got: "+a.toString();
         }
   }
```

```
   action sendtoChannel(string chan)
   {
      on all A() {
         print "sender really shouldn't get anything!";
      }
      send_plugin.sendEventToChannel(A("Hello, World").toString(), chan);
   }

   action receiver()
   {
      A a;
      on all A(): a {
         print "receiver got: "+a.toString();
      }
   }

   action sendto(context c)
   {
      on all A() {
         print "sender really shouldn't get anything!";
      }
      send_plugin.sendEventTo(A("Hello, World").toString(), c);
   }
}
```

Sample plug-ins in Java

A plug-in in Java that subscribes to receive events

This sample shows how a plug-in subscribes to receive events sent on a particular channel. This sample is comparable to the similar C and C++ subscription plug-in samples.

The Java code for the `SubscribePlugin` class contains the public static `createHandler` method. (Methods that will be called from EPL code need to be public and static.)

```
import com.apama.epl.plugin.Correlator;
import com.apama.epl.plugin.EventHandler;

/** Demonstrates a plugin subscribing to channels to receive events */

public class SubscribePlugin
{
   public static class SubscribeHandler implements EventHandler
   {
      public void handleEvent(String event, String channel)
      {
         System.out.println("Got event "+event+" on channel "+channel);
      }
   }
   public static Object createHandler(String channel)
   {
      SubscribeHandler h = new SubscribeHandler();
      Correlator.subscribe(h, new String[] { channel });
      return h;
   }
}
```

The deployment descriptor file contains the following `<plugin>` stanza that illustrates how to specify the plug-in.

```
  <application-classes>
    <plugin>
      <plugin-name>SubscribePlugin</plugin-name>
      <plugin-class>SubscribePlugin</plugin-class>
      <description>A test plugin</description>
```

```
    </plugin>
  </application-classes>
```

The EPL code imports the plug-in and calls the `createHandler()` method.

```
event A {
   string str;
}

monitor SubscribePluginTest
{
  // Load the plugin
  import "SubscribePlugin" as subscribe_plugin;

  action onload {
    chunk handler := subscribe_plugin.createHandler("SampleChannel");
    send A("Hello World") to "SampleChannel";
  }
}
```

Sample plug-ins in Java