AIAA SOSTC 2010

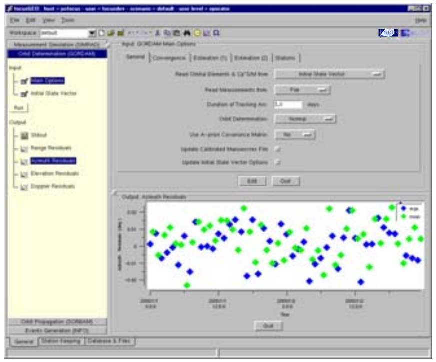
21st century operational procedure automation with Python and *autofly*

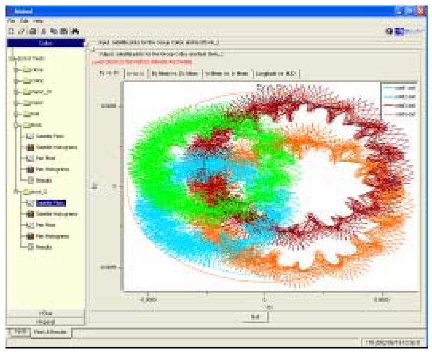
LESSONS LEARNED FROM MIGRATING LEGACY SYSTEMS AND DEVELOPING NEW ONES

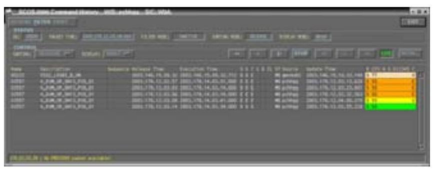
Mike Palsson
Gonzalo Garcia
Thomas Morel

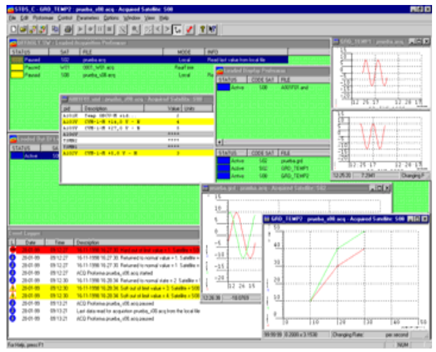
INTRODUCTION

GMV IN SPACE: INTRODUCTION



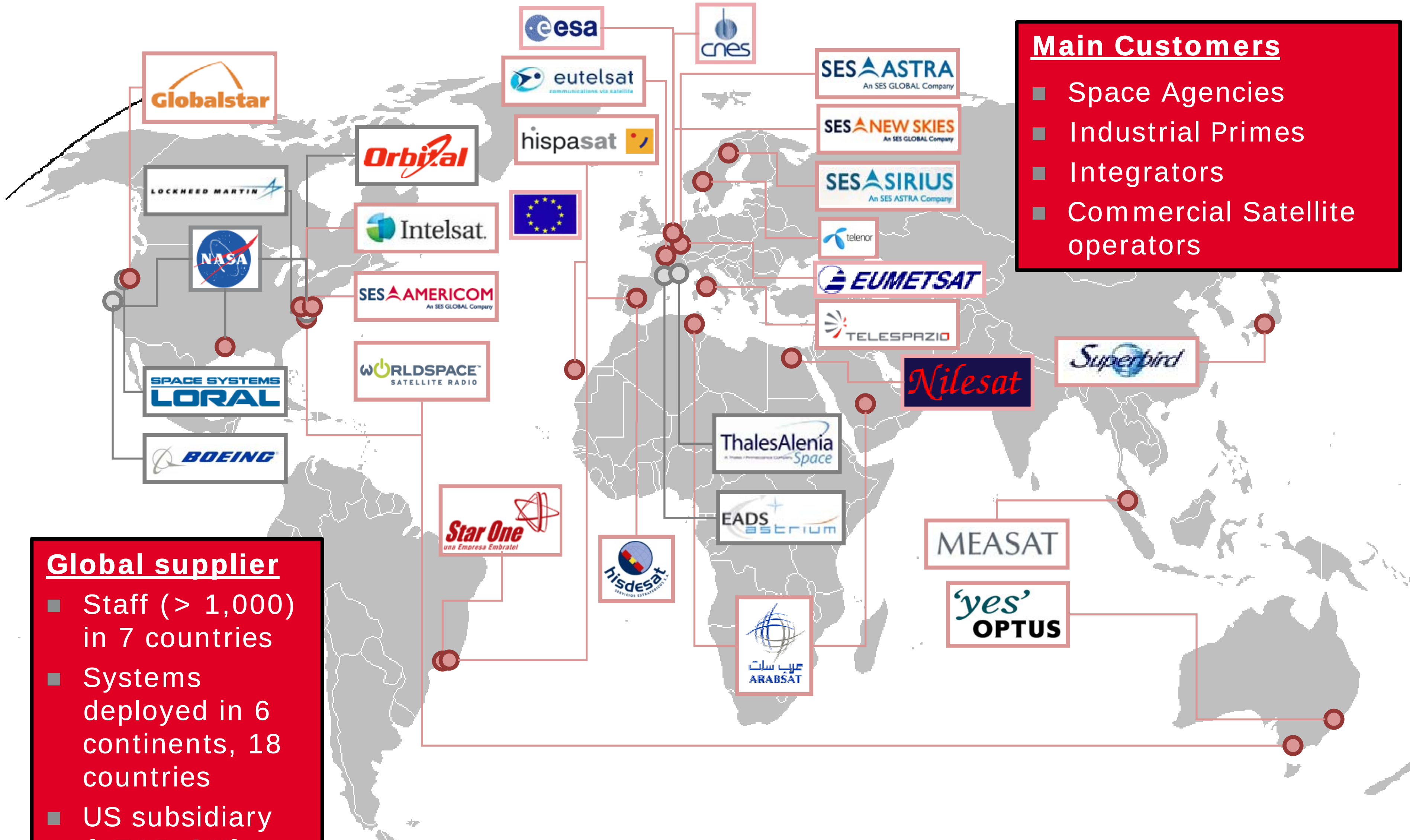






- **Mission Analysis** studies and mission analysis tools (station keeping, collocation, launch window analysis, ...)
- **Operational systems** for satellite control (inc. on-station and LEOP):
  – **Real-Time TM/TC M&C**
  – **Flight Dynamics**
  – **Mission Planning and Scheduling**
- **Special operational needs** (e.g. collision prediction/analysis, rendezvous, interstellar)
- Satellite **capacity management**:
  – Satellite capacity management
  – Payload Reconfiguration
- **Operations support**

GMV IN THE WORLD: MAIN SPACE CUSTOMERS



Main Customers

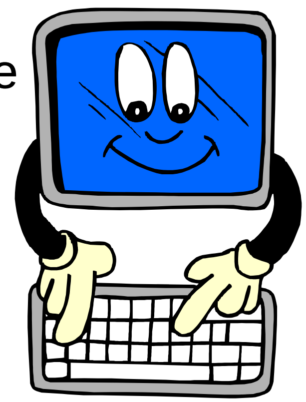
- Space Agencies
- Industrial Primes
- Integrators
- Commercial Satellite operators

Global supplier

- Staff (> 1,000) in 7 countries
- Systems deployed in 6 continents, 18 countries
- US subsidiary **(ITAR OK)**

INTRODUCTION (1/2)

- Presentation shares the results of GMV's experience regarding:
  - **Development of a new automation layer** for Satellite Command and Control (SCC) integrated with the ***hifly*** suite
    - Python as automated procedures language
    - Layer was expected to support
      - Automation of **operational** satellite control procedures
      - Automation of **non-regression SCC testing** during development, integration and maintenance.
      - All previous points also applied to **ground equipment**



Broad view of automation, applied well beyond actual operational procedures

- Experience migrating legacy and new procedures in this new infrastructure using **Python** as the scripting language

INTRODUCTION (2/2)

- Experience shared is based on several recent programs:
  - **COMPLETED:** Migration of the complete ground system of **Star One's Brasilsat B series fleet**:
    - 4 Boeing BSS-376W satellites
    - State-of-the-art ground system with cost-effective software and hardware components
  - **IN PROGRESS:** Development of an extension for a new satellite (**Star One C3**, built by Orbital Sciences Corporation), with new procedures originally in STOL.
  - **IN PROGRESS:** Development of new procedures for **Hispasat H1E**, built by Space Systems Loral. Starting with paper procs.

SCC AUTOMATION LANGUAGES

EXISTING LANGUAGES FOR SCC AUTOMATION

- SCC automation approaches:
  - **procedural scripts**
    - Space-specific languages
    - General purpose languages

our focus

  - rule-based expert systems
  - finite state models

- Multiple **space-specific languages currently used**:
  - **STOL**: Satellite Test and Operations Language
    - Originally developed by NASA, multiple flavors
    - Widely used by many GOTS and COTS
  - **PLUTO**: some ESA missions (SCOS-2000)
  - Multiple **proprietary languages** used by different companies: SOL (GMV), CCL (Harris), OCIL / CECIL (Raytheon), PIL (Astrium), SCL (ICS), etc

- **General purpose languages** used in some missions: Perl, Tcl

CUSTOM vs GENERAL PURPOSE LANGUAGES

	SPACE-SPECIFIC (eg. STOL)	GENERAL PURPOSE (eg. Python)
PROS	❑ (Sometimes) more user friendly for non-programmers ❑ Adapted to satellite operations ❑ High reliability	❑ Open source ❑ Very powerful ❑ Portable ❑ Language can be easily restricted / extended ❑ Wide availability of tools and programmers
CONS	❑ Proprietary language and/or tools ❑ Portability issues ❑ Limited, enhancements are expensive	❑ Potentially less readable if coding is not done carefully ❑ Too powerful?

HOW ABOUT PYTHON?

GMV INNOVATING SOLUTIONS

Python

- Python is a **portable, open, high-level, object-oriented, dynamic language**
- Conceived in the 80s, **used massively since the 90s**
- Recognized widely for its **readability**, **maintainability** and **modifiability**, key aspects for complex procedures that may be modified multiple times throughout a mission.
- **Performance** is much better than most other dynamic languages.
  – Compiled to bytecode
- **Widely supported by the software community**, which guarantees the availability of good programmers, Integrated Development Environments (IDEs) and extensions.

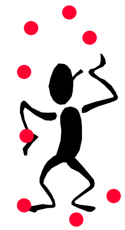
```
def add5(x):
    return x+5

def dotwrite(ast):
    nodename = getNodename()
    label=symbol.sym_name.get(int(ast[0]),ast[0])
    print ' %s [label="%s' % (nodename, label),
    if isinstance(ast[1], str):
        if ast[1].strip():
            print '= %s"];' % ast[1]
        else:
            print '"]'
    else:
        print '"];'
        children = []
        for n, child in enumerate(ast[1:]):
            children.append(dotwrite(child))
        print ' %s -> {' % nodename,
        for name in children:
            print '%s' % name,
```

Multiple successful applications in space business
E.g. Shuttle Mission Design

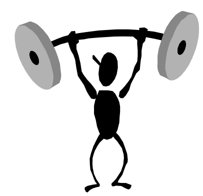
ADVANTAGES OF THE USE OF PYTHON (1)

- **Portable**
  - Windows (XP, CE, Pocket PC), Linux, UNIX, Macintosh
  - Many others: AIX, AROS, AS/400, iPOD, OS/2, Palm OS, Playstation, Psion, VxWorks, Nokia cell phones, .NET, Java Virtual Machine, ...

- **Open**
  - Free, even for commercial use.
  - Interpreter can be embedded in products (no license fee)
  - Open source, no GPL-like traps



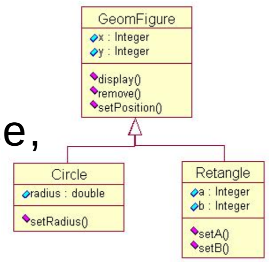
- **Dynamic**
  - Dynamically typed and interpreted, ideal for fast scripting



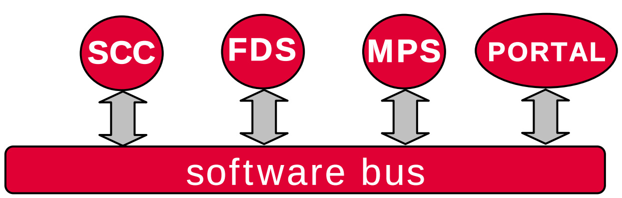
- **Powerful**
  - Complex built-in data structures (e.g. flexible arrays, lists, dictionaries)
  - Great variety of program control instructions
  - **Productivity 5 – 10 times higher than Java**
  - Supports exception handling
  - Automatic memory management and garbage collection
  - Language is extensible

ADVANTAGES OF THE USE OF PYTHON (2)

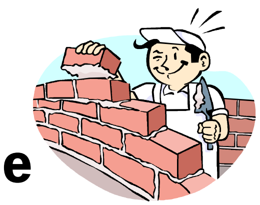
- **Object orientation**, with all the associated benefits (reuse, abstraction, scalability, ...)
  - Supports classes, inheritance, templates



- Easy integration with
  - existing Service Oriented Architecture (**SOA**) implementations
  - **Web Services** (WSDL)
  - **GMSEC** API (Python supported)



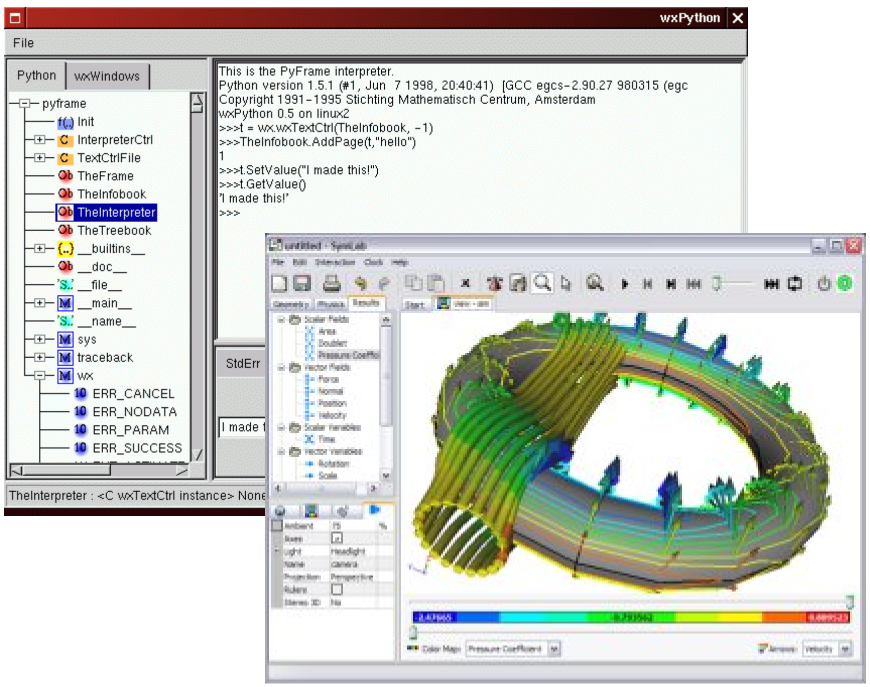
- **Built-in development capabilities**, given as language modules
  - Automatic documentation generation
  - Unit testing, regression testing
  - Debugger, profilers, interpreter, compiler



- Availability of **multiple modules** for
  - **XML** processing: Multiple applications: XTCE DB parsing, SOAP messaging, etc
  - **Communications**: Sockets, Internet access, RPC, email
  - Time **performance measurement**
  - **Many others**: database access, math, data compression, multi-threading, cryptography, operating system access, etc

ADVANTAGES OF THE USE OF PYTHON (3)

- Availability of bindings for multiple **GUI-development toolkits** (Qt4, GTK2, Tk, wxWidgets, etc)



- Wide variety of **plug-ins for Eclipse** (a popular open development platform) can be used to work with Python.

- Availability of multiple, powerful, **free tools** for
  - Development
  - Source code inspection and metrics generation
  - Debugging, testing
  - Configuration management

- Wide support by **commercial tool vendors**

RISKS OF THE USE OF PYTHON

- Language may be ***too powerful*** and complex for non-programmers.
  - This can be handled by restricting the use of certain instructions from the development environment
- **Readability** may be worse than space-specific languages if coding is not done carefully
  - Strict coding standards are needed
  - Coding can be abstracted for non-programmers using a visual environment
- **Evolution of language** is controlled by others
  - This is part of the deal of using a general-purpose language
  - Compensated by all the advantages
  - A mission can just freeze the Python version & development environment and use updates on a case-by-case basis
- **Dependency** on third-party software (the interpreter)
  - But it is open source

autofly
Overview

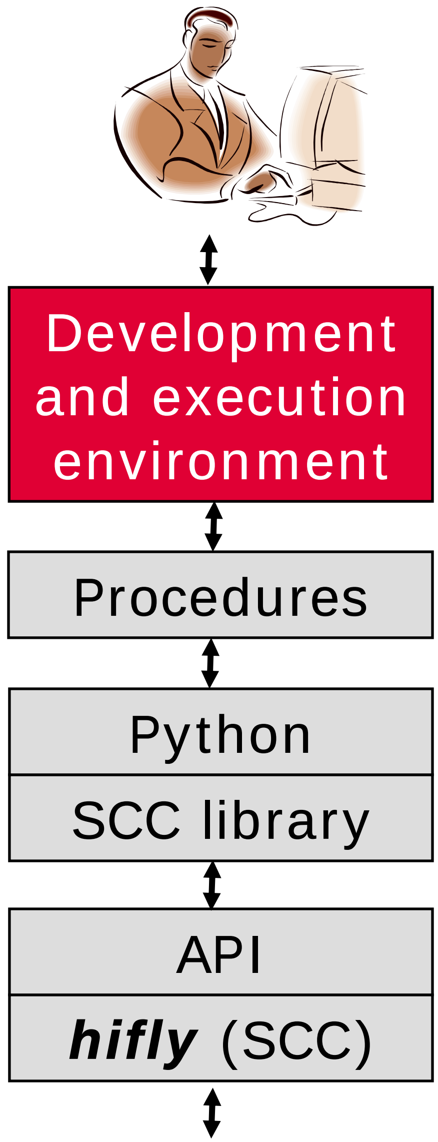
INNOVATING SOLUTIONS

autofly OVERVIEW

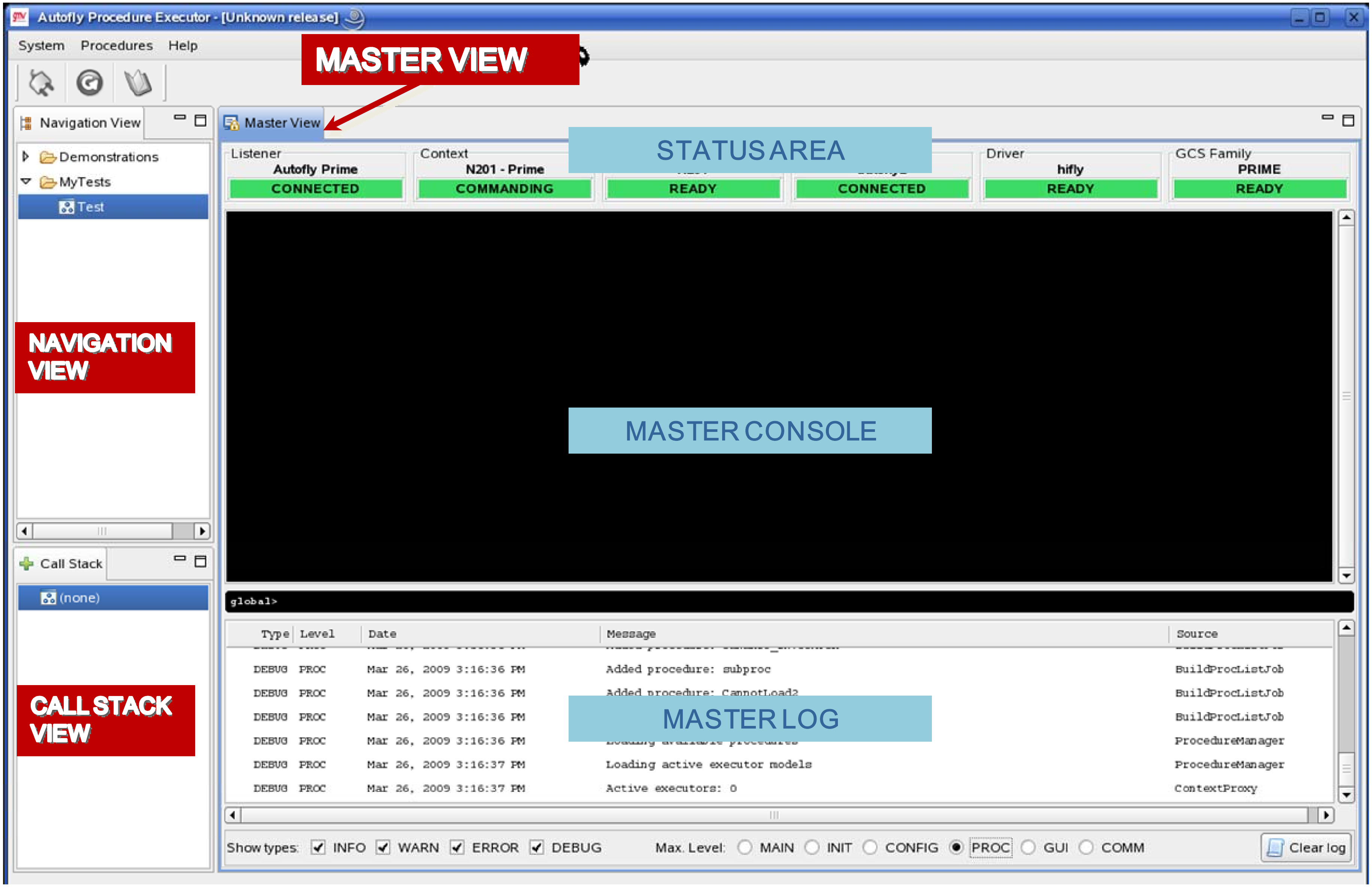
• ***autofly*** is a native ***hifly***® component allowing execution of satellite operations through automated procedures

• ***autofly*** provides the following capabilities:

- Creation and modification of procedures from the GUI
- Validation of procedures (statically and dynamically)
- Scheduling (based on time or TM condition) of procedures for execution
- Monitoring of procedure execution progress



autofly Main Window

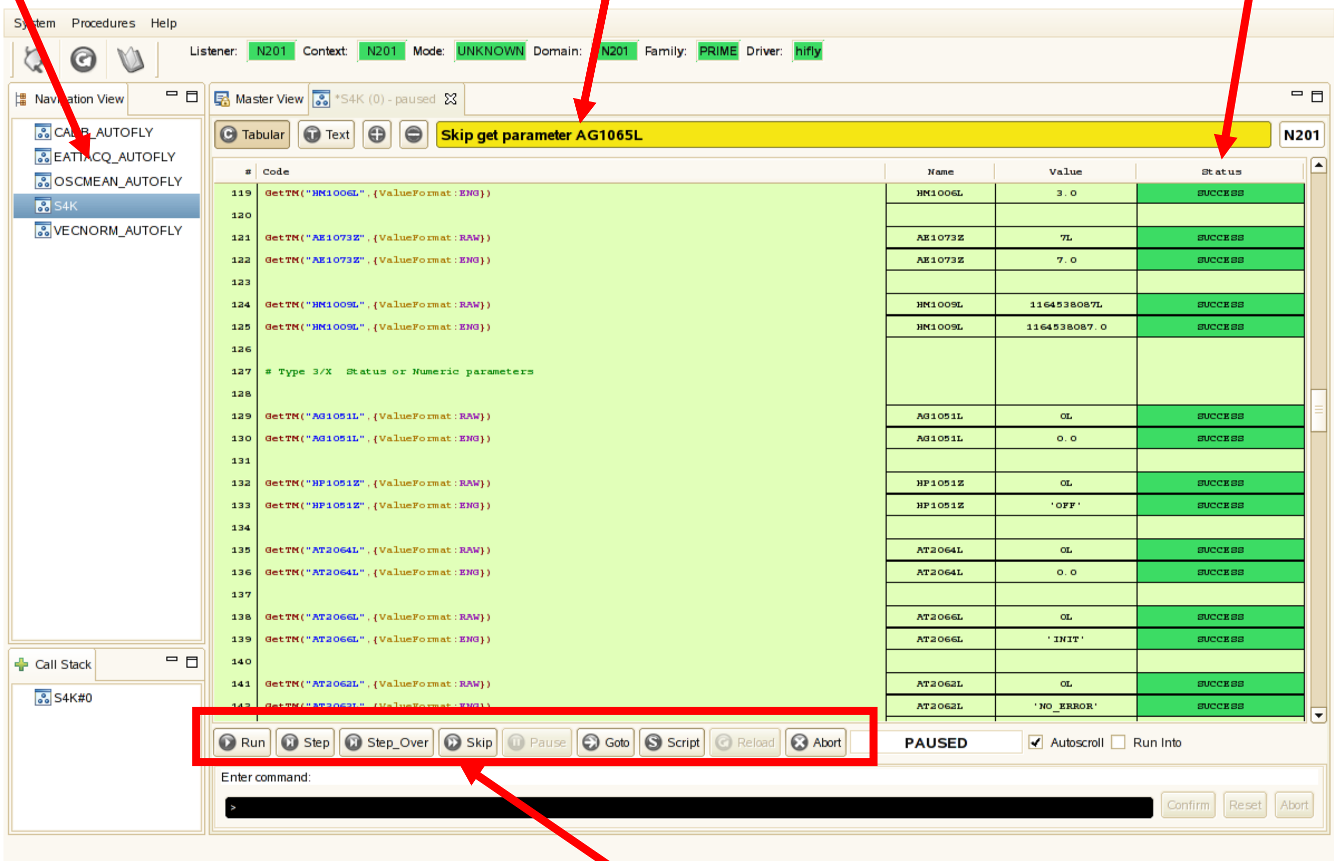


autofly Procedure Execution

Procedure List

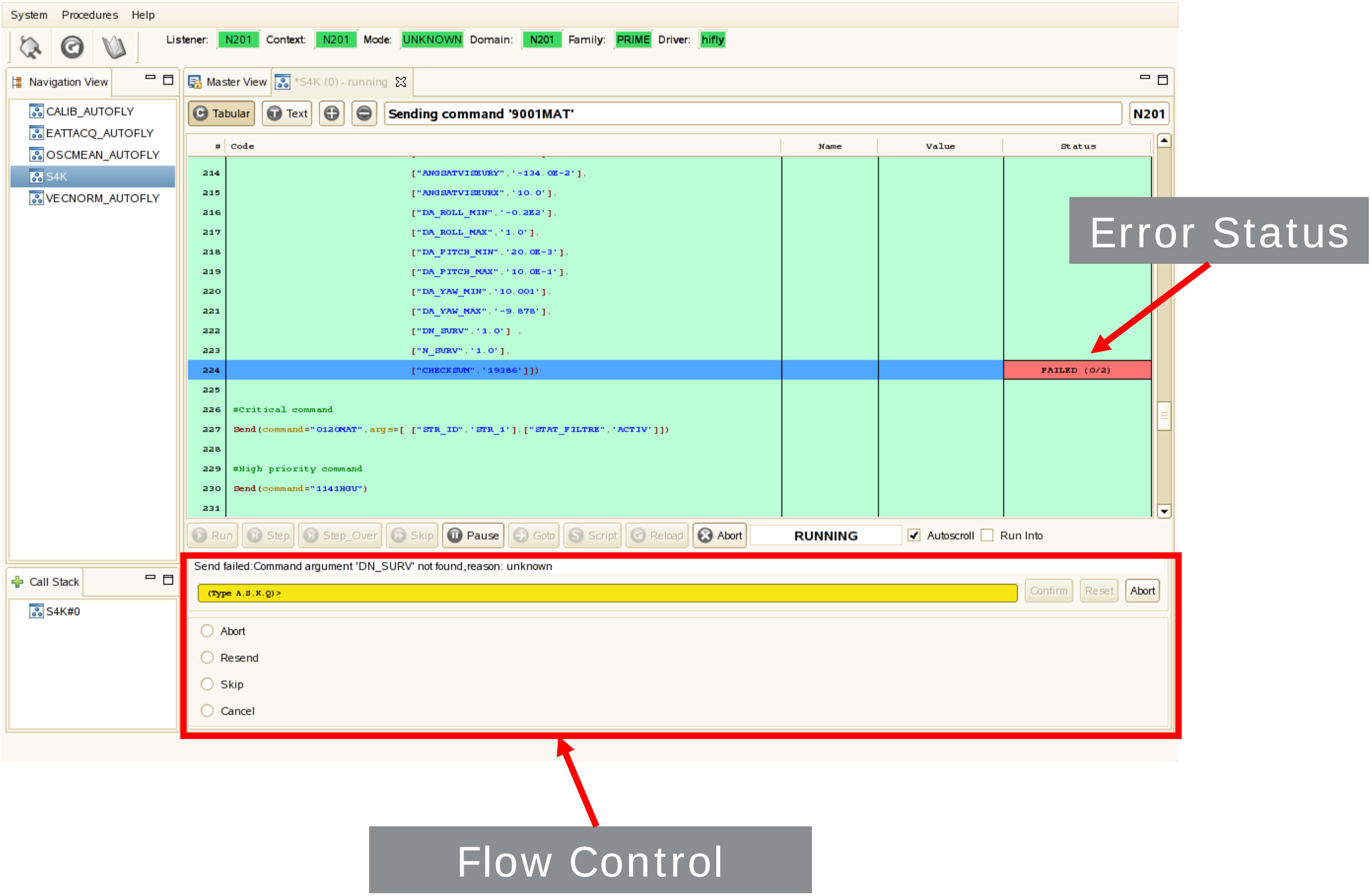
Status Bar

Execution Status



Execution Controls

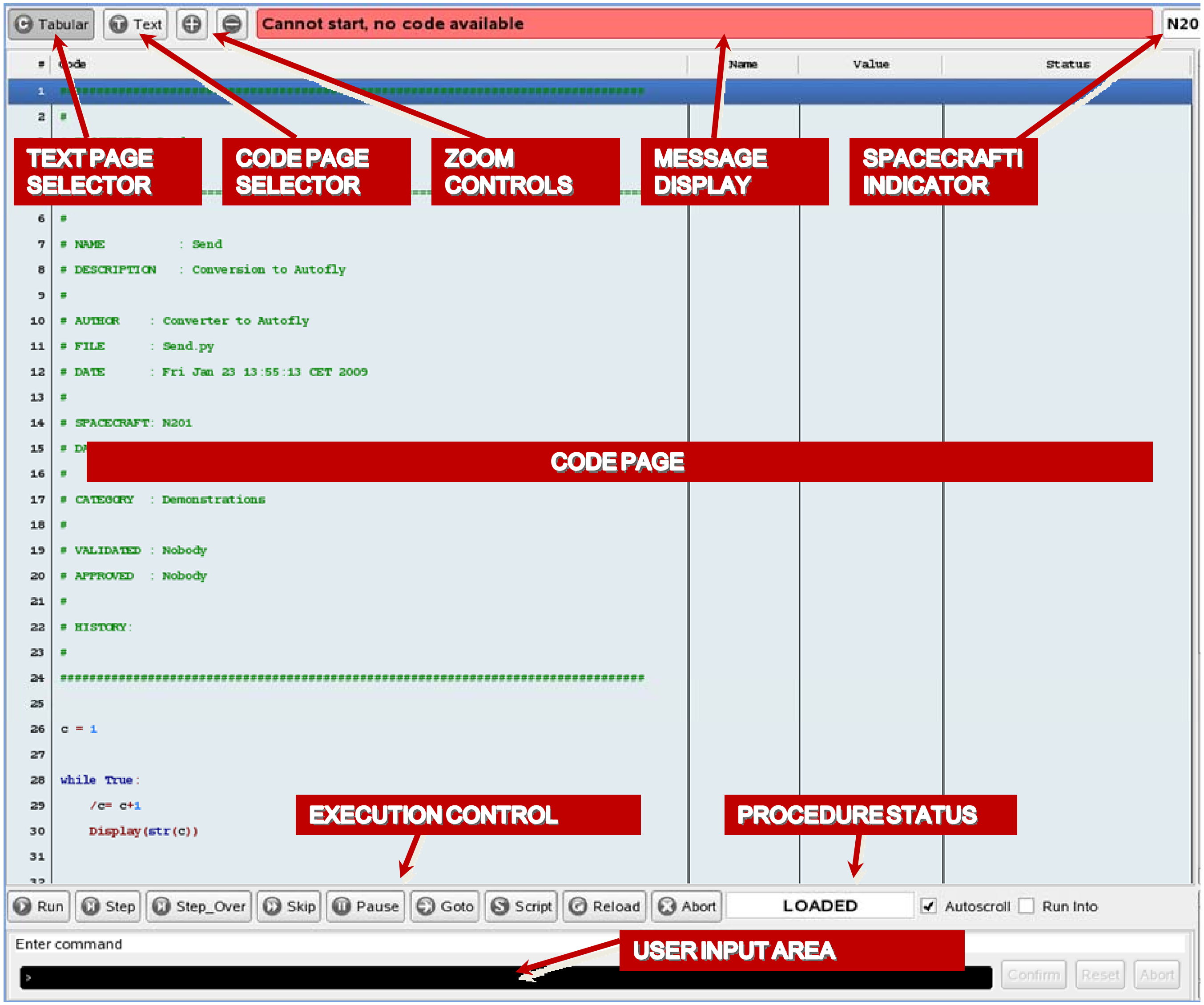
autofly Procedure Execution (cont.)



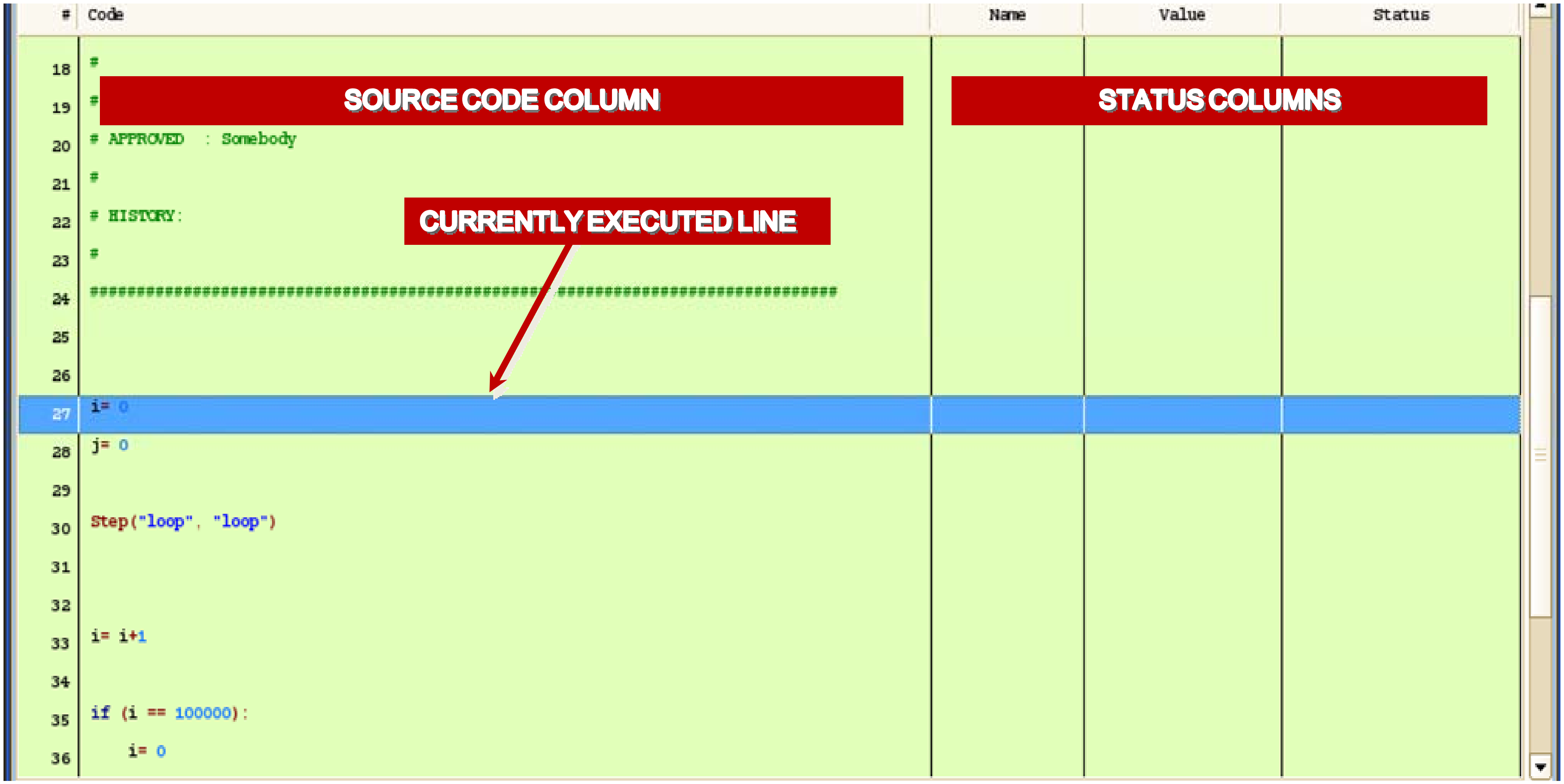
autofly Procedure Execution (cont.)

- 2 views: **Tabular** (displays procedure view highlighting the line being executed) and **Text** view (displays procedure output).
- **Execution controls**:
  - Run / Pause / Step / Step over (avoids going into sub-procedures)
  - Skip: skips the next statement
  - Goto: go to a given line number in the procedure
  - Script: executes basic statements
    - **Send**("T001"): send command T001
    - **GetTM**("A001"): get TM A001
    - **Display***(str(a))*: display value of variable "a"
    - a = 1: sets value of variable a to 1
    - **StartProc**("Operational/Test")
  - Reload: reload the procedure execution
  - Abort: abort the procedure execution
  - Entry to pass inputs to the running procedure

autofly Procedure Execution (cont.)



autofly Procedure Execution (cont.)

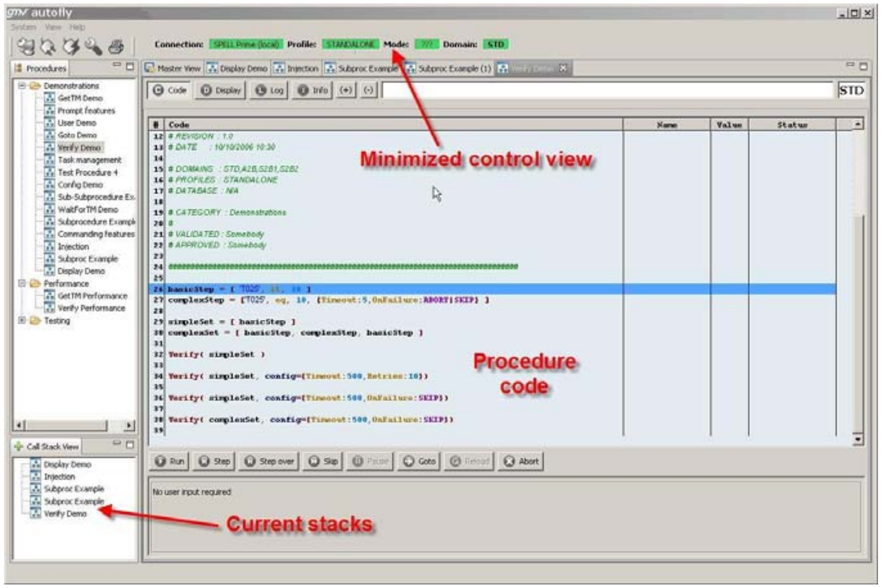


PROCEDURES MIGRATION

INNOVATING SOLUTIONS

OVERVIEW

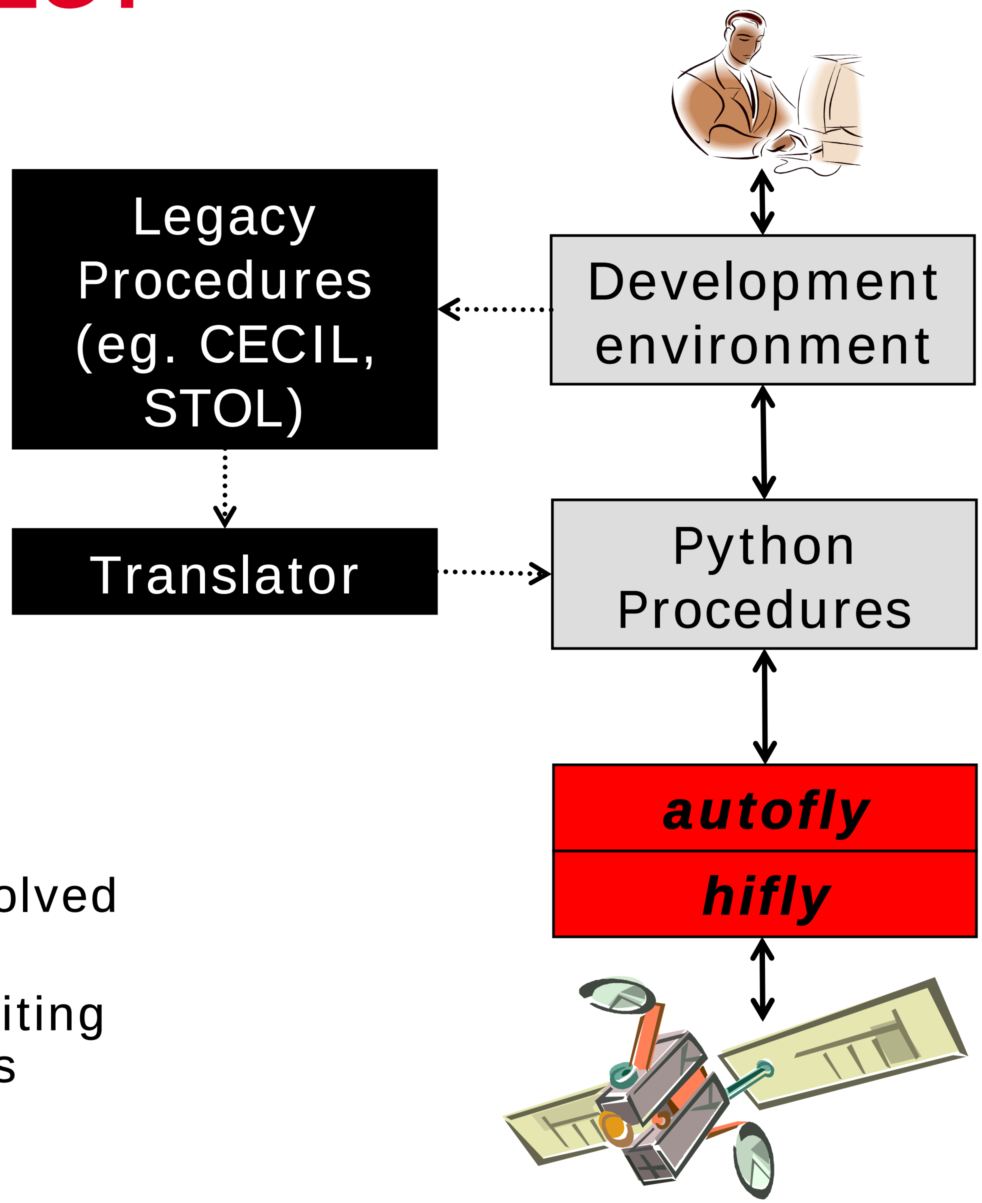
- ***autofly*** allows the operator to develop, test, modify, schedule and execute Python procedures, with:
  - Procedure execution
    - Parallel execution supported
  - Procedure control
    - Supports Step-by-step execution
  - Procedure monitoring



- ***autofly*** supports:
  - TM access and injection
  - TC injection and status monitoring
  - Event and out-of-limits access
  - Event injection
  - Modification of out-of-limit definitions
  - Open predefined TM displays
  - Display operator messages and prompt for input
  - Procedure nesting

MIGRATION STRATEGY

- In the case of the Star One migration, a **translator** script was created to directly translate CECIL code to Python:
  - Avoid creating Python procedures from scratch
  - Iterative process
    - Testing the procedures
    - Updating the translator
    - Re-translating the procedures
  - Repeated conversion issues solved in translation script
  - Minimal amount of manual editing for one time conversion issues
  - Assures **traceability** is easily maintained



VALIDATION STRATEGY

Step 1

Internal Error Reporting in Translation Script

- Invalid characters
- Unexpected logical constructs and arithmetic operators
- Incorrect syntax

Step 2

Automatic Validation of Python code in *autofly*

- Ensure Python code valid
- Sub-procedures called correctly

Step 3

Procedure Execution Against the Dynamic Satellite Simulator

- All logical branches tested
- TCs recognized by the DSS and executed correctly
- TM values received, initiated execution of correct logical branch
- Parameter characteristics updated
- Setting of system variables correct
- Sub-procedures initiated with variable values set

Python fully able to support the logic of legacy procedures

LESSONS LEARNED

INNOVATING SOLUTIONS

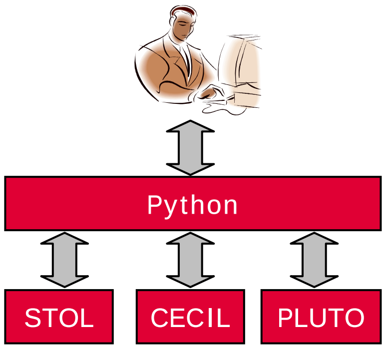
LESSONS LEARNED (1)

- Many **space-specific languages** currently used for the development of operational procedures were **defined decades ago** and are **not used outside of the space industry**. In many cases they are **proprietary** and require **expensive** products.

- Future **support for proprietary languages and** availability of **tools** is not guaranteed. Some operators have had serious problems replacing a system once the HW became obsolete, typical in a GEO mission (> 15 years)
- Lessons from **Ada**:
  – Language designed under contract to the US DoD during 1977 – 1983
  – Targeted at embedded and real-time systems
  – Mandatory for new software DoD projects since 1987
  – Excellent language, used successfully for thousands of projects
  – 2003, Software Engineering Institute:

"Due to a dearth of tools and compilers and lack of trained, experienced programmers [...] Ada is a programming language with a dubious or nonexistent future"

LESSONS LEARNED (2)



- Operators with a **heterogeneous fleet** usually end up having to use different languages. This increases training & operations costs and increases the complexity of the system.

- Python allows the definition of a **homogeneous front-end** for a heterogeneous fleet

- **Coding rules, customized development environment and training** needed to guarantee the high quality & maintainability of procedures

- With Python, operators can **benefit** enormously from the **software community**:
  – Using **modern, powerful, open source languages** like Python and **tools** like Eclipse/RCP widely supported
  – Approach allows the operators to have an **open, integrated environment** for operational and testing procedure development, verification, execution, configuration management and metric generation
  – It also **reduces** the **dependency** on proprietary technologies and the **risks** of software obsolescence

LESSONS LEARNED (3)

- A **close collaboration** between the operator and GMV's team was essential in the cases of **procedures migration**
  - Achieve a complete understanding of the legacy system procedures
  - Ensure a **smooth transition**
  - Presence of operator experts in **long testing campaigns** extremely important

- Performing **procedures testing** against a satellite simulator prior to installation at customer site was highly beneficial
  - Procedures can be used to **validate** TM/TC **database translation**
  - **Identify issues** leading to modifications at an early stage
  - **Reduce risk** for surprised during on-site testing

Thank you!!

www.gmv.com

gmv®

INNOVATING SOLUTIONS

BACKUP SLIDES

WRITING PROCEDURES IN PYTHON

autofly python constructs

- Comments: line starting by "#" character
- Indentation: python groups statements using indentation
- Variables have to be initialized before being used
- Variable names are case sensitive
- Arrays
- Arithmetic expressions
- Conditional statements
- Loops
- User-defined functions

autofly python constructs (cont.)

- Indentation examples:
  - Example 1:

```
if (a == 1):
   b= 2
   c= 3
   d= 4
```

  - Example 2:

```
if (a == 1):
  b= 2
  c= 3

d= 4
```

autofly python constructs (cont.)

- Variables:

```
# initialization of variables
boolean_variable = True
integer_variable = 1
float_variable = 1.0
string_variable = ‘string’

# python allows using either single quotes or double quotes
# this allows to flexibly use quotes as part of the string
string_var1= “This string contains ‘word’ quoted with single quotes”
string_var2= ‘This string contains “word” quoted with double quotes’
```

- Arrays:

```
array_variable = [False, 222, 222.99, ‘any string’]

# access first element in array to get value
first_element= array_variable[0]

# access first element in array to change its value
array_variable[0]= True
```


autofly python constructs (cont.)

- Arithmetic expressions:

```
a = abs(-7.5)                         # Absolute Value
b = asin(0.5)                         # Arc sine, returns in rads
c = pow(b,3)                          # c = b^3
d = ((max (a, b) + 2.0) % 10) * a     # Compound expression
```

- Boolean expressions:

```
a = 1
b = 2
c = 3
boolean_variable = (a != b) and (not (b > c))
```

- String expressions:

```
i= 5
Message = "Integer: " + str(i) + " , Binary: " + bin(i)
Display(Message)
```

autofly python constructs (cont.)

- IF statement:

```
a = 1
b = 2
if (a > b):
   Display ("a is greater than b")

boolean_variable = a > b

if (boolean_variable):
   Display(“a is greater than b”)
elif (boolean_variable):
   Display(“a is less than b”)
else:
   Display(“a is equal to b”)
```


autofly python constructs (cont.)

- Loops:
  - for

```
# loop 10 times (with variable "i" taking values 0 to 9)
array_variable = [0,1,2,3,4,5,6,7,8,9]
for i in array_variable: Display(str(i))
```

  - while

```
# loop 10 times (with variable "i" taking values 0 to 9)
i = 0
while (i < 10):
   Display(str(i))
   i=i+1
```

autofly python constructs (cont.)

- Functions:

```
# Function declaration
def funcA(x, y):
    a= x+y
    b= x+1
    c= y+1

    return a, b,
```


autofly built-in functions

- GetTM: timeout, format, wait

  variable = GetTM('TM NAME', **Wait** = **False)**

- Verify: timeout, format, delay, wait, retries, tolerance

  Verify (['TM NAME', eq, 'VALUE'], **Delay= 20, Timeout= 5, Retries= 4)**

  verif = [['PARAM1', eq, 'VALUE1',],
  ['PARAM2', eq, 'VALUE2', {**Retries**:**3**}],
  ['PARAM3', eq, 'VALUE3', {**Wait**:**False**}]]
  Verify (verif, **Timeout = 10)**

- Send

  Send(command = 'CMDNAME', args = [['ARG1', ' ON'],
  ['ARG2', 0xA3C, {**ValueType**:**LONG**, **Radix**:**HEX**}],
  ['ARG3', 0.34]] **)**

autofly built-in functions

- GetTCparam

  execCommand = GetTCparam('CMDNAME', 'execCommand' **)**

- SetGroundParameter

  SetGroundParameter ('VARIABLE NAME', Value**)**

- GetTMparam: get limits

  # Get the lower hard (Red) limit defined:

  LoRed= **GetTMParam**("A0001", **LoRed**)[0]

- SetTMparam: set limits

  # Set all limits

  ret = **SetTMparam**("A0001",**LoRed**=1.0,**LoYel**=0.5,**HiRed**=3.0, **HiYel**=2.0)

  # Set all limits (with hard and soft limits set to the same value)

  ret = **SetTMparam**("A0001",**LoBoth**=1.0,**HiBoth**=3.0)

autofly built-in functions (cont.)

- Event: raise events with a specific severity

  # By default severity is INFO if qualifier is not specified

  Event (‘MESSAGE’, **Severity** = INFORMATION**)**

  Event (‘MESSAGE’, **Severity** = WARNING**)**

  Event (‘MESSAGE’, **Severity** = ERROR**)**

- WaitFor: holds the execution. Different options: interval, time, delay, TM condition.

  WaitFor ([‘TM NAME’, eq, ‘VALUE’], **Delay** = **20)**

  WaitFor(2)

- Display: displays a message in the GUI and log the message

  Display (‘a = ‘ + a + ‘i = ‘ + str(i))

autofly built-in functions (cont.)

- Prompt: prompts the user for input

  user_choice= Prompt ('Please make choice', OK_CANCEL)

  user_supplied_value= Prompt ('Enter value: ', NUM)

  user_supplied_date= Prompt ('Enter date: ', DATE)

- OpenDisplay:

  OpenDisplay ('Display_name')

- PrintDisplay:

  PrintDisplay ('Display_name', Format= "vector")

  PrintDisplay ('Display_name', Printer= "lpt1")

- Flow control: Step, Goto, Pause, Abort.

  Step('Step_name', 'Title')

  ..

  ..

  Goto('Step_name')