

# TASTE

Using model-driven code generation  
for safety-critical applications

Thanassis Tsiodras

*European Space Agency  
ESTEC/TEC-SWE*

# A bit about me



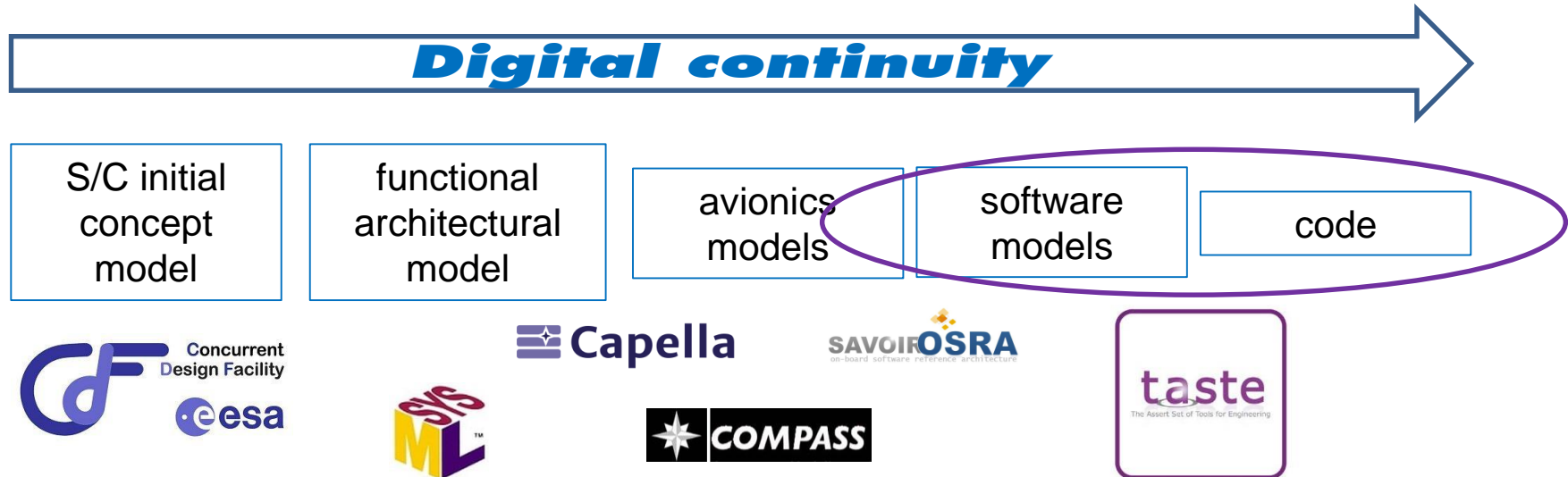
- 2 decades in 2 minutes
- PhD from NTUA in 1999; founded Semantix in 2001 and was its Lead Engineer for 13 years.
- **Discovered MDE “in the field”** - put it to use in our product lines, with spectacular results
- Became a part of the team that built the core TASTE tools
- Joined ESA in January 2016.



# In context: MeTRiD and ESA



- Methods and Tools for Rigorous System Design
  - ESA promotes and advocates Model Based System Engineering at all levels
  - Model Based *System*; Model Based *Avionics*; Model Based *Software*
  - ESA demonstrator on Digital Continuity



# What is TASTE?



- A set of tools and technologies that target the creation of **complete applications for safety-critical systems** using **MDE and code generation**
- A long-running effort in ESA to **showcase MDE** and its immense benefits **to the domain of space SW**
- An open-source **testbed of ideas** to improve the quality of space SW



# Rule No 1: The most maintainable code...



- ...is the one **you don't write at all**
- Don't write code – have machines **write it for you**
- To do that, **create models** representing the logic
- ...and have **code generators write the code** for you



# Rule No 2 – Reuse, don't “NiH”



- When creating said models, **don't reinvent the wheel**
- NiH syndrome: *“I can do it from scratch and do it better!”*  
(No, most of the time **you can't**)
- Study the pre-existing tech – and build on top of it **unless you have to**
- Use the appropriate modelling languages for each domain



# Rule No 3 – Keep it open



- When choosing modelling languages, you also need to **address tooling** concerns
- **Always prefer open-source tooling** whenever possible
  - avoid vendor lock-in
- Make sure the tools **store the models in interoperable form** - ideally, in the textual syntax of the modelling language itself
- Develop the toolchain **in the open**, involve **as many outside partners as possible** – the safety-critical domain is targeted by many.



# Rule No 4 – Keep it real

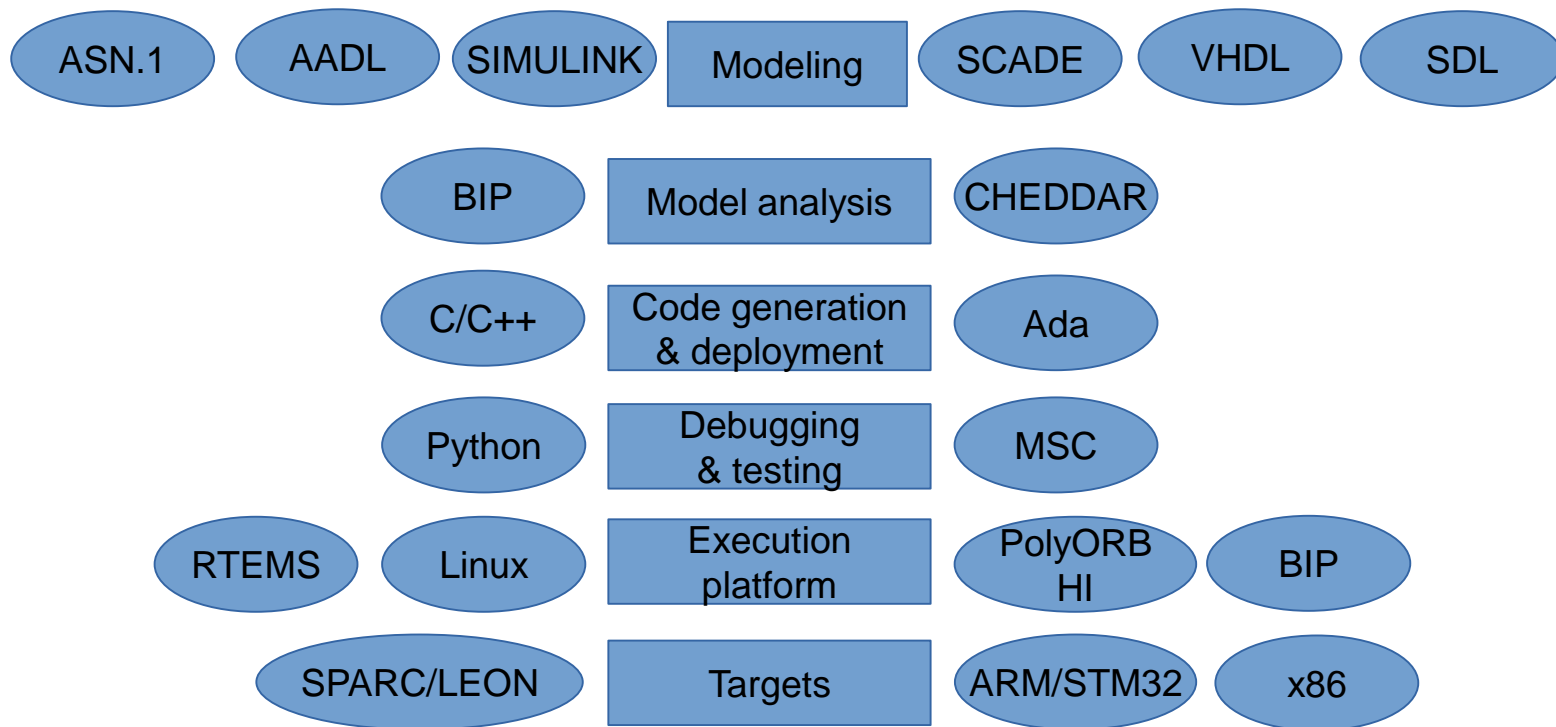


- The truth: there are so many research results out there that **are basically unusable** in the real world
- It is one thing to invent technology – making it usable in **everyday workflows** is quite another
- If we are to “tip the scales” and make MDE a part of the safety critical domain, **we need to create a toolchain that works**
- Address the “mundane”: How to make a toolchain that works **everywhere**? That can be used by **everyone**? That **auto-updates** itself?





# A subset of the technologies supported



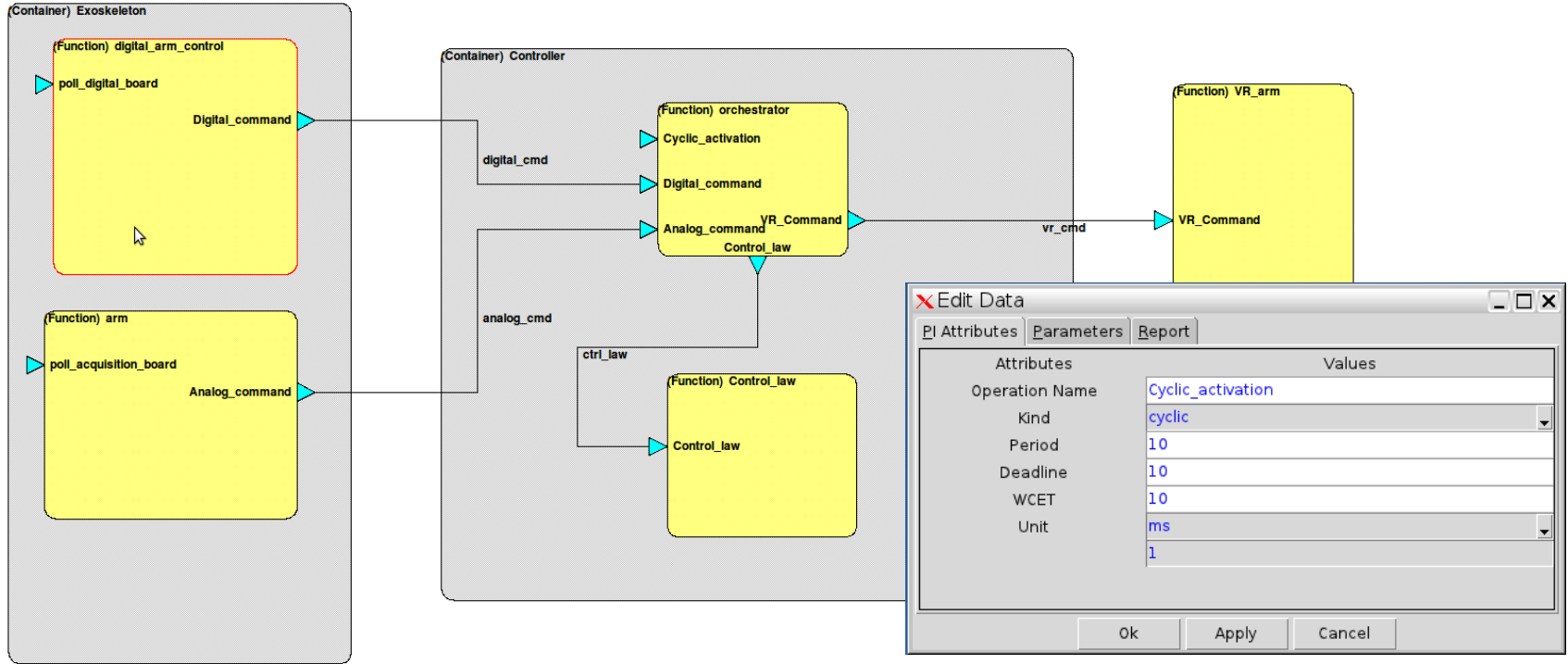
# How to put it all together?

- Manually
  - Executive summary: “while True: Hack()”
  - Interface/spec changes? Costly changes...
  - Mismatches in the specs? Bugs that slipped through the cracks?  
*“Oh well, let’s patch in orbit”*
- Stick to one vendor - *“one tool can rule them all”*
  - Actually, no - it can’t.
  - There’s no silver bullet that addresses all domains and all their requirements.
- Develop each functional block with **the appropriate tool for the job**
  - Use TASTE to perfectly **“glue”** all the pieces together - automatically.

# The mile-high view of the workflow

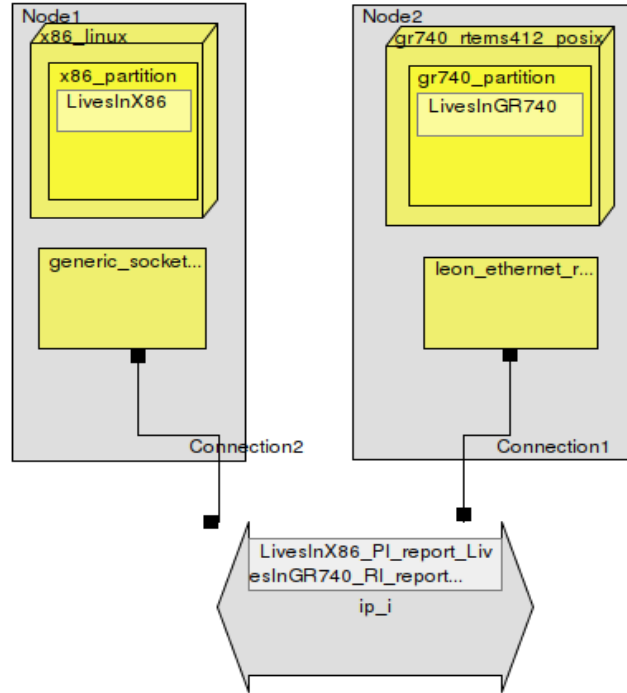
- Describe your interfaces' **parameter types** in **ASN.1**
- Describe the **interfaces** and the **deployment** of the functional blocks in **AADL**
- TASTE will then generate **skeletons** for your functional blocks
- Fill-in the skeletons and **verify the implementation at model level**
- TASTE finally builds the system automatically.

# Interface View



# Deployment View

- Choose your **deployment nodes**
- **Bind** your functional blocks inside them
- **Connect** them via devices and buses, and configure what each message will use to “travel across”



# DataView – ASN.1, an ITU standard



- Let's look at one of the TASTE technologies in depth: the data modelling.
- ASN.1 is a good standard – one that **works so well you forget about it**  
*(it fades into the background)*
- Your phone uses it every minute, your bank's ATM every time you take out some cash, your browser every time you visit an HTTPS site...
- So describe your interfaces' messages in ASN.1 – and **get optimal encoders and decoders in C or Ada** by ASN1SCC ; our Space Certifiable Compiler.
- But what does that mean, in practice?



# ASN1SCC in action

- Write the description of your messages in an abstract form
- That is, you **don't** specify “*int16\_t*”, “*uint32\_t*”, little endian, big endian...
- These details are taken care of by the **ASN.1 compiler**...

```
MyInt ::= INTEGER (0 .. 20)
My2ndInt ::= MyInt ( 1 .. 18)
AType ::= SEQUENCE {
    blArray      SEQUENCE (SIZE(10)) OF BOOLEAN
}
My2ndAType ::= AType
TypeEnumerated ::= ENUMERATED {
    red(0),
    green(1),
    blue(2)
}
My2ndEnumerated ::= TypeEnumerated
TypeNested ::= SEQUENCE {
    intVal      INTEGER(0..10),
    int2Val     INTEGER(-10..10),
    int3Val     MyInt (10..12),
    intArray    SEQUENCE (SIZE (10)) OF INTEGER (0..3),
    realArray   SEQUENCE (SIZE (10)) OF REAL (0.1 .. 3.14),
    ...
}
```

# ASN1SCC in action

- ...that processes your grammar
- creates an **Abstract Syntax Tree** (AST) describing all the type information, and then uses it;
- ...to generate encoders and decoders for each one of the message types

```
taste@tastevm ~/tmp/demo
$ l
total 12
drwxr-xr-x 2 taste taste 4096 Apr  7 08:34 ./
drwxr-xr-x 3 taste taste 4096 Apr  7 08:33 ../
-rw-r--r-- 1 taste taste 1505 Apr  7 08:34 DataTypes.asn

taste@tastevm ~/tmp/demo
$ asn1.exe -c -uPER -atc DataTypes.asn

taste@tastevm ~/tmp/demo
$ l
total 524
drwxr-xr-x 2 taste taste 4096 Apr  7 08:35 ./
drwxr-xr-x 3 taste taste 4096 Apr  7 08:33 ../
-rw-r--r-- 1 taste taste 32632 Apr  7 08:35 acn.c
-rw-r--r-- 1 taste taste 24626 Apr  7 08:35 asn1crt.c
-rw-r--r-- 1 taste taste 19988 Apr  7 08:35 asn1crt.h
-rw-r--r-- 1 taste taste 1505 Apr  7 08:34 DataTypes.asn
-rw-r--r-- 1 taste taste 42237 Apr  7 08:35 datatypes_auto_tcs.c
-rw-r--r-- 1 taste taste 3187 Apr  7 08:35 datatypes_auto_tcs.h
-rw-r--r-- 1 taste taste 63045 Apr  7 08:35 DataTypes.c
-rw-r--r-- 1 taste taste 34896 Apr  7 08:35 DataTypes.h
-rw-r--r-- 1 taste taste 1104 Apr  7 08:35 mainprogram.c
-rw-r--r-- 1 taste taste 1310 Apr  7 08:35 Makefile
-rw-r--r-- 1 taste taste 48826 Apr  7 08:35 real.c
-rw-r--r-- 1 taste taste 226335 Apr  7 08:35 testsuite.c
-rw-r--r-- 1 taste taste 606 Apr  7 08:35 testsuite.h
```



- The encoders and decoders look like this...

```
flag TypeNested_Encode(const TypeNested* val, BitStream* pBitStrm, int* pErrCode,  
flag TypeNested_Decode(TypeNested* pVal, BitStream* pBitStrm, int* pErrCode);
```

- They will **verify all your message constraints**, and report specific errors...

```
#define ERR_MyInt          1001 /*(0 .. 20)*/
```

- The amount of memory necessary to **statically reserve enough space** for all possible configurations of your message types is also provided:

```
#define TypeNested_REQUIRED_BYTES_FOR_ENCODING    389  
#define TypeNested_REQUIRED_BITS_FOR_ENCODING    3110
```

- ...as are the **automatically generated test cases** - that exercise these encoders and decoders at 100% statement and branch coverage:

# ASN1SCC in action



- ...so you get all necessary combinations of values put in your fields, the messages are encoded to a stream and decoded back, the content is then checked to make sure **it remained as-is through the round-trip** – and in so doing, all lines of the encoders and decoders are also **fully exercised**.
- That's a lot of work! That you'd otherwise be forced to do yourself.

```
ttsiod@mbair ~/tmp/demo
$ make coverage
make && ./mainprogram && \
    gcov datatypessimulink.c
make[1]: Entering directory '/home/ttsiod/tmp/demo'
gcc -c -g -Wall -Werror -Wextra -Wuninitialized -Wcast-qual -D_DEBUG -I . -O0 -fprofile-arcs -ftest-coverage -o acn.o acn.c
gcc -c -g -Wall -Werror -Wextra -Wuninitialized -Wcast-qual -D_DEBUG -I . -O0 -fprofile-arcs -ftest-coverage -o asn1crt.o asn1crt.c
gcc -c -g -Wall -Werror -Wextra -Wuninitialized -Wcast-qual -D_DEBUG -I . -O0 -fprofile-arcs -ftest-coverage -o datatypessimulink.o datatypessimulink.c
gcc -c -g -Wall -Werror -Wextra -Wuninitialized -Wcast-qual -D_DEBUG -I . -O0 -fprofile-arcs -ftest-coverage -o DataTypesSimulink.o DataTypesSimulink.c
gcc -c -g -Wall -Werror -Wextra -Wuninitialized -Wcast-qual -D_DEBUG -I . -O0 -fprofile-arcs -ftest-coverage -o mainprogram.o mainprogram.c
gcc -c -g -Wall -Werror -Wextra -Wuninitialized -Wcast-qual -D_DEBUG -I . -O0 -fprofile-arcs -ftest-coverage -o real.o real.c
gcc -c -g -Wall -Werror -Wextra -Wuninitialized -Wcast-qual -D_DEBUG -I . -O0 -fprofile-arcs -ftest-coverage -o teststest.o teststest.c
gcc -g -o mainprogram acn.o asn1crt.o datatypessimulink.o DataTypesSimulink.o mainprogram.o real.o teststest.o
profile-arcs
make[1]: Leaving directory '/home/ttsiod/tmp/demo'
All test cases (109) run successfully.
datatypessimulink.gcno:cannot open notes file

ttsiod@mbair ~/tmp/demo
$ gcov DataTypesSimulink.c
File 'DataTypesSimulink.c'
Lines executed:100.00% of 1434
Creating 'DataTypesSimulink.c.gcov'
```



- When targeting Ada, the generated code includes SPARK contracts:

```
20 function GetZeroBasedCharIndex
19   (CharToSearch : Character;
18    AllowedCharSet : in String) return Integer with
17   Pre => AllowedCharSet'Last >= AllowedCharSet'First and
16   AllowedCharSet'Last <= Integer'Last - 1,
15   Post =>
14   (GetZeroBasedCharIndex'Result >= 0 and
13    GetZeroBasedCharIndex'Result <=
12    AllowedCharSet'Last - AllowedCharSet'First);
11
10 function CharacterPos (C : Character) return Integer with
9   Post => (CharacterPos'Result >= 0 and CharacterPos'Result <= 127);
8
7 procedure BitStream_AppendBit
6   (S : in out BitArray;
5    I : in out Natural;
4    BitVal : in BIT) with
3   Depends => (I => I, S => (S, BitVal, I)),
2   Pre => I >= S'First - 1 and I <= S'Last - 1,
1   Post => I = I'Old + 1;
0
```

...so by using the proper tools you can *prove* that your usage scenarios (i.e. the caller code) will e.g. **never overflow their buffers**

- **Bulletproof** message marshalling
- The compiler is **maintained in the open** ( <https://goo.gl/76Yo7R> )
- ...is already **used in missions** (3 use cases already)
- And we've barely scratched the surface...
- ...because when you have an abstract model, you find **you can do more things**.  
For example, space standards require formal documentation...

... parts of which you can **generate automatically**: e.g. from this grammar...

```
TASTE-Dataview DEFINITIONS ::=
BEGIN

TASTE-Boolean ::= BOOLEAN

Telecommand ::= SEQUENCE {
  a INTEGER (0..7),
  b TASTE-Boolean
}

Telecommand [] {
  a [size 12, encoding pos-int],
  b [true-value '00100'B]
}
```

# ASN1SCC in action

...you get this Interface Control Document (ICD)...

## Module TASTE-Dataview

Defined in: *DataView.asn, DataView.acn.*

TASTE-Boolean ( <i>anonymous</i> BOOLEAN) <u>ASN.1</u> Min: 1 bytes Max: 1 bytes		
Constraints	Min Length (bits)	Max Length (bits)
N.A.	1	1

Telecommand (SEQUENCE) <u>ASN.1</u> <u>ACN</u> Min: 3 bytes Max: 3 bytes							
No	Field	Comment	Present	Type	Constraint	Min Bits	Max Bits
1	a		always	INTEGER	(0 .. 7)	12	12
2	b		always	<u>TASTE-Boolean</u>		5	5

...correct by construction - and always in sync with your design.

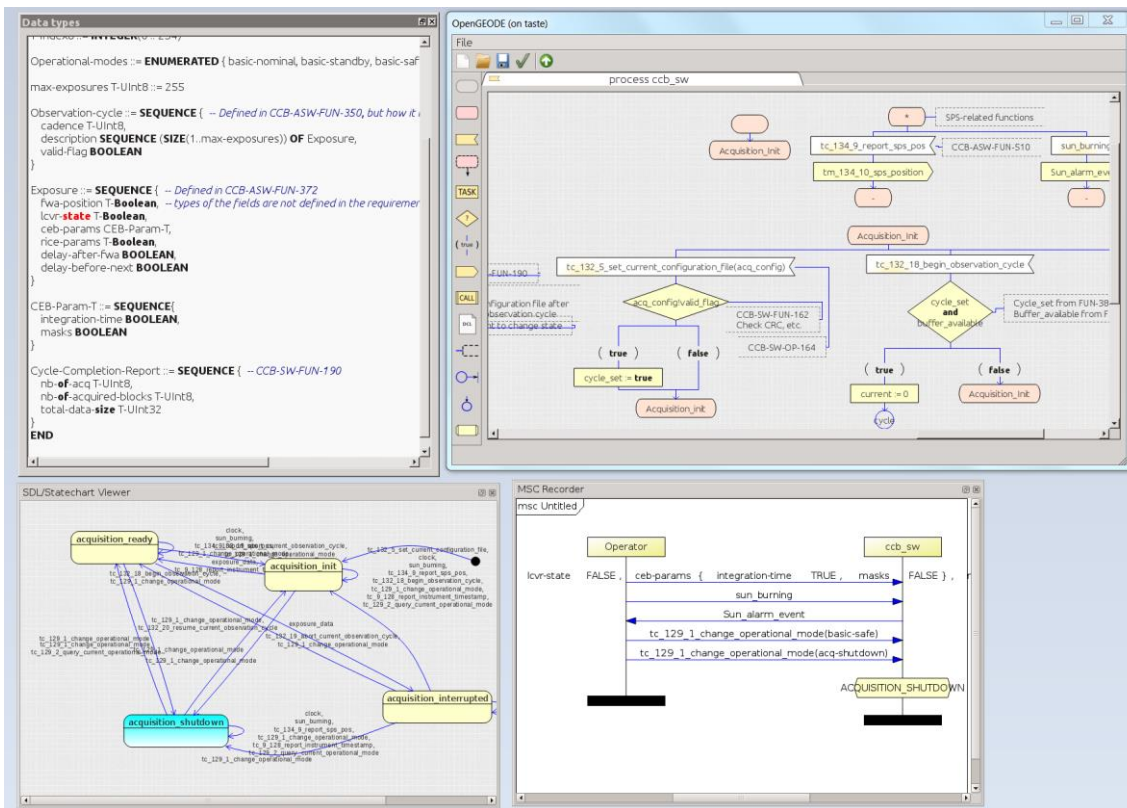
# Models – the power behind TASTE



- Having access to the underlying models, enables many things
- Far more impressive things than just serializing data...
- Models extend to **the behavioural logic** of the functional blocks.
- e.g. if you are using state machines, you can describe them in **SDL**; and TASTE offers OpenGEODE - an open-source (<https://goo.gl/JsXv6Z>) **SDL editor**.



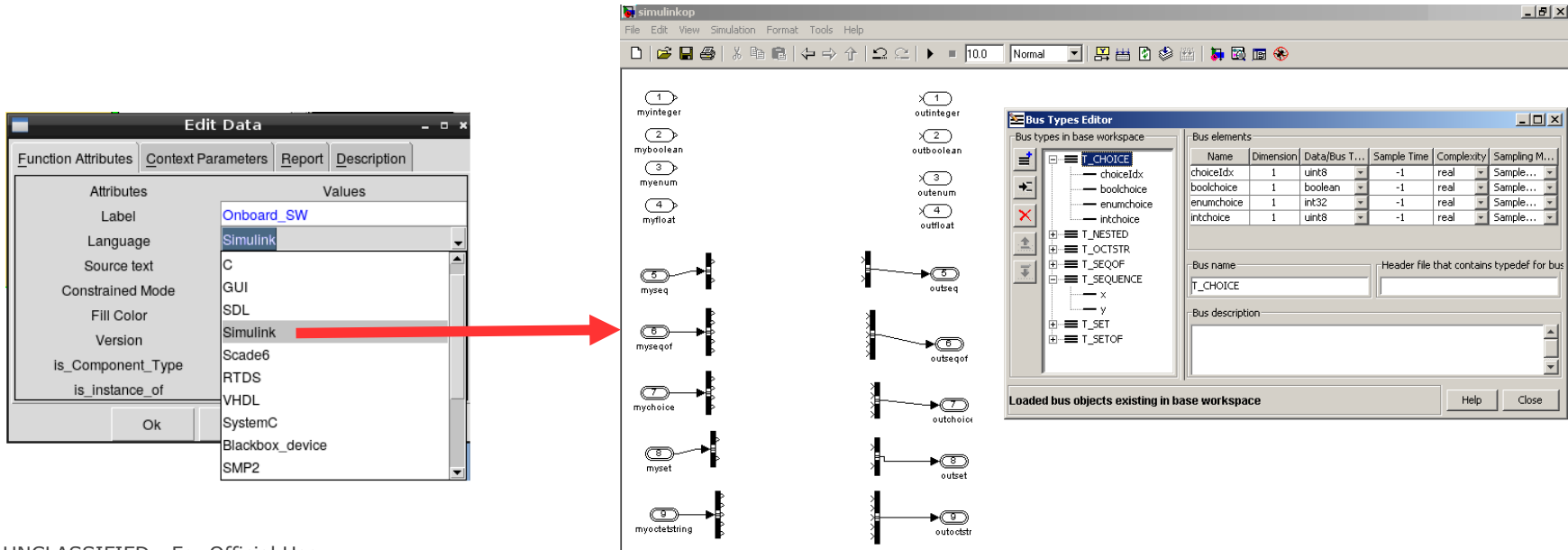
# Models – the power behind TASTE





# “Glueing” functional blocks together

- Or perhaps your Guidance Navigation and Control (GNC) people design and verify with **Simulink** (or SCADE) - and then autogenerate the related code...
- In which case, TASTE can create the skeletons for your design...



The screenshot displays the Simulink environment with three main windows:

- Edit Data:** A dialog box for editing data, showing a list of values including `Onboard_SW`, `Simulink`, `C`, `GUI`, `SDL`, `Scade6`, `RTDS`, `VHDL`, `SystemC`, `Blackbox_device`, and `SMP2`. A red arrow points from the `Simulink` value to the block diagram.
- Block Diagram:** A Simulink diagram showing two columns of blocks. The left column contains blocks labeled `myinteger`, `myboolean`, `myenum`, `myfloat`, `myseq`, `myseqof`, `mychoice`, `myset`, and `myoctetstring`. The right column contains blocks labeled `outinteger`, `outboolean`, `outenum`, `outfloat`, `outseq`, `outseqof`, `outchoice`, `outset`, and `outoctstr`. Each block is connected to its corresponding output block.
- Bus Types Editor:** A window showing the bus types in the base workspace. It displays a tree view of bus types and a table of bus elements.

Name	Dimension	Data/Bus/T...	Sample Time	Complexity	Sampling M...
choiceIdx	1	uint8	-1	real	Sample...
boolchoice	1	boolean	-1	real	Sample...
enumchoice	1	int32	-1	real	Sample...
intchoice	1	uint8	-1	real	Sample...

# “Glueing” functional blocks together



- ...and can also create the code that will **translate the data at run-time** between the C structures generated by Simulink’s Embedded Coder and the ones generated by ASN1SCC
- And since TASTE can do this for quite a number of modelling tools and languages, this means your Simulink / SCADE block can now “speak” to the outside world **at zero integration cost**
- Your types, your interfaces, your integration code, they all “magically” become available – because there’s an underlying ASN.1 model describing the data, and an underlying AADL model describing the interfaces
- MDE power – use the best tool for each job, and then “glue” it all together.



# Space Robotics (video)

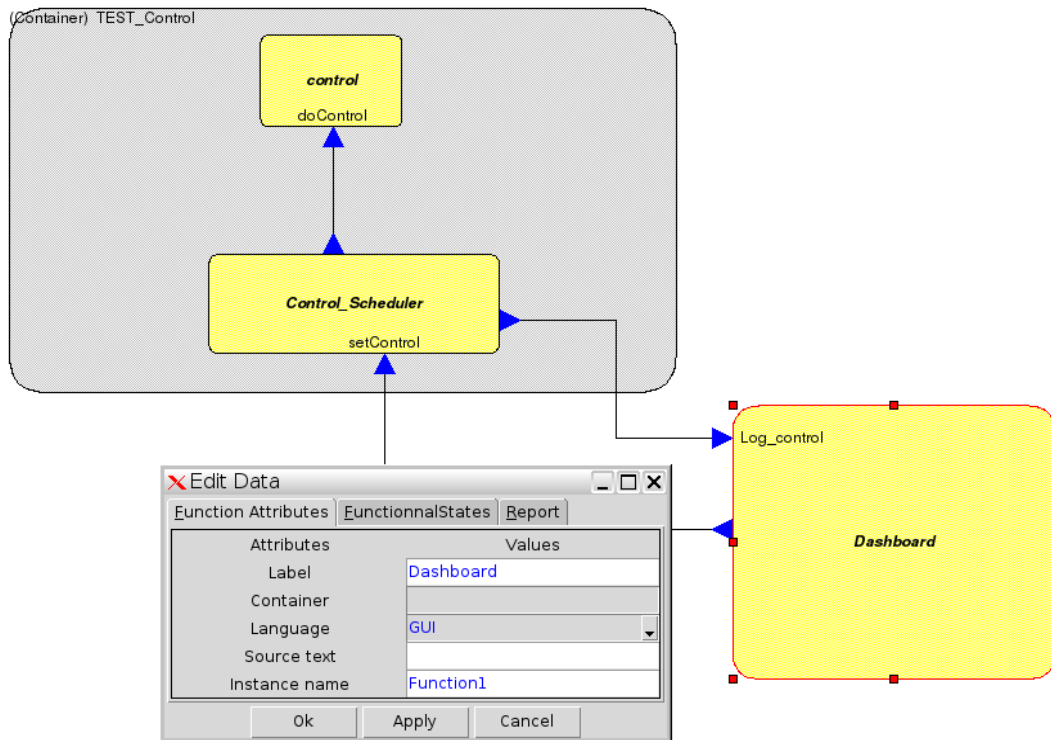
Enough theory... here's a TASTE-y Space Rover:

*(in this video, the Robotics Division is using TASTE to control EXOTER)*



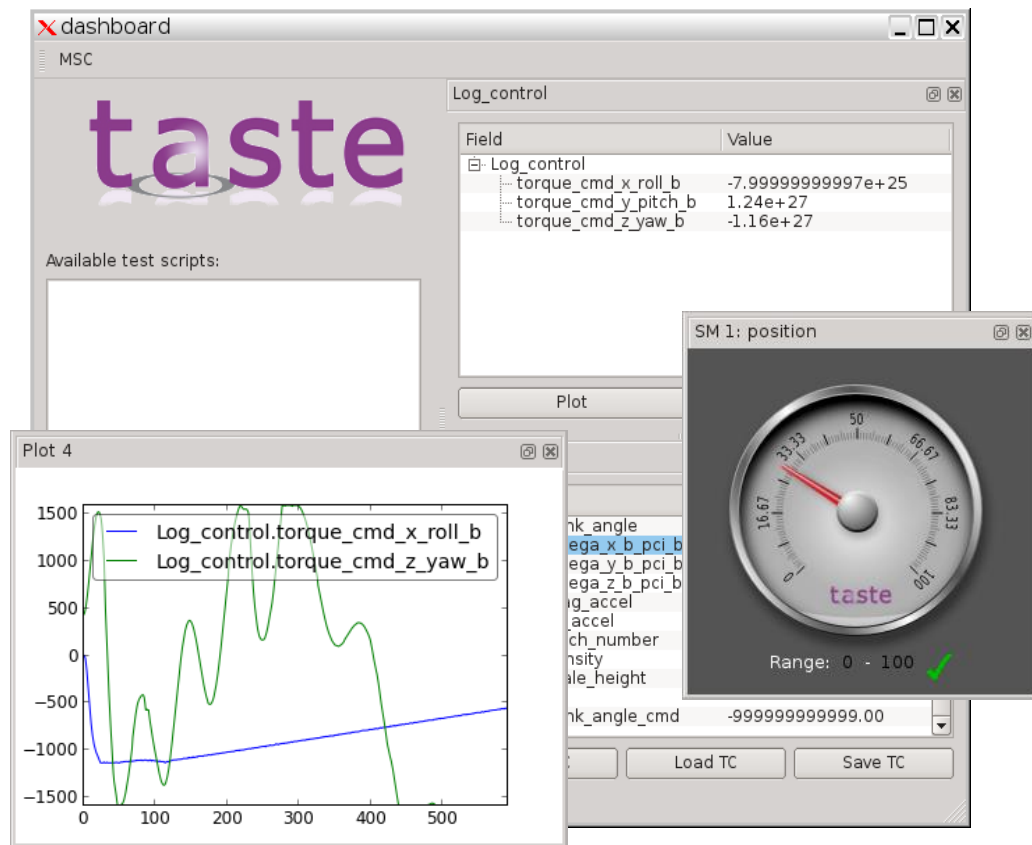
# Graphical User Interfaces

- If you specify “GUI” as the implementation language of a functional block...



# Graphical User Interfaces

- ...you will then get an automatically generated “Ground” application - that allows you to graphically call all the *Required interfaces* (i.e. fill-in the parameters and invoke the **telecommands**) and also monitor the calls being done to its *Provided interfaces* (i.e. the **telemetry** – as well as plot/monitor any field inside it in real-time).



# Graphical User Interfaces



**orchestrator**

MSC SDL

# taste

Internal state

Field	Value
- a	one
fixed	3
elem_0	1
elem_1	2
elem_2	3
anint	479
seq	5
elem_0	0
elem_1	1
elem_2	0
elem_3	1
elem_4	0

telemetry

Field	Value
peek_fixed	3
elem_0	1
elem_1	2
elem_2	3

Simulation bay

Send parameterless telecommands and timers

Paramless\_TC

mytimer timeout

Log

New state: WAIT  
New state: NEXT  
Sent pulse(one)

Reset Undo Redo

SEQ0F SIZE(3)

MSC Recorder

msc Untitled

```
sequenceDiagram
    participant Operator
    participant orchestrator
    Operator->>orchestrator: pulse(one)
    orchestrator-->>Operator: telemetry(one)
    Operator-->>orchestrator: peek_list( { 0, 1, 0, 1, 0 } )
    Operator-->>orchestrator: peek_fixed( { 1, 2, 3 } )
    Note over orchestrator: NEXT
```

SDL/Statechart Viewer

```
stateDiagram-v2
    [*] --> wait
    wait --> next : pulse
    next --> wait : pulse
    wait --> timeout : mytimer
    timeout --> next : Paramless_TC
```





# Graphical User Interfaces (video)

A video recorded during the ESA Open Days last October – where you can see TASTE-y Quadcopters being flown and controlled via TASTE GUIs.

The scenario reproduced is that of the PROBA3 mission – flying in formation, with one copter “shielding” the other from the “sun” (just as the real mission will do, to allow examining the Sun’s corona)



# Graphical User Interfaces



- In so doing, the GUI can also **record the exchanges** in a standard form - an MSC (a Message Sequence Chart). This will include all the exchanged TM/TC information.
- The recorded .msc files can then be processed by a TASTE code generator that **translates them to Python scripts...**
- ...that can be used for automated testing - basically, record a scenario, and run the Python script inside your Continuous Integration (Jenkins, etc) to make sure what you just enacted indeed stays functional during implementation of the system (*easy regression checking*)





# Python and uPython



- Speaking of Python test scripts – not only do we expose all the necessary bindings that allow you to “speak” to a TASTE-generated system at run-time...
- ...we also support **MicroPython** as one of the implementation languages.
- If you mark a functional block as implemented in MicroPython, then a qualified executor is bundled inside the generated binary - and will execute MicroPython bytecode that is compiled from your functional block’s Python code
- Which means you can do your On-Board control procedures with this – and generally speaking, even “**ship**” **bytecode at run-time**



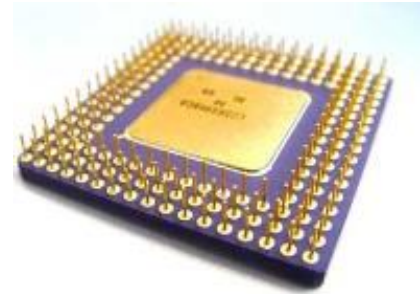
- The environments where TASTE binaries are meant to execute in, need to obey **very strict verification and execution semantics** - they need to be qualified for space flight.
- So in addition to Linux, TASTE specifically targets RTEMS – the OS used (and qualified) in many of our missions
- That is itself an evolving target – which is why the “mundane” part of automatic updates **is very important**: we need to keep track of the RTEMS mainline, build the appropriate cross-compilers for our targets (Leon2, Leon3, GR712RC, GR740) and bundle them in TASTE environments with minimal fuss.



And we do – just execute “*Update-TASTE.sh*”! - Remember: “*keep it real*”.

# Many more features...

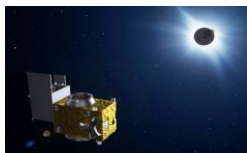
- When writing your test scripts, it would sure be nice if you could save and restore the related messages **inside a relational database**... and indeed you can: We create SQL DDL statements for setting up of your DB, and we also support major DB engines (PostgreSQL, MySQL, etc) via **SQLAlchemy**
- We support **FPGAs** and **VHDL**: when specifying a functional block as a HW block, TASTE will then create both the VHDL skeletons as well as a **complete device driver** that speaks to the synthesized device at run-time - marshalling all input and output parameters both ways.



Watch this hands-on video creating a TASTE-y VHDL design: <https://goo.gl/XYwWRn>

# Real use cases – and not just in space

**CHEOPS:** ASN.1 and ASN1SCC is used by DLR to generate the message encoders and decoders for the application SW.



**PROBA3:** the payload onboard and ground segments make extensive use of the Data Modelling Tools from TASTE, for both code and documentation purposes.

**FBK** uses TASTE in operational projects that are not funded by ESA:

- *Contest* (solar/stirling cogeneration; modeling, deployment)
- *GreenerSys* (single-unit Organic Flow-Batteries; model, deployment)
- *GreenerNet* (grid of OFBs; modeling), etc.



# Are we done? (technical challenges)



- We have **more work** to do.
- The maturity of TASTE's individual pieces **varies**.
- For some of the pieces we're still at the “*first make it work*” stage – we need to move them to the “*make it work right*” and “*make it work fast*” stages.
- More modelling tools, techniques and technologies keep coming up. We want to take advantage of them and merge them in – but remember, we need to “*keep it real*”
- This is **not** an academic exercise! We want TASTE to become the **backbone of our missions**.



# Are we done? (non-technical challenges)



- Very few people **trust** these technologies enough to try them out
- Education on MBSE **is lacking** – nowadays, few new engineers know about it
- Proprietary new languages pop up all the time => **people lose focus**
- Lots of **legacy code and processes** are in place in the industry
- For companies, unproven short-term ROI **is a risk**
- Many decision-making people think of this as a "religious" debate  
...because they don't know *the impact it can have*.





# Questions?

