FINAL PRESENTATION

EGS-CC PROCEDURE EXCHANGE WITH OTX

EGS-CC PROCEDURE EXCHANGE WITH OTX

Agenda

Agenda

1. Introduction
2. OTX (Open Text sequence eXchange Format)
3. Procedures Exchange with OTX
4. Support for the EGS-CC
5. OTX Development Environment (IDE)
6. Open Test Framework (OTF)
7. The Proof of Concept (PoC)
8. Demo
9. Further Activities
10. A.O.B.

EGS-CC PROCEDURE EXCHANGE WITH OTX

Introduction

Introduction

Introduction

Main objective: Analysis of suitability of OTX as a common procedure exchange format between EGS-CC-based systems.

It should be considered that in the target scenario, each of the parties involved in a mission (manufacturer, providers, operators, etc.), **using their own language** are able to exchange procedures for their execution in different contexts that covers their lifetime.

Aiming at this objective, this Study has performed a formal / theoretical analysis of the OTX language compared to the selected DSLs used in the Space Industry: **SPELL**, **PLUTO**, **TOPE** and **ISIS**.

The **Proof of Concept** (PoC) prototype supports the analysis of key aspects of the eventual adoption of OTX as exchange language in EGS-CC. In this particular case, the effective translation has been restricted to SPELL. In a further translation, the OTX has been converted into Java code, executed against the MCM emulator.

The provided **MCM emulator** keeps certain degree of fidelity to the EGS-CC philosophy and structures, according the currently available documentation

The Study has followed two main objectives:

- **Translation** of a procedure from **DSL to OTX and back to the DSL**
- **Execution** of the translated procedure in an EGS-CC-compliant runtime environment to proof the compatibility with EGS-CC concepts like the MCM (PoC provided emulator)

Debugging capabilities have been demonstrated at OTX level, although the debugging at DSL environment level has been also considered but in a theoretically way

EGS-CC PROCEDURE EXCHANGE WITH OTX

The OTX (Open Test eXchange) language

What's OTX?

OTX – **O**pen **T**est sequence e**X**change, standardized in ISO 13209

Domain specific language (DSL) on high abstraction level to create **executable test sequences**

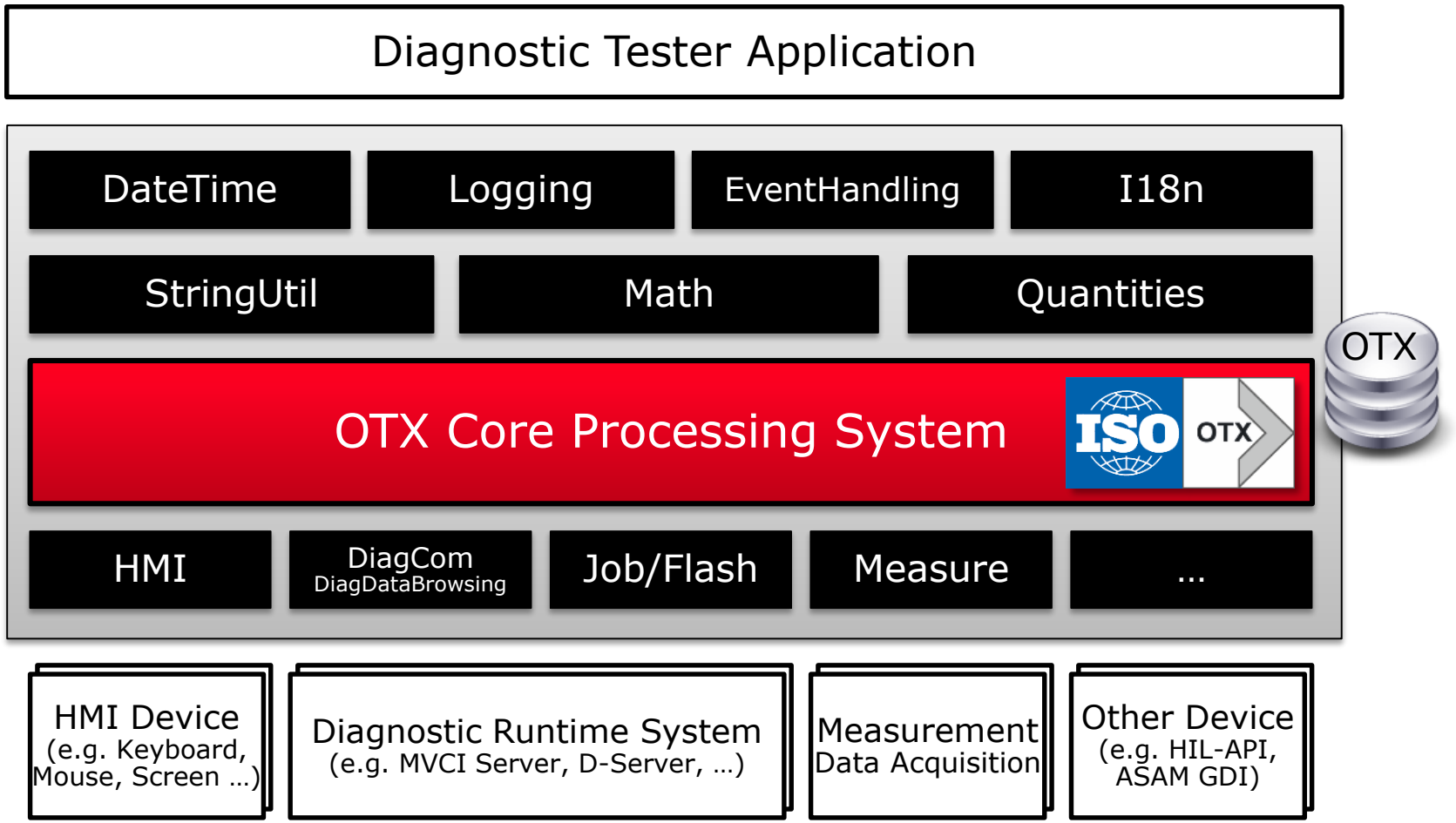
Platform and tester independent **exchange format** for formal description of test sequences

Includes **domain specific concepts** to reduce the presentation to the actual test logic

Application area: Vehicle diagnostics and Test automation

Goal: Creation, exchanging, archiving and execution of verified test sequences

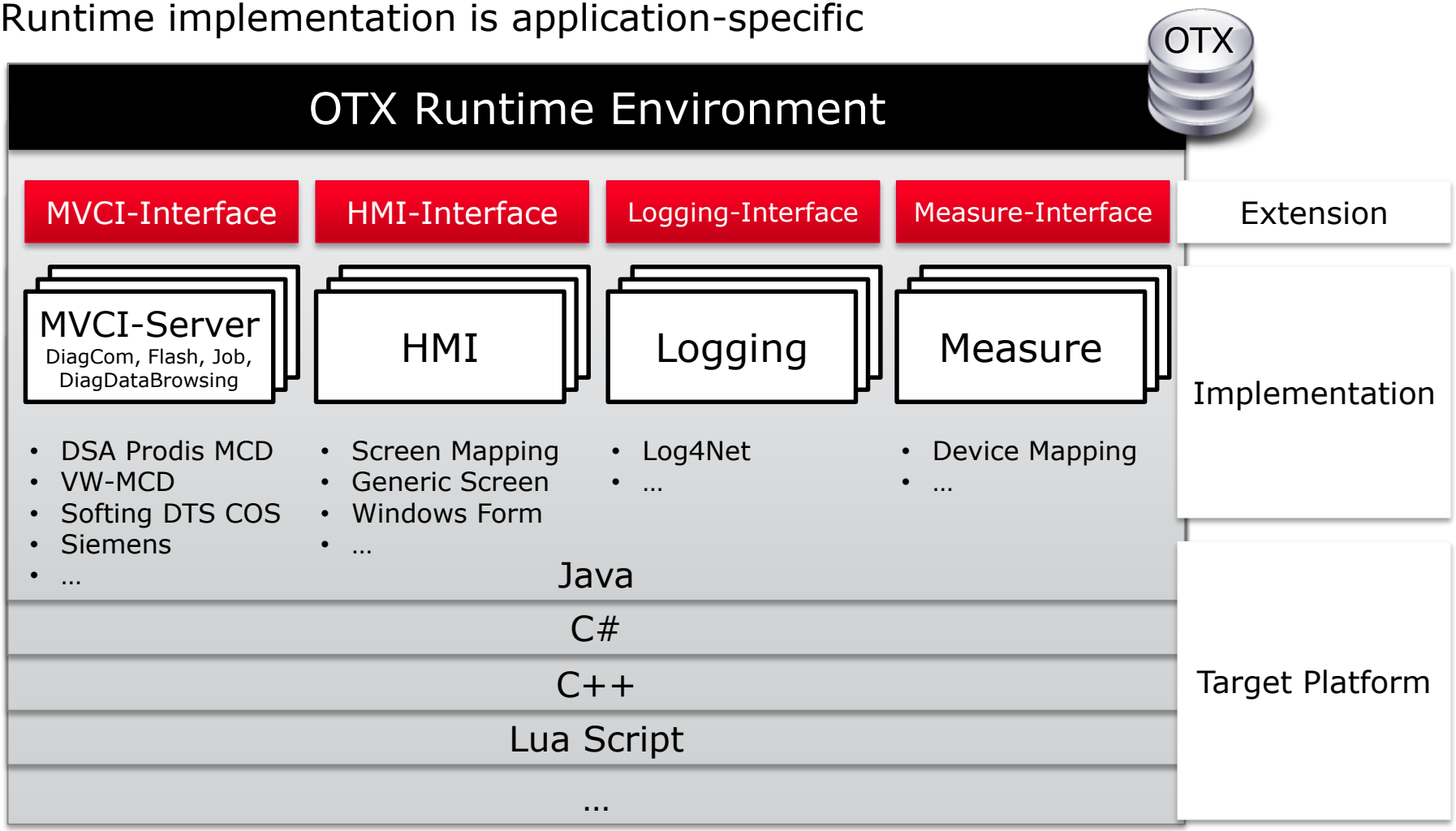
OTX Extensions



OTX Execution – Runtime Environment

OTX only describes the test logic

Runtime implementation is application-specific



Main advantages of OTX

OTX is Domain Specific

Specialized language with a reduced, well-formed command set to describe and execute Tests in an industrial environment

- Allow solutions to be expressed in the idiom and at the level of abstraction of the problem domain
- Self-documenting and better readable code
- Less technical code with less redundancy
- **Allows validation of semantic errors during design time (> 100 checker rules)**
- Easy to learn and to understand
- Enhanced quality, productivity, reliability, maintainability, portability and reusability

OTX is an ISO Standard

Gives each user the guarantee that investments are protected

- Long term archiving of testing knowledge
- Availability of standard tool solutions

OTX Conclusion

Automotive moves from proprietary solutions to standards to be able to manage the increasing complexity: Number and variants of Electric/Electronic components in the car.

Ongoing transition from proprietary solutions in C++ to Java and now to OTX to standardize testing procedures

OTX was created to replace Java-Jobs in diagnostics applications as these are not process safe

OTX is already proven in automotive industry

OTX allows to concentrate on test logic abstracting from implementation details

EGS-CC PROCEDURE EXCHANGE WITH OTX

Procedures Exchange with OTX

General Translation Aspects

The translation from one programming language to another is a **complex task**

A **programming language** is a formal constructed language designed to communicate instructions to a computer

A programming language consist of **Syntax** (form) and **Semantic** (meaning)

The Syntax consist of a **Data Model** and the **Notation**

The **Data Model** mainly consist of I/O Commands, Type System, Control Structures and a set of basic methods to handle numbers and characters

Except the **Type system** the Data Model of different programming languages are similar but mostly not identical

Type Systems:	Java	SPELL	OTX
Weakness	Strong	Strong	Strong
Validation	Static	Dynamic	Static
Explicit / Implicit	Explicit	Implicit	Explicit

For **Execution** a programming language have to be translated into an executable format.

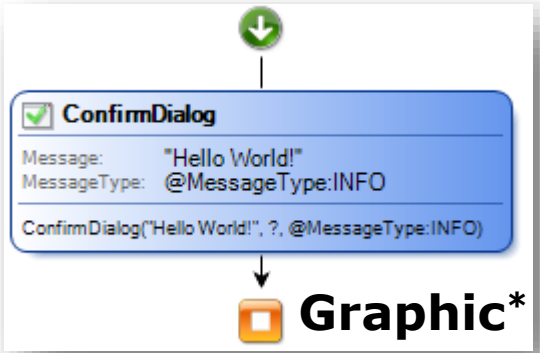
Same Data Model (OTX) different Notation

```xml
<otx xmlns="http://iso.org/OTX/1.0.0" xmlns:hmi="http://iso.org/OTX/1.0.0/HMI"
     id="id_0a4aa7adb9484acd80fd9b8cc0889d1c" name="MyFirstDocument"
     package="NewDefaultPackage1" version="1.0.0.0" timestamp="2014-09-08T17:41:38.7630549+07:00">
  <procedures>
    <procedure id="id_9fd318a128cd4945aa7ed8d5cd57c0b8" name="main" visibility="PUBLIC">
      <realisation>
        <flow>
          <action id="id_80c40aeaec1a4b4bb23cbb72635e316b">
            <realisation xsi:type="hmi:ConfirmDialog">
              <hmi:title xsi:type="StringLiteral"/>
              <hmi:message xsi:type="StringLiteral" value="Hello World!"/>
              <hmi:messageType xsi:type="hmi:MessageTypeLiteral" value="INFO"/>
            </realisation>
          </action>
        </flow>
      </realisation>
    </procedure>
  </procedures>
</otx>
```

XML (according to ISO 13209)

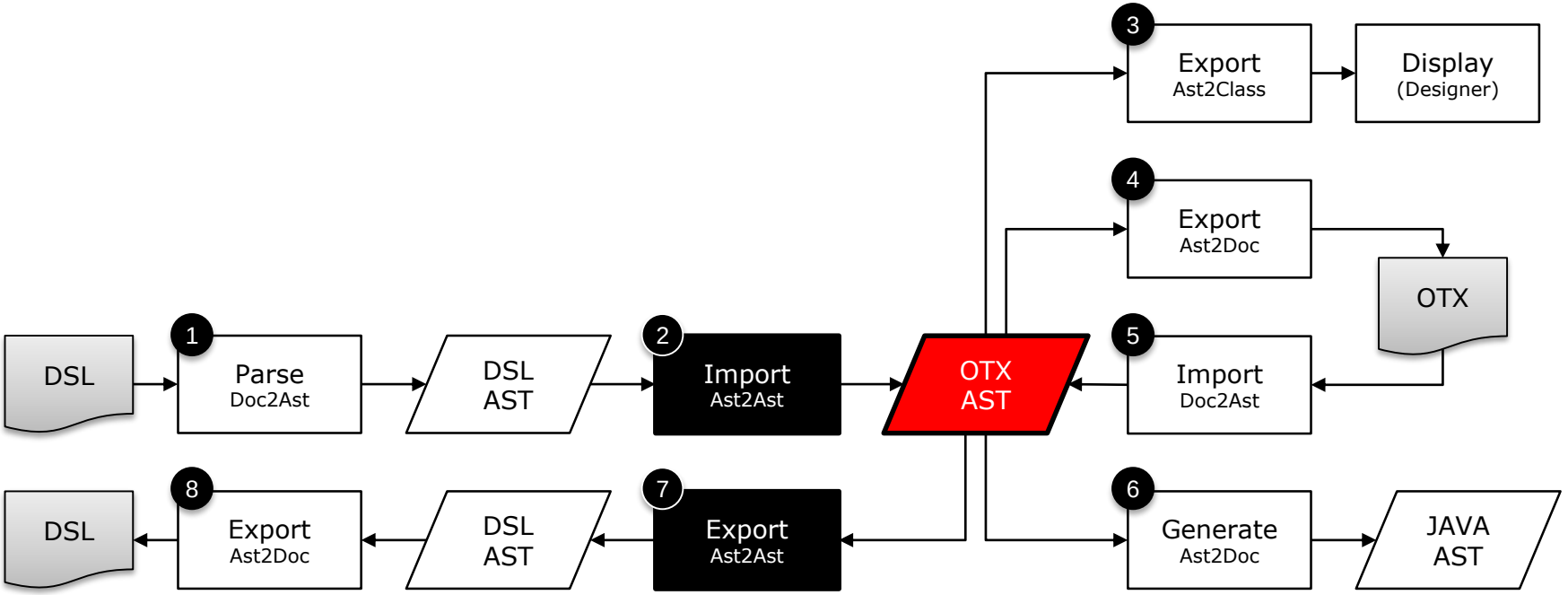
```
[Name, MyFirstDocument]
[Version, 1.0.0.0]
[TimeStamp, 2014-09-08T12:41:38.763+02:00]
namespace NewDefaultPackage1
{
    public procedure main()
    {
        Hmi.ConfirmDialog("Hello World!", NULL, @MessageTypes:INFO, NULL);
    }
}
```

OTL* (**O**pen **T**est sequence **L**anguage)



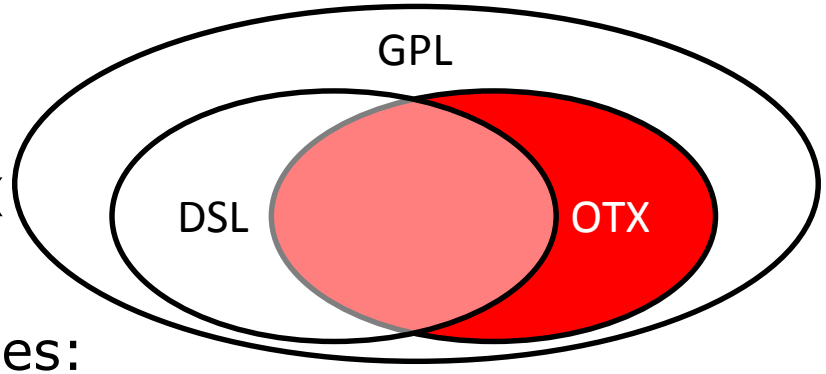
* Developed by EMOTIVE

General architecture of translating DSL → OTX → DSL



DSL = Domain Specific Language
AST = Abstract Syntax Tree = Data Model

Basic Translation Problem DSL/OTX



Different Data Models DSL/OTX

Three possible main translation types:

- One element in DSL can match to exactly one element in OTX
- One element in DSL can match to more than one elements in OTX
- One element in DSL cannot match to one or more elements in OTX → **OTX Extension needed**

In practice, all three types will occur! Therefore, the back translation will result in a different DSL.

Basic Translation Problem Samples DSL1 → OTX → DSL1

An **unsigned integer variable in a PLUTO** procedure is translated to a **signed integer variable in OTX** (64 Bit). The translator, when obtaining PLUTO code back from OTX, will then generate a signed integer variable because it cannot know that the original was unsigned.

A **SPELL** statement containing a **bit shift expression** is translated to a set of OTX statements using the **OTX bit shift statement**, temporal variables and **ByteField** values. The translator from OTX code back to SPELL will keep the multiple statements and may use a different type for ByteField.

A **Watchdog in a PLUTO** procedure is translated to multiple checking statements interposed between the main procedure statements (one translation choice suggested in the project). The translator from OTX code back to PLUTO will keep the statements instead of creating a watchdog procedure.

Basic Translation Problem Samples DSL1 → OTX → DSL2

An **unsigned integer in a PLUTO** procedure is translated to a **signed integer in OTX** and then to an **unlimited length signed integer in SPELL**.

If SPELL is translated to OTX by extending OTX with a **new type Variant** instead of using type inference, then when translating from OTX to PLUTO the same problem appears again (translating from a dynamic to an static language) but this time **PLUTO cannot be extended** in the same way.

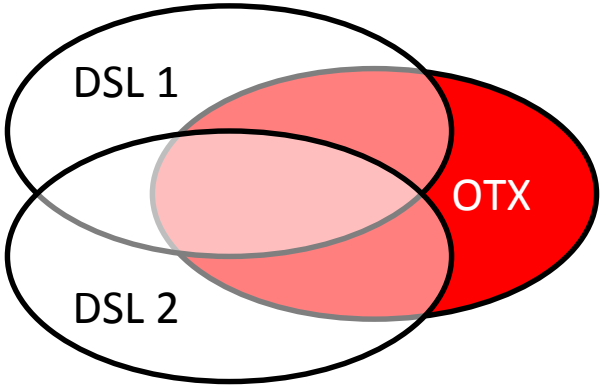
A SPELL bit shift expression is translated to a new extended bit shift expression in OTX including a bit shift statement and ByteField values, and then to multiple statements in PLUTO using a multiplication (or division) by a power of two, because PLUTO does not have bit shift operators.

A procedure with parallel steps and **watchdogs in PLUTO** is translated to OTX, losing at least some behavior for the watchdogs. This procedure **cannot be translated to SPELL** because it does not have constructs for in-procedure concurrency.

Translation Problem Conclusion

The main problem is the different data model between each DSL included OTX!

- Differences in basic types and functions
- Differences in control flow constructs
- Differences in the type system



But **with a certain effort it is possible to translate a DSL to OTX** and also back to the original DSL

A translation from **DSL 1 to OTX and back to DSL 2** seems to be **impossible** without an huge effort

Not all statements OTX can directly translated into a related OTX statement (Action or Term)

OTX have to extent to new Actions and Terms

Translation back from OTX to DSL results in a different DSL code

All problems above are **general problems** and not related to OTX itself!

Translation/Generation of executable Code (execution language – Java)

For the execution the OTX documents have to be translated into an executable language → called Code Generation

The generated target code can be compiled and executed

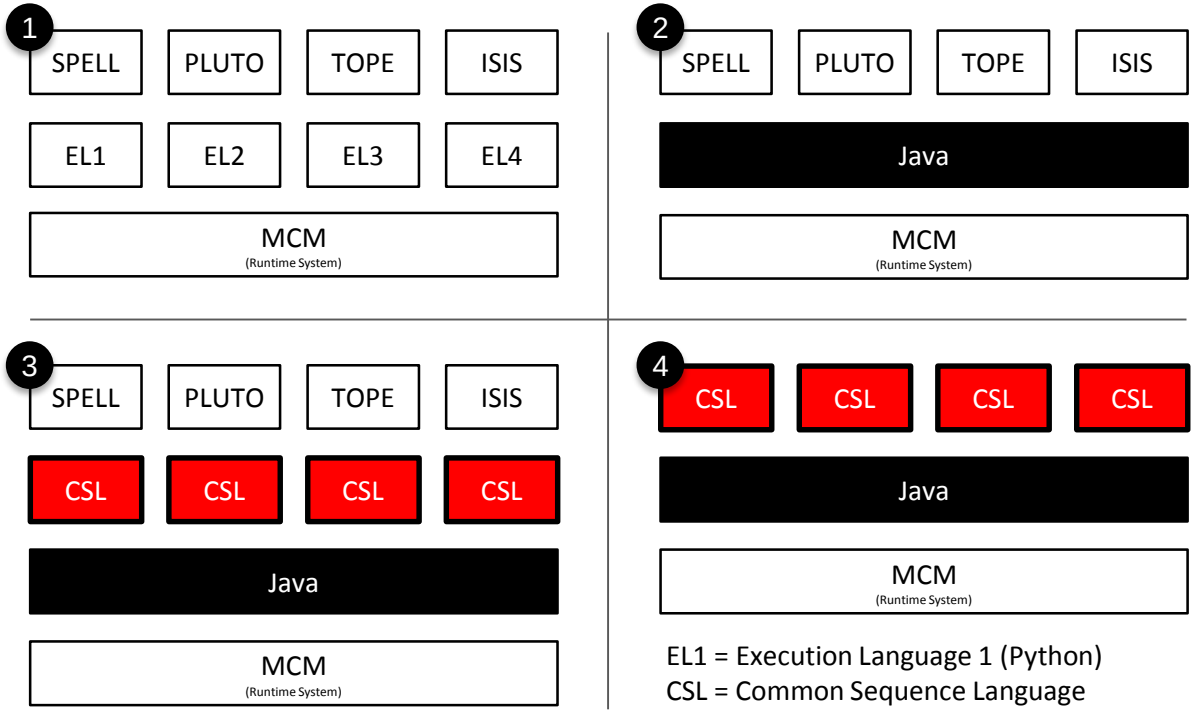
In most cases the target code should be a GPL like Java, C# or C++

ESA decides for Java

Comparing the DSL/OTX translation problem, the code generation OTX to GPL

- expected no blocking problems
- is easy to implement and
- independent from the target language

Main approaches comparison



Criteria	Original (1)	DSL + Java (2)	DSL + CSL + Java (3)	CSL + Java (4)
Binary Compatibility	Bad	Very Good	Very Good	Very Good
Debugging at abstract Level	Good	Good	Good	Very Good
Reusability & Exchangeability	Very Bad	Bad	Good	Very Good
Process Safety	Very Bad	Normal	Normal	Very Good

Comparison table for DSL candidate for approach 4

Criteria	One existing DSL	Java	OTX
Level of Standardization	N/A	Good	Very Good
Independency	Good	Very Good	Good
Reusability	Normal	Normal	Very Good
Exchangeability	Good	Good	Very Good
Extensibility	Normal	Very Good	Good
Scalability	Normal	Very Good	Good
Usability	Normal	Bad	Very Good
Separation of Test Logic and Runtime	Normal	Bad	Very Good
Completeness	Normal	Very Good	Normal
Long-term Maintainability	N/A	Bad	Very Good
Data Driven	Bad	Bad	Very Good
Process Safety	Very Good	N/A	Very Good
Domain Specific Validity	Bad	Bad	Very Good
Change Effort	Good	Good	Normal
Productivity	Good	Bad	Good

Use of OTX as a real exchange format

This approach use OTX as the core language format. Due to the fact, that the execution language shall be common to all EGS-CC systems, it offers this possibility for a clear and straight process.

1. The User delivers the DSL procedure, which is translated into OTX but no translation back to the DSL. The executable language will be generated from OTX.

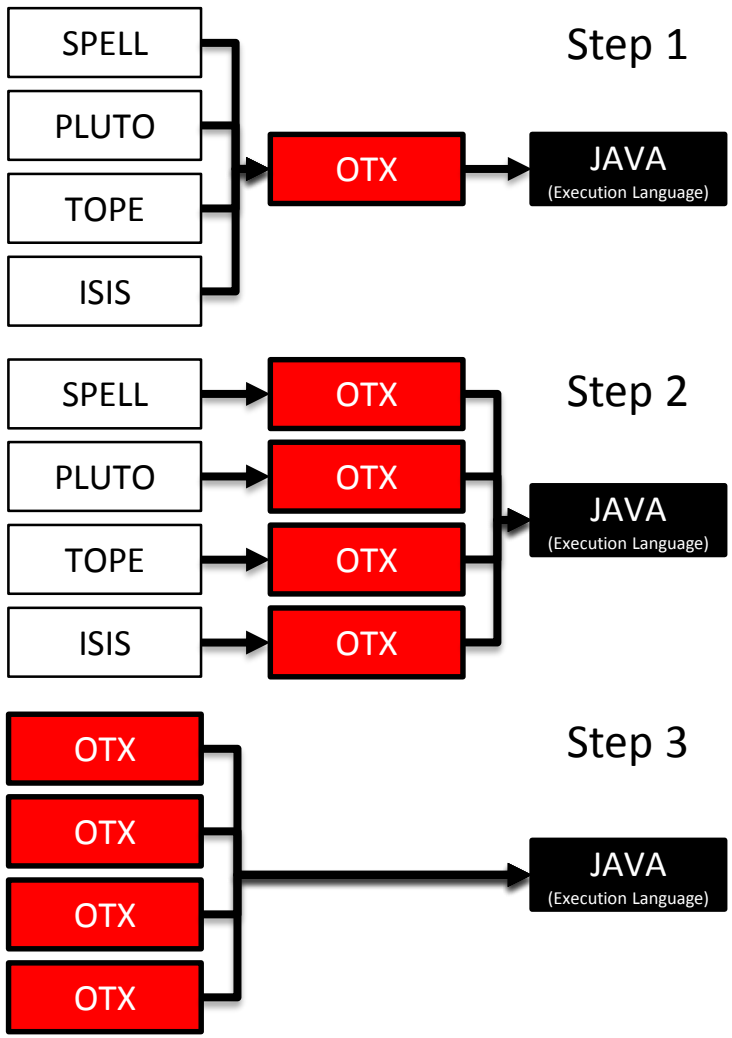
   For a DSL user, nothing has been changed. The System Integrator has to perform the work to translate each DSL procedure into OTX. The System Integrator remains responsible for the results as is already the case. Nevertheless, the solution is better than the current, because the interface is well-defined and the quality of the translated OTX code can be verified.

2. The responsibility is transferred to the DSL User. The DSL user should deliver an already proven OTX code. The responsibility for the System Integrator decreases and the overall quality increases.

3. The DSL user has changed its process to OTX. Now he is also able to share code with other DSL users. OTX is now a real exchange language.

Main **advantages**:

- Forward-looking process
- Interfaces and responsibilities are clearly defined
- Seamless migration from old to new is possible
- Validation and process safe



EGS-CC PROCEDURE EXCHANGE WITH OTX

Support to the EGS-CC

Support to EGS-CC

There is no impact caused by the use of OTX in the requirements of the EGS-CC procedure development environment. Reason: OTX defines only the language itself and not its development or execution environment.

Some possible conflicts are more about the language itself than about the development environment:

- *EGSCC-PREP-REQ-010250 Automation procedures language* requires that the language used to write procedures shall be compatible with the Space Engineering

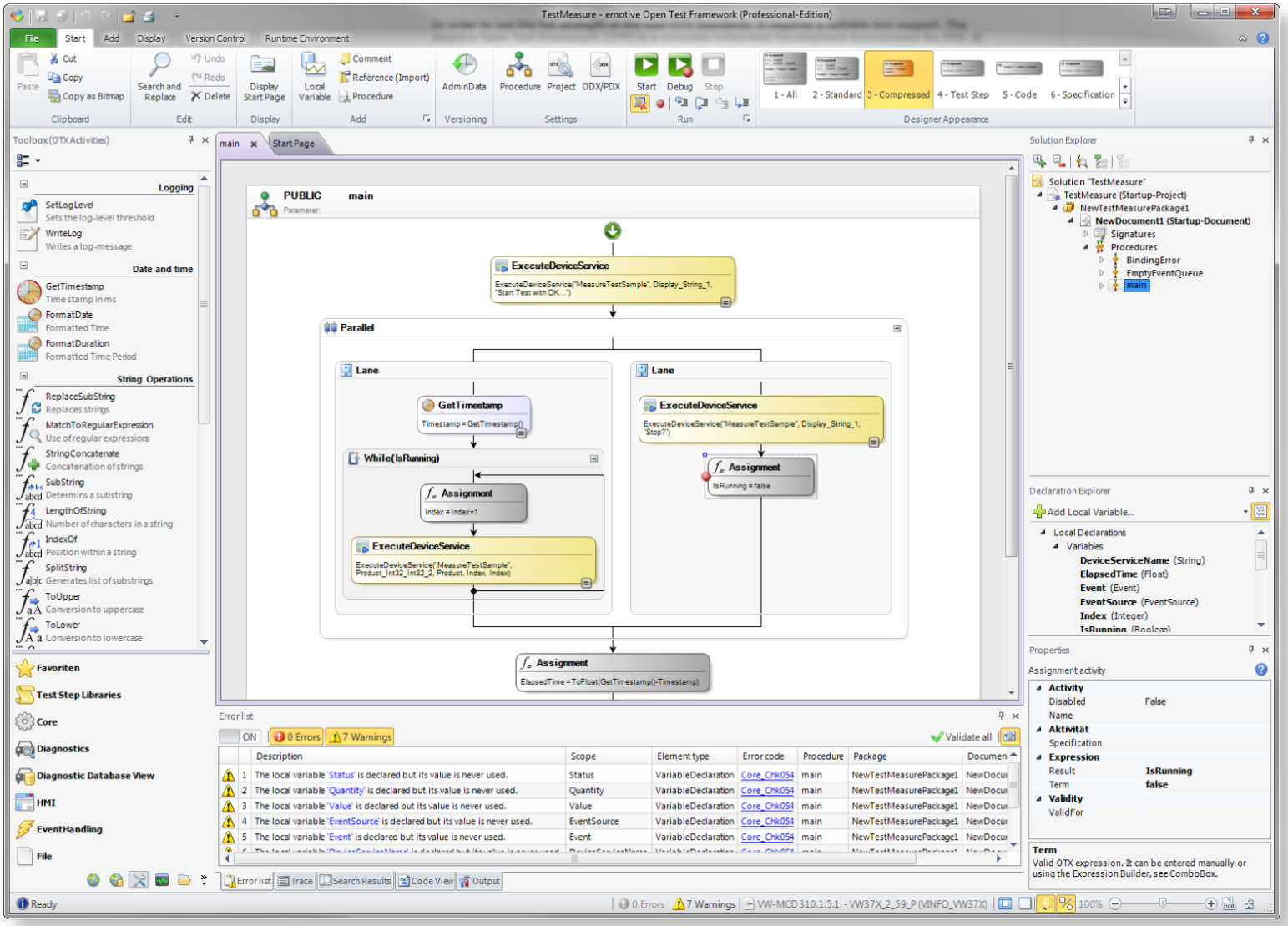
  In particular, OTX does not directly support the data types unsigned integer, time, duration, multidimensional arrays or arbitrary structured data, and the units of measure provided by OTX do not fit exactly in what is specified by Space Engineering. But OTX can be extended for such types.

- *EGSCC-PREP-REQ-010270 Automation procedures language extension* requires that the language provides global variables. This seems to be in conflict with the OTX requirement "No global variables with global scope". Note: Global variables are possible in OTX, but they shall have the visibility *private*. This means, that they can only use within and document and not outside.

EGS-CC PROCEDURE EXCHANGE WITH OTX

Open Test Framework

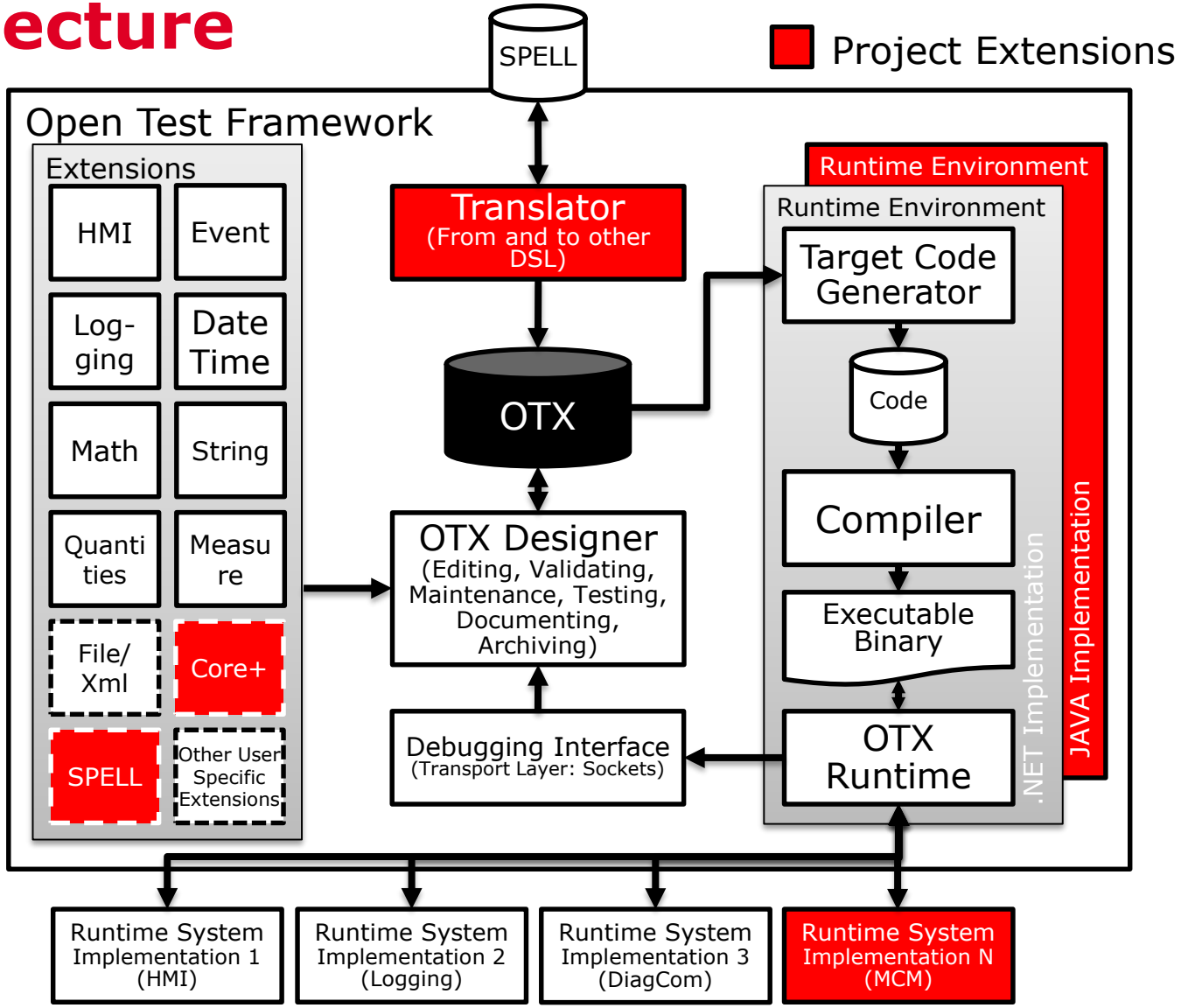
Open Test Framework (OTF)



EGS-CC PROCEDURE EXCHANGE WITH OTX

Proof of Concept (PoC)

Proof of Concept (PoC) General Architecture



Proof of Concept (PoC) Main Elements

- Translator from DSL to OTX (*Importer*)
- Translator from OTX to DSL (*Exporter*)
- Translator from the OTX to the Executable language – JAVA
- Interfaces with the EGS-CC Emulator (MCM) applied to:
  - The Space libraries (database) interface in OTX, for adapting the Executable Language to the Space Domain specific needs
  - The interface to the MIB database for specify the variable types used in the provided operational procedure and the verification of the addressed data and displays
  - The interface and mechanisms for debugging execution of the translated operational procedure in the Java runtime environment. For simplicity and cost reduction, Java shall not be directly generated. The currently generated C# code is transformed into Java. Please note that no matter how the code is generated, the result is the same.
- EGS-CC (MCM) emulator – a mock-up application emulating the actual approach for the EGS-CC system

EGS-CC PROCEDURE EXCHANGE WITH OTX

Conclusions

Conclusions

1. It has been demonstrated, that OTX is fully suitable to specify test procedures in the Space domain
2. Translation from SPELL to OTX is possible with 100% functionality
3. Translation back from OTX to SPELL is possible but it results in a different, but semantically identical SPELL code (It's a general translation phenomena)
4. Connection between OTX and MCM/SSM has been successfully demonstrated
5. Due to 3. it has been decided not to recommend OTX as exchange format for the EGS-CC if the stakeholders keep procedures in their DSLs
6. We recommend to use OTX with Space related Extensions as an exchange format and DSL

Thank you

EGS-CC PROCEDURE EXCHANGE WITH OTX

e-mails:
R. Sánchez-Beato Fernández <rsbf@gmv.com>
Jörg Supke <joerg.supke@emotive.de>

Further Activities

1. Definition, coordination and implementation of new Space relevant OTX Extensions
2. Implementation of complete OTX-Runtime in Java
3. Fully support of MCM-Runtime-System from OTX
4. Proof of concept in a real Space environment
5. Setup of an OTX driven process for test procedures in Space industry

EGS-CC PROCEDURE EXCHANGE WITH OTX

Demo

Proof of Concept (PoC): Demo