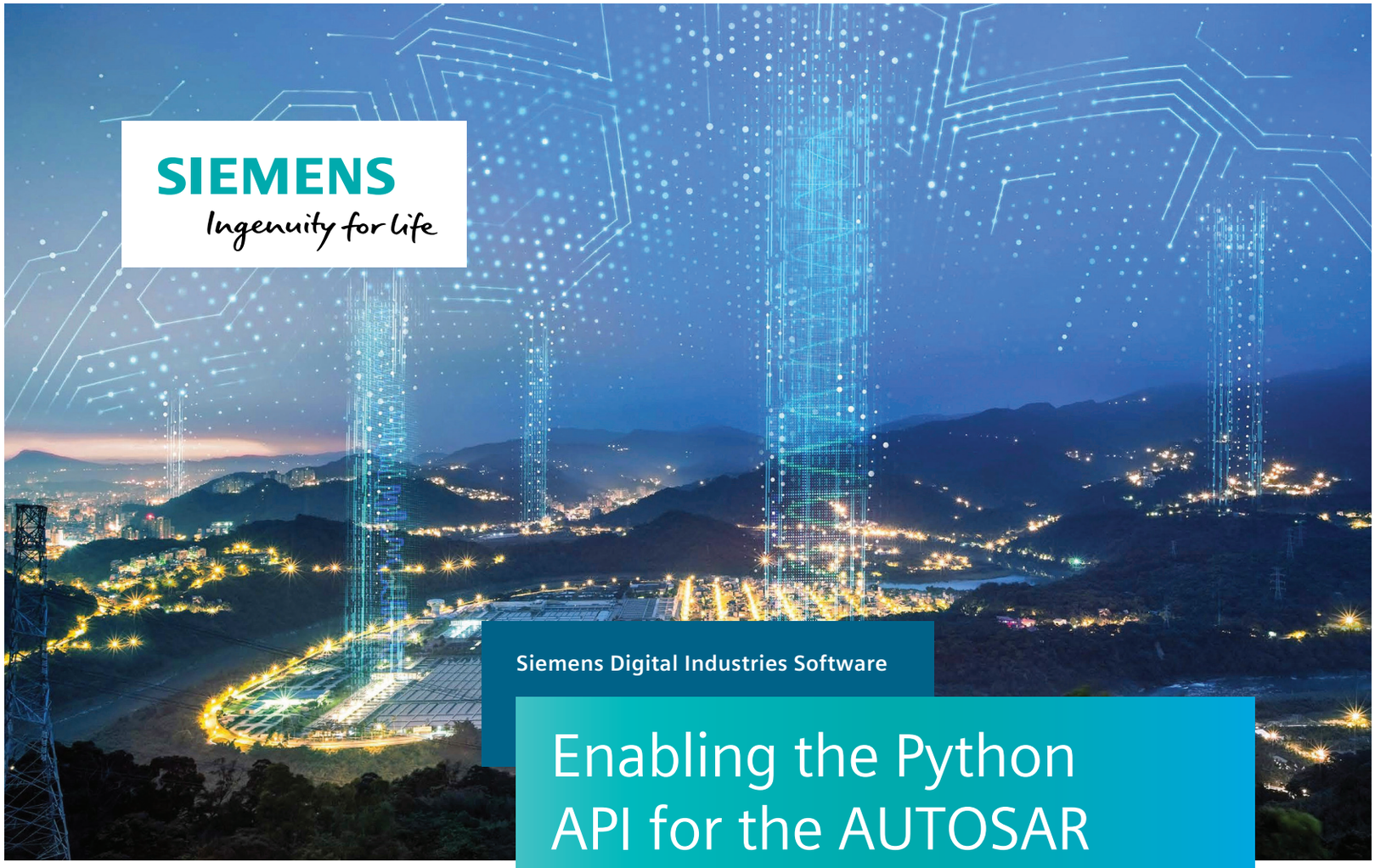

Siemens Digital Industries Software

Enabling the Python API for the AUTOSAR Adaptive Platform

Embedded systems

Executive summary

Analyzing the high-level APIs of the most widely used ML frameworks such as Tensorflow, PyTorch, Keras, Gluon, Chainer and Onnx, it's easy to recognize that the dominance of the Python language is overwhelming. The intensive computations running on a GPU are programmed in low-level code, but Python appears to be convenient for defining and configuring the algorithms in high-level code. In this paper we explore how the Python API for the AUTOSAR Adaptive platform can enable researchers to focus on the mathematical problems instead of dealing with the excess compilation time and possible debugging quirks of a C++ program.

Karoly Molnar
AUTOSAR Software Architect

Introduction

The automotive software industry is making continuous attempts to standardize certain common components in order to improve the compatibility between different vendors and reduce cost. In the past decade, the AUTOSAR consortium has played a key role in defining methodology, protocols and middleware functionality to handle the complexity of the software design of a modern car. Considering embedded middleware, the consortium has focused initially on microcontroller-based distributed systems under the term AUTOSAR Classic. In the past years, however, the demand for more powerful software algorithms and the supply of capable hardware has made possible the introduction of the new set of specifications. This new AUTOSAR Adaptive Platform aims to meet the recent challenges of the industry by defining service oriented communication concepts and methods to upgrade software using modern APIs.

The AUTOSAR Foundation Specification mandates "appropriate language bindings" at the highest level, but it does not define which language to use. However, the AUTOSAR Adaptive specification set is refined to C++, or more precisely C++11.

AUTOSAR does so with a good reason, the chosen language, C++11 is powerful and mature. With proper constraints, it can be used in performance-optimized embedded systems for automotive. It is therefore assumed the remainder of the AUTOSAR Adaptive standard is specifying and referring to variables, classes and APIs in C++ syntax. This a versatile and widely available language that could satisfy all goals that the automotive industry is set on in-vehicle application software.

Looking at it from a wider perspective, the Siemens Digital Industries Software automotive software team believes that each project shall use the tools that are optimal for reaching the goal. In this context, the programming language is just a tool. If a given development activity benefits from using another programming language then better results can be achieved in shorter time.

Therefore, the goal of this paper is to explore several use cases that benefit software developers from binding Adaptive Runtime for AUTOSAR (ARA) to other programming languages.

The consequence of recent changes in the automotive industry

In the past decade we have experienced a major shift in the automotive industry with new innovations around ADAS and EV technologies – emerging at a very fast pace. Detailing these innovations is not in the scope of this paper, but the paper will emphasize a number of aspects that may justify considering alternative programming languages.

The importance of rapid prototyping

The R&D path from idea to final product is seldom straightforward. The necessary experimentation to unfold a concept requires different tools compared to the tools used for creating the final product. The tools that enable creation of rapid prototypes may be a good addition to the innovator's toolset. In this sense, C++ is normally not the fastest language to write code in, or even to compile, to say at least. There are programming languages that may fit better to the need for faster design iterations in early phases.

Bringing programming languages of innovation and research into automotive

The increased attention on ADAS solutions has resulted in a growing demand for data scientists and researchers in the automotive job market. These Machine Learning (ML) experts have strong programming skills and are comfortable with C++. However, in ML research it is more common to use other high-level programming languages. By enabling these scientists to use the programming language that they are most comfortable with their work efficiency could significantly improve.

The Python API for AUTOSAR Adaptive Platform

By analyzing the high-level APIs of the most widely used ML frameworks such as Tensorflow, PyTorch, Keras, Gluon, Chainer and Onnx, it's easy to recognize that the dominance of the Python language is overwhelming.

There is a reason for this. The intensive computations running on a GPU are programmed in low-level code, but Python appears to be very handy for defining and configuring the algorithms in high-level code. This allows researchers to focus on the mathematical problems instead of dealing with the excess compilation time and possible debugging quirks of a C++ program.

Recognizing the gap between the high-level APIs, the ML frameworks and their automotive applications, Siemens has created a set of Python packages for AUTOSAR. These packages allow simple, high-level access to AUTOSAR APIs.

Architectural considerations

In order to provide the Python APIs, two fundamental challenges had to be solved.

#1: API Syntax

We have chosen a highly pragmatic approach mimicking the C++ APIs as much as possible, and in a most practical manner. The primary aim was to minimize the difference between the API names in order to allow faster transition later from the prototype model to the production code.

#2: Native implementation vs binding
There were two options; whether the Python APIs should be built as a complete replacement of the C++ runtime, or in a form of a binding layer on top of C++ (figure 1).

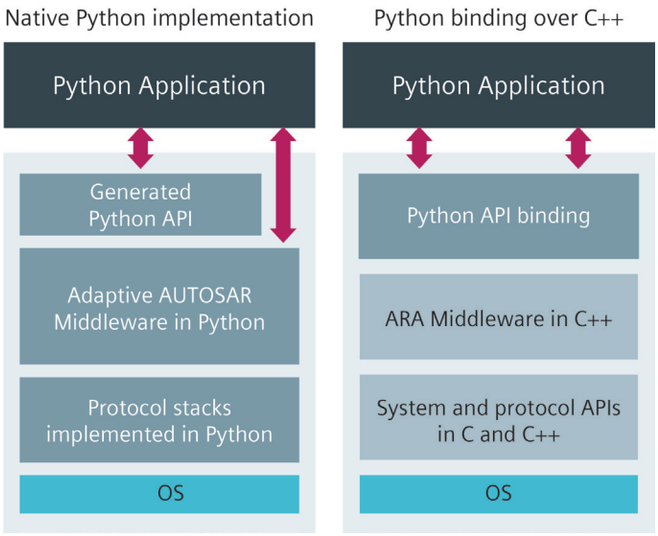

Figure 1: Architectural choices with different extent of using Python in layered architecture.

While both approaches have their distinct benefits, there were stronger arguments for building Python support as a binding layer on top of the existing C++ runtime.

Specifically, these were the benefits/advantages of merit:

- Smaller code base can follow the standard API changes easier
- Better CPU load balancing with C++ multithreading
- Re-use of third party components available in C++ only

One could argue that any Python application can implement its own persistency and logging in a few lines of code without the overhead of using ARA-compliant architecture. While this is true for the simple use cases, the advantage of using the redundancy and encryption features of the ARA persistency, or the interfacing to the standard DLT logging, would not be available. One of the key benefits of AUTOSAR is that it encapsulates years of automotive lessons learned across the industry.

Wherever the API is dynamically generated from the configuration, the related Python API is generated. This is primarily applicable for COM in the first iteration of the binding solution. The generator is using the very same manifest artifacts as each of the C++ generator back-ends. This means no additional configuration is needed for the sake of Python bindings.

In certain functional clusters such as logging, persistency, or execution management, the Python APIs remain static, following the behavior of the underlying C++ implementation.

Figure 2 provides an architectural overview with selected functional clusters. The binding layer is on top of the ARA Functional Cluster APIs.

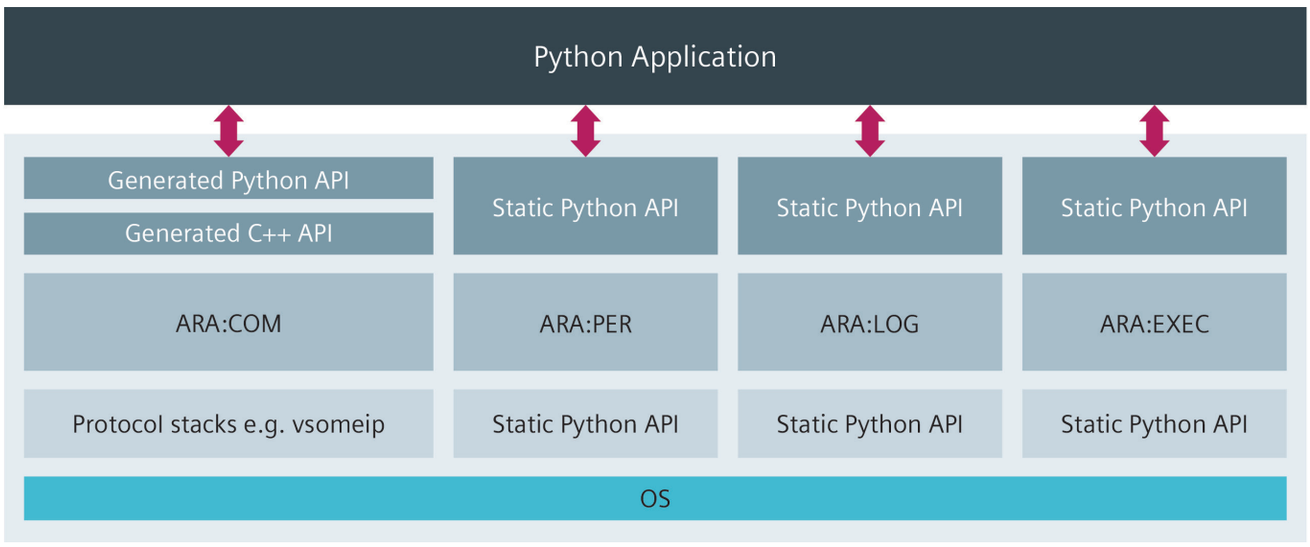

Figure 2: The boxes in light blue color are the scope of this paper. The green boxes represent the standard Adaptive stack.

Practical implementation details

The goal is to provide a Python API facility for a set of features that would be immediately usable for an algorithm developer. Therefore, the initial set of binding packages are available for Communication Manager, Persistency, Log and Trace and Execution Management.[1] The APIs support the latest 19-03 AUTOSAR specification.

A number of representative code snippets are demonstrated below in order to provide some understanding on the available Python APIs.

Log and trace

Logging is fully supported. Logged data is reported as String for convenience. This simple example shows how to log information from a Python application that ends up in ARA Log and Trace:

Python

```
import py_ara_log as log
log.InitLogging("APPY", "Logging ID", log.LogLevel.kWarn, log.LogMode.kConsole)
ctx = log.CreateLogger("CTXX", "test context")
ctx.LogFatal("Fatal Error happened!")
```

C++

```
#include "ara/log/logging.h"
using namespace ara::log;
InitLogging("APPY", "Logging ID",LogLevel::kWarn, LogMode::kConsole,"");
Logger& ctx = CreateLogger("CTXX", "test context");
ctx.LogFatal() << " Fatal Error happened!";
```

The steps of initializing the logging and creating a context are the same as in C++. Similarly, after the logger context is created, the actual logging can be performed via the usual LogFatal, LogError, LogWarn, LogInfo, among other methods.

Persistency

The supported features include both Key-Value Storage (KVS) and a File storage-type of Persistent storage. ARA compliant Persistency can be also used via a few lines of code:

Python

```
import py_ara_per as per
kvsDatabase = per.OpenKeyValueStorage("applDatabase.json")
kvsDatabase.SetInt32Value("thisIntValue", 42)
val = kvsDatabase.GetStringValue("otherStrValue")
```

C++

```
#include "ara/per/key_value_storage.h"
auto kvsDatabase = ara::per::OpenKeyValueStorage("applDatabase.json");
kvsDatabase.Value()->SetValue("thisIntValue", static_cast<int32_t> 42);
auto val = kvsDatabase.Value()->GetValue("otherStrValue")
```

In the above example,a KVS Persistent storage mechanism is used with an underlying database, which is a JavaScript object notation (JSON) file in our use case. A value is written and another is read from the database.

Communication

The language binding supports publishing of and subscribing to services. APIs are used to handle events, methods and fields and are available on both the proxy and skeleton sides including the necessary call-back methods. The ARA:COM APIs both on the Proxy and Skeleton side are heavily configuration dependent. Therefore, the binding layer built on top of the generated C++ APIs are dynamically created as well.

1. The Communication Management cluster is responsible for interaction between adaptive applications. Persistency is responsible for non-volatile data storage. The Log and Trace cluster handles logging both locally and remotely. Execution Management is responsible for managing application startup and shutdown.

In the example below the imported controllerproxyimpl package is a generated Python module wrapping C++ interfaces. In order to initialize and use the features from Python, the same steps must be performed as in the C++ counterpart.

Python

```
from controllerproxyimpl import ara, mentor, SensorProxyImpl

# callback function that is called when an appropriate Service is found
def SensorServiceFound(handles, handler):
    if handles:
        SensorHandle = handles[0]
        global SensorServiceProxy
        SensorServiceProxy = SensorProxyImpl(SensorHandle)

# We need an appropriate Service for the SensorProxy. So we call StartFindService. If
# the matching Service is found, the function SensorServiceFound is called.
mentor.demo.sensor.proxy.SensorProxy.StartFindService(SensorServiceFound)

# We register a callback function for the received Events.
SensorServiceProxy.SensorEvent.SetReceiveHandler(EventReceived)

# The EventReceived callback function is not listed here, but in general it shall
# subscribe to the interesting Events, like this:
SensorServiceProxy.SensorEvent.Subscribe(ara.com.EventCacheUpdatePolicy.kNewestN, 1)

# The Set Method of the SensorField is called,
# that will change the value remotely on the Service side:
SensorServiceProxy.SensorField.Set(10)
```

C++

```
#include “mentor/demoapp/sensor/sensor_common.h“
#include “mentor/demoapp/sensor/sensor_proxy.h”

void SensorServiceFound( ara::com::ServiceHandleContainer
<mentor::demoapp::sensors::proxy::sensorProxy::HandleType> handles, ara::com::FindServiceHandle handler)
{
    if(handles.size() > 0) {
        mentor::demoapp::sensors::proxy::sensorProxy::HandleType SensorHandle = handles.front();
        SensorServiceProxy = std::make_shared
        <mentor::demoapp::headlights::proxy::headlightsProxy>(headlightsHandle);
    }
}
auto Sensor_fsh =
mentor::demoapp::sensors::proxy::sensorProxy::StartFindService(SensorServiceFound);
SensorServiceProxy->SensorEvent.SetReceiveHandler(EventReceived);
SensorServiceProxy->SensorEvent. Subscribe(ara::com::EventCacheUpdatePolicy::kNewestN, 1);
SensorServiceProxy->SensorField.Set(10);
```

Execution management

The API set of the ExecutionClient class are fully supported. This allows Python applications to report back their operation status to the Execution Manager. Note: DeterminsticClient APIs are not supported.

In the example below, the application reports back it's running state to the ExecutionManager using just a few lines of code.

Python

```
import py_ara_exec as exec
em = exec.ExecutionClient()
em.ReportExecutionState(exec.ExecutionState.kRunning)
```

C++

```
#include <ara/exec/execution_client.h>
ara::exec::ExecutionClient em;
em.ReportExecutionState(ara::exec::ExecutionState::kRunning);
```

Machine learning support

A complete machine learning example would not fit in the extent of this paper. Therefore, only a few lines of code are presented to demonstrate the simplicity of using an ML framework with the ARA Python binding API:

```
import torch
import py_ara_log as log
import roadSignDetectorSkeletonImpl

# Initializing ARA LOG infrastructure
log.InitLogging("SIGNS", "PyTorch ARA example", ara_log.LogLevel.kInfo, ara_log.LogMode.kConsole)
logger_signs = ara_log.CreateLogger("MAIN", "Main log context")
logger_signs.LogInfo("Main started")

# Offering signs service for other applications
signs = roadSignDetectorSkeletonImpl(ara.com.InstanceIdentifier(0x02))
signs.OfferService()

p_in = torch.autograd.Variable(torch.randn(10, 10))
p_out = torch.autograd.Variable(torch.randn(10, 20), requires_grad=False)
model = torch.nn.Sequential(torch.nn.Linear(10, 32),torch.nn.Linear(32, 20))
f_loss = torch.nn.MSELoss()
loss = f_loss(model(p_in), p_out)

# Logging loss data to ARA LOG
logger_signs.LogInfo("Loss Data " + str(loss.data[0]))

# Sending loss data to loss event of signs service via ARA COM
signs.loss.Send(loss.data[0])
```

In the example on the previous page, the results of a random loss calculation of a PyTorch model are provided via service events. The lines marked with yellow are related to ARA COM functionality for Service Initialization and Event sending. The blue lines are for logging to the ARA Log and Trace.

What's left out?

While the covered functionality and the number of supported APIs may grow over time, there are a number of domains that are just not practical to introduce in interpreted languages. Features that are related to safety critical operation or hard, real-time behavior are not in the current scope of the Python binding layer. Given this philosophy, the ARA:EXEC DeterministicClient APIs or the Platform Health Management APIs are not in the scope of this package.

OS support and third-party code

The Python package provided by Siemens runs on desktop Linux® out-of-the-box. A Yocto Project-based Linux can also be used with minor configuration. Upon request other embedded solutions can also be supported.

Conclusion

Sometimes the complexity of the safety and performance oriented C++ API makes it difficult to use in the early phases of the development. Using Python binding provides the synergy between the simplicity of the Python language and the versatility of AUTOSAR Adaptive Platform. Python, being the most widely used language of major ML frameworks, helps data scientists with academic backgrounds deliver results quickly in automotive prototypes. The resemblance of the API names allow the team to quickly familiarize themselves with the AUTOSAR concepts and the syntax of the final C++ product.

Recognizing the gap between the high-level APIs, the ML frameworks and their automotive application, Siemens has created a set of Python packages for AUTOSAR. These packages provide simple, high-level interfaces between Python programs and AUTOSAR Adaptive.

To learn more about Siemens' involvement in AUTOSAR, please visit this website.

References

1. AUTOSAR_RS_Main.pdf, Version 1.5.1. "AUTOSAR Main Requirements."

Siemens Digital Industries Software

Headquarters
Granite Park One
5800 Granite Parkway
Suite 600
Plano, TX 75024
USA
+1 972 987 3000

Americas
Granite Park One
5800 Granite Parkway
Suite 600
Plano, TX 75024
USA
+1 314 264 8499

Europe
Stephenson House
Sir William Siemens Square
Frimley, Camberley
Surrey, GU16 8QD
+44 (0) 1276 413200

Asia-Pacific
Unit 901-902, 9/F
Tower B, Manulife Financial Centre
223-231 Wai Yip Street, Kwun Tong
Kowloon, Hong Kong
+852 2230 3333

About Siemens Digital Industries Software

Siemens Digital Industries Software is driving transformation to enable a digital enterprise where engineering, manufacturing and electronics design meet tomorrow. Our solutions help companies of all sizes create and leverage digital twins that provide organizations with new insights, opportunities and levels of automation to drive innovation. For more information on Siemens Digital Industries Software products and services, visit siemens.com/software or follow us on LinkedIn, Twitter, Facebook and Instagram. Siemens Digital Industries Software – Where today meets tomorrow.

About the author

Karoly Molnar is an AUTOSAR Software Architect at Siemens. During Karoly's 14 years of service at Siemens he has been a software development engineer, an operating manager, to his current position as software architect, which he has held since 2014. Mr. Molnar received a master's degree in electrical engineering from the University of Budapest.

siemens.com/software

81411-C3 9/19 N