JOURNAL OF AEROSPACE COMPUTING, INFORMATION, AND COMMUNICATION
Vol. 7, November 2010

Prototyping a Domain-Specific Language for Monitor and Control Systems

Matthew Bennett*, Richard Borgen†, Klaus Havelund‡, Michel Ingham§, and David Wagner¶
Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109

DOI: 10.2514/1.40331

This paper describes a domain-specific language prototype developed for the NASA Constellation launch control system project. A key element of the launch control system architecture, the domain-specific language prototype is a specialized monitor and control language composed of constructs for specifying and programming test, checkout, and launch processing applications for flight and ground systems. The principal objectives of the prototyping activity were to perform a proof-of-concept of an approach to ultimately lower the lifecycle costs of application software for the launch control system, and to explore mitigations for a number of development risks perceived by the project. The language has been implemented as a library that extends the dynamically-typed Python scripting language, and validated in a demonstration of capability required for Constellation. A study of the statically typed Scala programming language as an alternative domain-specific language implementation language is also presented.

I. Introduction

IN 2004, the White House announced a new Vision for Space Exploration [1], calling for a campaign of ambitious missions involving human and robotic exploration of the Moon, Mars, and beyond. In response to this call, NASA established the Constellation program**, which is planning to replace the aging space shuttle fleet with new vehicles capable of transporting crew and cargo to the International Space Station, then on to Moon and, ultimately, Mars.

Constellation represents the first major human space flight program start in many years. Budget constraints will require the program to make gradual progress towards its far-reaching goals over an extended schedule, placing unprecedented requirements on the evolvability of the supporting infrastructure and challenging NASA to significantly reduce the operations and sustaining costs associated with its missions. Consequently, the program's success will require significant upgrades to the ground-based infrastructure needed to assemble, test, and operate these new vehicles.

One such system is the launch control system (LCS) at the Kennedy Space Center (KSC). In addition to coordinating and controlling the launch sequence, this system will be used to test the spacecraft, launch vehicles, and possibly their component subsystems as they are delivered to the Space Center and assembled for launch, to control various pieces of ground support equipment used during operations, and to ensure the safety of all of these operations.

Received 8 August 2008; accepted for publication 20 October 2009.

* Senior Software Systems Engineer, Flight Software Systems Engineering & Architectures, matthew.b.bennett@jpl.nasa.gov
† Senior Software Systems Engineer, Flight Software Applications & Data Product Management, richard.l.borgen@jpl.nasa.gov
‡ Senior Research Scientist, Flight Software Applications & Data Product Management, klaus.havelund@jpl.nasa.gov
§ Technical Group Supervisor, Flight Software Systems Engineering & Architectures, michel.d.ingham@jpl.nasa.gov
¶ Senior Software Systems Engineer, Flight Software Applications & Data Product Management, david.a.wagner@jpl.nasa.gov
** Constellation Program web site: http://www.nasa.gov/exploration.

Within the LCS, software applications will command distributed hardware “end items”, including the vehicles and associated ground support equipment, and monitor the telemetry produced by these end items.

Under the auspices of the Constellation Ground Operations project, engineers at KSC, in collaboration with personnel from JPL and other NASA Centers, conducted an LCS proof-of-concept activity over a roughly one-year period from late 2006 to late 2007. This proof-of-concept effort was intended to explore ways to mitigate a number of development risks perceived by the LCS project. Key among those risks were projections of the costs associated with the development and maintenance of software applications used to conduct launch processing operations. Experience with the current Space Shuttle test and checkout system suggests that a significant part of the ongoing operations costs are related to the development and maintenance of monitor and control applications. Particularly expensive is the extensive process by which system engineers express requirements for test procedures in prose, software developers translate these requirements into code, and then both sets of experts are engaged in verification of the resulting application’s correctness. The new system will have to significantly reduce these costs while assuring a consistent high level of safety and security.

One element of the LCS proof-of-concept activity was an investigation of the potential benefits of using a prototype domain-specific language (DSL) that systems engineers would be able to use to write executable specifications of monitor and control applications (i.e., capture detailed requirements in a form that would either be directly executable or automatically translatable to a software implementation). The principal objective of this DSL prototype was to demonstrate an approach that will ultimately lower the lifecycle costs of monitor and control applications for launch processing, test, and checkout. An experimental implementation of the DSL was developed as a library in the dynamically typed Python scripting/programming language [2]. An additional experimental implementation of elements of the same DSL was later developed in the statically typed programming language Scala [3]. The paper presents the results from the LCS DSL prototyping task. The paper is an extension of [4].

Section II introduces the problem domain and provides a brief overview of the envisioned LCS architecture. Section III goes on to describe our process for design and development of the LCS DSL, including a survey and assessment of existing DSLs against the LCS requirements and evaluation criteria. Section IV gives a description of the DSL design in Python, and Sec. V presents an overview of the prototype implementation. Section VI discusses how the language and prototype implementation were evaluated by application to two different problems for the Space Shuttle system. Section VII presents a Scala solution to the DSL and discusses its pros and cons compared to the Python solution. Finally, the paper ends with a discussion of observations and issues raised during the DSL prototyping effort in Sec. VIII, and with a conclusion in Sec. IX.

II. Problem Domain and System Architecture

The LCS problem domain has a number of defining characteristics that pose important challenges and constraints on the system architecture. Given that human lives are at stake, there are especially high demands for stability, security, verifiability, and fault tolerance. With the routine handling of hazardous materials, it is essential to physically distribute the system to separate human operators from potentially catastrophic events. In addition, given the large past investments in physical infrastructure, significant interfaces to legacy systems are unavoidable. Finally, the Constellation program represents a massive multi-center, multi-decade effort, so the new LCS will also be a very long-lived system, making total life-cycle costs a major concern. Such overarching issues drive the qualities required of the LCS architecture.

Before the start of the proof-of-concept activity, the project evaluated several architecture options. The existing architectures for the Space Shuttle, the International Space Station, and the Evolved Expendable Launch Vehicles (EELV) systems were not ultimately selected, due to a number of distinguishing requirements for the Constellation program and lessons learned from the experiences of developing and maintaining these systems. Nonetheless, various aspects of these architectures served as a source for the significant subset of requirements that the Constellation LCS will share with these previous systems. For example, the scripting language used in the current Space Shuttle command and control system, called the ground operations aerospace language (GOAL) [5], was used as a basis for comparison during the DSL survey and assessment, and as a source of inspiration for the specification of the LCS DSL.

At the end of the architectural evaluation process, the Constellation Ground Operations project opted to develop a “standards-based architecture”, that is, an architecture that calls for extensive use of industry standards and standard

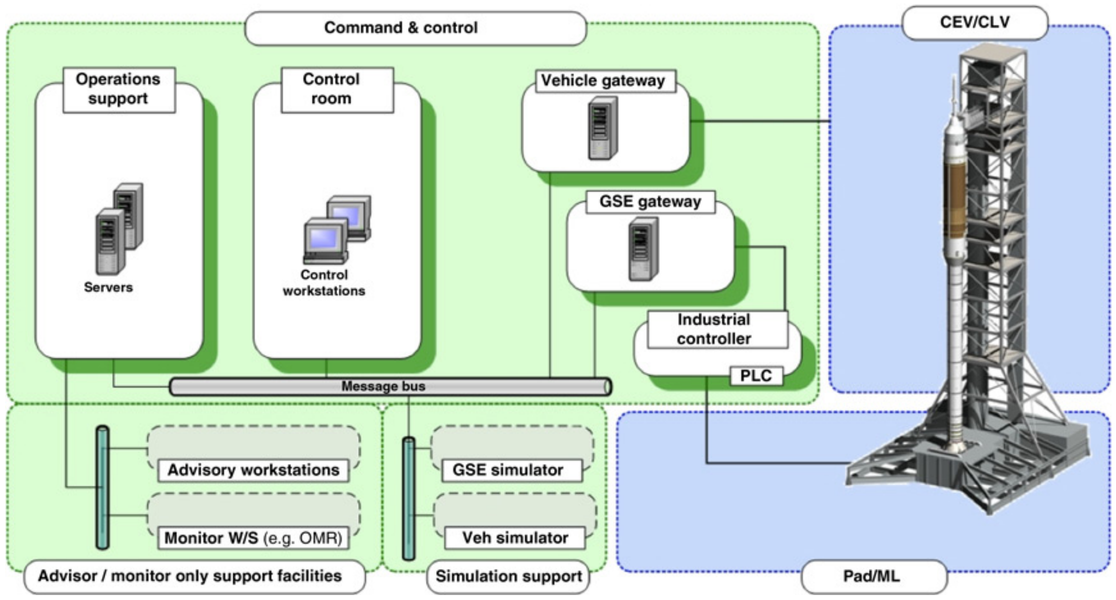


Fig. 1 The Constellation LCS distributed architecture.

software interfaces to achieve interoperability, adaptability, portability, and lower development costs. More broadly, the standards-based architecture calls out the following architectural principles:

- Distribution of control responsibilities, especially to separate supervisory vs real-time control;
- Incorporation of standard industrial controls and processes;
- An adaptive strategy that permits variations or extensions with commercial or legacy components;
- Decoupling of software components with communication middleware; and
- Decoupling of software components by use of layering.

The purpose of the LCS system is to support testing and checkout functions as a spacecraft or other vehicle hardware is processed from arrival at KSC through launch. These functions include integrated tests to assure launch readiness. Major elements of the system would be distributed geographically across the KSC launch site, connected by institutional networks, as illustrated in Fig. 1. These elements are divided into five major categories:

- *End items* include the vehicle, launch pad, their subsystems, and other test articles and support equipment that are the targets of control. Different test and checkout applications may interact with different end items.
- *Gateways* provide hardware and software interfaces between the LCS system network and the end items, including the formats needed to communicate with particular end items.
- *Applications* are the control programs that direct and coordinate (or supervise) control of the end items. Control includes the evaluation and maintenance of system health and safety, and performing checkout procedures needed to verify proper assembly and configuration.
- *Displays* provide operator feedback and a mechanism for operators to interact with the applications and, indirectly, with the end items.
- *Middleware* provides a data distribution fabric that isolates applications from the details of network protocols, data formats, and other programming details.

In part, this distributed architecture addresses a physical system constraint: because checkout and launch operations can be hazardous, the launch facility maintains significant distance between areas where those operations are conducted, and areas where the control computers and operators supervise them. Leveraging commodity (and in some cases, existing) networking capabilities to support the bulk of the connectivity would significantly reduce the hardware costs compared to any custom-built solutions. However, many of the target end items would not be able to directly support standard network connections or protocols. To bridge the gap between the LCS and various end items, the architecture provides gateways whose job it is to translate protocols between the LCS and middleware standards, and the diverse media and protocols needed to communicate with the end items. External protocols might include various supervisory control and data acquisition (SCADA) interactions with embedded controllers or industrial controllers, space telecommunications protocols such as those specified by the Consultative Committee for Space Data Systems, and various system bus protocols such as MIL-STD-1553.

Supervisory control would be conducted from test-specific software applications running in application servers in a secure control facility. A key architectural principle is the division of control responsibilities between supervisory control applications and the end items themselves, based on real-time constraints. Because of the distributed nature of the system, and the technology used to support standard network communications and processing, there are limitations to the ability of these networks to deliver data with the kind of deterministic, low-latency timing needed to support real-time closed loop control. Thus, the architecture defines supervisory control to exclude any closed loop control that would require reactions within time constraints shorter than a given threshold (as of this writing, the specific threshold has yet to be determined).

The LCS architecture draws a separation between application servers and control room computers used to host displays and perform user interface functions. This division is intended to enable supervisory control applications to interface with operators indirectly, allowing, for example, one application to interface with multiple operators (operator consoles, or operator stations in the control room), and to allow the user interface functions to be redirected to other displays in the event of a display hardware failure without affecting the continuity of the control application.

This is just one example of many techniques employed to achieve reliability and safety. The architecture calls for various protections against operator errors, including automation of sequencing, automated rule enforcement, appropriate design of human interfaces, and multiple levels of automatic safing. Other measures are aimed at achieving a high level of safety for software and equipment, including security, reliable components, redundancy, and high availability techniques such as fault recovery.

A fundamental principle underlying the LCS architecture is the decoupling of software components through layering, as shown in Fig. 2. Although IP networking hardware and protocols are ubiquitous, they provide a rather

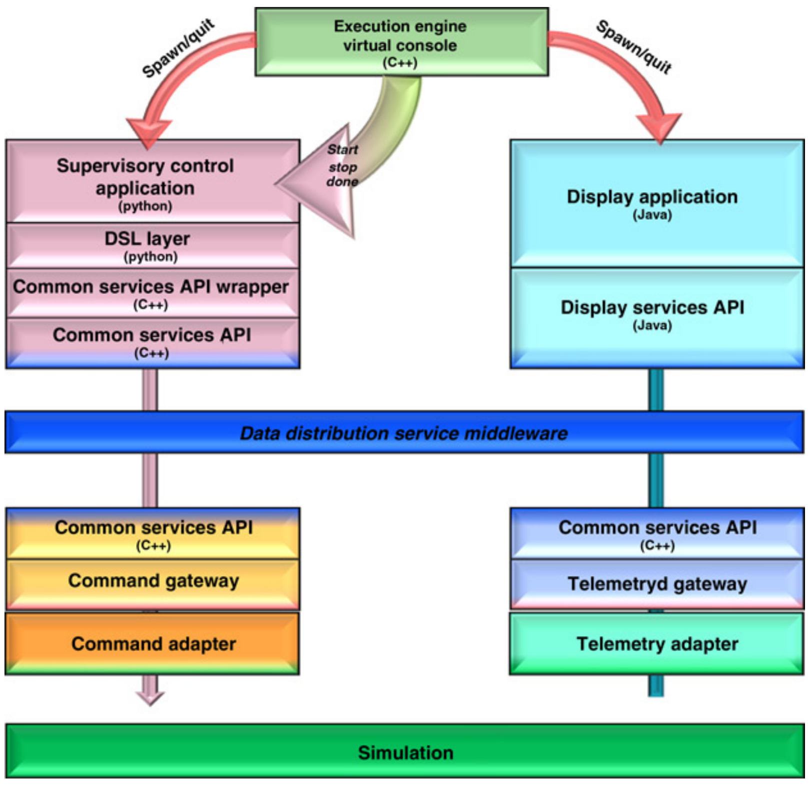


Fig. 2 Layered view of the LCS architecture.

primitive level of service. A message-oriented middleware (MOM) layer is used to provide higher-level application services such as data distribution using a publish-subscribe mechanism, and an abstraction of the logic needed to govern the qualities of service. Abstracting the management of data delivery allows applications to control the trade-offs between data latency, continuity, and volume, without having to get involved in the details of how these services are rendered on underlying hardware and network protocols.

A key characteristic of the publish-subscribe delivery pattern is that it decouples distributed applications from having to know all of the details about the system topology such as which applications need to receive information that a given application creates. The application can publish the message on a defined topic. Other applications needing to use that data can subscribe to the topic, and the middleware will manage the distribution and delivery of the data from the publisher to all subscribers according to the qualities of service defined for the topic.

To further isolate applications from the platform and middleware details, the architecture calls for a common services software framework layer. This layer defines the message topics and default qualities of service that the system will support, and adds some topic-specific services that support system-level interactions between applications. Additional frameworks are defined on top of this layer to provide support for specific kinds of applications (e.g., display applications, gateway applications, monitor, and control applications).

The proof-of-concept demonstration required the selection of concrete technologies and products to instantiate a representative prototype of the target architecture. For the MOM layer, the project used a commercial implementation of the data distribution service (DDS), a standard defined by the object management group (OMG) [6]. Another commercial product was used to provide gateway services supporting telemetry and commanding. The remainder of this paper focuses on the design of the Supervisory Control, the DSL, and the Common Services layers of the architecture.

III. Language Survey

One of the early investigations performed as part of the LCS proof-of-concept activity was a study of various programming languages appropriate for use in monitor and control applications. Following a broad survey of available GOTS/COTS DSL and general-purpose language solutions, a small set of candidate languages were selected for more detailed investigation, including six DSLs (with varying degrees of domain-specificity; specific references are omitted here) and three general-purpose programming languages (Python, C++, and Java). Each candidate language was assessed against a set of evaluation criteria. Functional requirements, life cycle costs, and various quality metrics were considered in the assessment process, which included prototype development and analysis for each candidate language. Prototyping of an example test scenario provided the hands-on experience needed to support the analysis results. This survey and assessment was intended to feed a “make vs buy” decision for the monitor and control language to be used in LCS application software development. This section of the paper describes the requirements and evaluation criteria used in the assessment and summarizes the results from this investigation.

A. Requirements

The requirements were organized into categories and were used to determine if a particular language could meet the needs of LCS:

- *General* included requirements for applications to be interoperable with C and C++ system software, be able to perform interruptible and non-interruptible delays, and invoke other applications that execute in parallel or in series.
- *Commanding* included requirements for applications to be able to command ground support equipment and flight end items.
- *Event handling and conditional execution* included requirements for applications to be able to define logical, numeric, and temporal conditions, and to perform conditional branching, immediate verification of conditions, verification of conditions within a time period, and continuous verification of conditions. There were requirements to be able to define events based on measurement value and attribute changes, time changes, external event notifications, and user inputs. In addition, there were requirements to subscribe and unsubscribe to events, and to respond to events with behaviors that include sending textual messages to users, computing derived measurements, sending commands, invoking other applications, and initiating safing programs. Finally, there were requirements to enable and disable events.

- *I/O interface* included requirements for applications to be able to prompt users for input, to be notified of user responses to prompts, and to publish or display a textual message.
- *Communication* included requirements for applications to be able to communicate with one another, and be able to subscribe to only specified message topics.
- *Data handling* included requirements for applications to be able to read the last known value, status, and attributes for a measurement, and to update a measurement's value and attributes. In addition, there were requirements to compare measurement data against criteria using equal, not equal, greater, greater than or equal, lesser, lesser than or equal, and range comparators. Finally, there were requirements to compare measurement data against specified criteria within a specified time period, and to compute criteria that are numeric and Boolean functions of other measurement data.

B. Evaluation Criteria

The criteria were organized into categories and were used to qualitatively evaluate how well a particular language meets the needs of LCS:

- *Usability* included criteria for simplicity, readability, ease of development, configurability of the code generator for compiled languages, modularity, ease of integration, ease of installation, and configuration of an integrated development environment.
- *Reliability* included criteria for verifiability, validatability, diagnosability, and ease of troubleshooting.
- *Maturity* included criteria for commercial availability, pre-existing NASA and Department of Defense usage, size of existing major development efforts, availability of implementations, and availability of support from commercial and user community sources.
- *Performance* included criteria for resource usage, responsiveness for reactive control, and scalability.
- *Versatility* included criteria for applicability to embedded applications as well as non-embedded applications.
- *Extensibility* included criteria for the ability to add new capabilities such as new APIs and libraries, extensions to the language syntax, and new modules and/or parameterized procedures.
- *Portability* included criteria for loose coupling with a particular toolset or execution environment, and for level of effort to adapt the language to the LCS computing platform.
- *Lifecycle costs* included licensing, infrastructure setup and maintenance, and perceived costs for application development (including the use of tools).

C. Survey Results and Conclusions

None of the evaluated languages seemed to be a "100% perfect" fit for the specific requirements of the Constellation LCS (for example, as compared to the GOAL DSL in the Space Shuttle ground operations system today [5]). That is to say, none of them ranked "high" against all of the established evaluation criteria. However, any of the surveyed domain-specific or general-purpose languages could be used to satisfy the functional requirements, though in some cases it might require significant effort, and result in a non-optimal system from the standpoint of the non-functional evaluation criteria. Although the DSLs offer syntax that is suited to monitor and control applications, they also tend to have more limited development and/or execution environments and tool support. General-purpose languages offer significant flexibility in terms of implementing required functionality and impressive portability and maturity characteristics, but suffer primarily in terms of readability, verifiability, and validatability, considering the targeted user base of systems engineers.

Considering the risk-reduction objectives of the LCS proof-of-concept activity, and the uncertainty and fluidity still associated with the Constellation Program and LCS Project requirements, a two-pronged strategy for the prototyping phase of the LCS proof-of-concept was recommended, in which both a COTS solution and a homegrown solution would be explored. This would enable the LCS proof-of-concept team to mitigate the risk associated with premature down-selection of a single application specification language that may not end up satisfying all requirements.

Based on the team's technical assessment, one particular DSL among the COTS options was identified as the strongest contender due to its human space flight qualification heritage, and generally favorable usability and portability characteristics. However, the focus here is on the design of the homegrown DSL, so discussion of the COTS solution is out of scope for this paper.

Among the general-purpose programming languages considered, Python [2] was seen as the most suitable for use as a monitor and control language, based on the evaluation criteria established for the LCS Project. Python was selected to be the base language for the homegrown prototype DSL solution, because it presents the smallest "semantic gap" and shallowest learning curve to a systems engineer user, among the general-purpose programming languages considered. This being said, a couple of areas of particular concern were identified regarding Python, which were examined during the prototyping activity to determine if either of these present a showstopper obstacle for the ultimate adoption of a Python-based DSL. The two areas were Python's lack of static type checking, and the uncertainty associated with the ability of Python to satisfy performance requirements.

D. Internal vs External DSL Approach

In implementing a homegrown DSL solution, two competing strategies were considered:

Internal DSL approach: that is, extending the base language with domain-specific constructs. In this approach, the DSL is Python extended with an LCS-specific Python library. Applications would be written in Python and the execution engine would be that of Python.

External DSL approach: that is, translating a custom DSL into the base language. In this approach the DSL would be distinct from Python, adopting a syntax appropriate to the target domain. The DSL would be translated to Python, and the execution engine would be that of Python, as in the internal DSL solution.

For the LCS proof-of-concept, the team decided to implement an internal DSL. Given the short timeframe for the proof-of-concept activity, it was determined that this approach presented the least schedule risk; the existing Python development environments and tools could be used to facilitate the quick production of a prototype.

Beyond the LCS proof-of-concept, the team issued a recommendation to consider the development of an external DSL and toolset. As discussed in the next section, the LCS team at KSC subsequently developed a prototype Tabular DSL, which is a form of external DSL built on top of the LCS (internal) Python DSL. This Tabular DSL has since been integrated with the rest of the LCS proof-of-concept system. There are significant long-term benefits for having a language customized for LCS applications. In particular, an external DSL has the greatest potential for reducing lifecycle costs. Applications should be more readable/reviewable, verifiable, easier to write by systems engineers, and ultimately more reliable. However, the external DSL option would require significantly more work than the internal option; namely, designing the language, building a translator, constructing a development environment, and building tools such as a debugger, static analyzer(s), and a test harness.

IV. Language Design in Python

The DSL is intended to facilitate writing control applications that remotely monitor and control the state of a set of end items. From the point of view of a control application, the state can be understood as a mapping from measurement variable names (each associated with a sensor in an end item) to values. In the LCS architecture, this mapping is provided through the MOM software sitting between the control application and the end item.

The prototype DSL has been implemented as a Python library consisting of a set of functions (and a few classes). The design is inspired by traditional procedural programming, rather than by object-oriented (OO) programming, in spite of the fact that Python is an OO programming language. That is, the emphasis is on functions rather than on classes and objects. This design was influenced by the goal of defining a domain-specific programming language, where each function in the library represents a language construct. In addition to these domain-specific constructs, all of Python's programming constructs are available to the programmer. In the following discussion, the key language constructs are presented.

A. Measurements

Measurements are views into the state of end items that are being monitored; they represent data samples collected from sensors in the end items. These samples are then drawn into the LCS system network via a gateway, and distributed as measurement messages on the middleware message bus. A measurement service buffers one or more of the most recent samples from each sensor so that these messages can be retrieved on demand by the application. The measurement service design is discussed in more detail in Sec. V of this paper.

A measurement object is an instance of the following class, for which only some of the methods are shown:

```
class Measurement:
  def __init__(self, id, value)
  def getId(self)
  def getValue(self)
  def getTime(self)
  def __lt__(self, other)
  def __eq__(self, other)
  def __add__(self, other)
  def __sub__(self, other)
  ...
```

A measurement object contains a time tag (defined using the Posix Time System), a numerical identifier, and a value. The time tag is automatically inserted in the object upon creation. The class is able to internally represent different value types, and the class also contains methods for examining the type and for extracting values depending on their type. The class defines a set of mathematical relational methods for comparing values (`__lt__` , `__eq__`, ...), corresponding to the relational operators < , ==, etc. The methods are named in such a way that they overload the built-in relational symbols. For example, given two measurements `m1` and `m2`, they can be compared using traditional syntax:

```
  if m1 < m2: ...
```

Measurements can be accessed by the name represented as a string, by calling the function `getByName(name)`, for example: `getByName("pressure")`. An alternative approach is offered through a specially engineered object named `read`: the same measurement can be accessed by the term `read.pressure`, without having to indicate quotes. This is implemented by overriding the Python method `_ _getattr_ _(self,name)` that is part of any object and which gets called on an attribute (in this case `pressure`, converted to a string) when the attribute is referred to but not found in the object.

B. Monitoring Functions

The DSL monitoring functions offer capabilities for testing the values of named measurements as a function of time. A monitoring function is characterized along five dimensions:

Condition: the condition on a subset of the monitored measurements. For example: `pressure >= 300`.

Timing: indicating when the condition should be verified to be true. There are three possibilities:

- now;
- continuously during a specified time period, given as an extra parameter; and
- eventually within a specified time period, given as an extra parameter.

Reaction: a reaction to be executed in case the property gets violated.

Repetition: a Boolean indicating whether the verification should continue even if the property gets violated.

Blocking: a verification can be specified to be *blocking*, in which case the calling application will wait until the verification has been performed. Alternatively, the verification can be *non-blocking*, where the checking is "spawned" to the background whereas the calling application continues to execute. In a non-blocking case where a reaction is to be executed upon violation of the condition, there is a further choice between letting this reaction execute in parallel with the calling application or letting it interrupt the calling application.

Some of these monitoring functions are described in the following. The construct descriptions use the following symbols: *C* stands for a condition to be verified and *R* stands for a reaction to be executed in case a condition gets violated. Both *C* and *R* are assumed to be parameter-less functions. *D* stands for a duration, expressed in seconds. *S* stands for a string—generally a name associated with the verification operation. *N* stands for a natural number. Finally, *F* stands for a Boolean flag indicating whether verification should be repeated in case of property violations. Arguments in square brackets [...] denote optional arguments (note that this is not Python syntax). In Python, such optional arguments can be given default values, but these are not indicated here.

1. *verify(C, [R], [S])*

This blocking construct examines whether the specified condition *C* is true now. The reaction *R* indicates the response that should be executed if the condition is not true. The default reaction is a dialog that is initiated with the user, giving the choice between iterating (redoing the verification), returning from the *verify* function, or aborting the control application entirely. The name *S* passed to the function helps identify the call within an execution log in case a property gets violated. The function returns true or false depending on whether the property succeeds. An example of the use of this function is subsequently provided, where the condition and the reaction are first defined as Python functions, and then the *verify* function is called with these as parameters:

```
def C(): return read.PRESSURE >= 300
def R(): print("Pressure less than 300 PSI")
verify(C,R)
```

Note that due to Python's nameless lambda functions (also known as "lambda abstractions"), one can write the same call of *verify* as follows, without defining the condition as a separate function:

```
verify(lambda: read.PRESSURE >= 300, R)
```

2. *verify_within(C, D, [R], [S])*

This blocking construct verifies that the condition *C* eventually becomes true within the time duration *D*. The reaction *R* indicates the response that should be taken if the condition does not become true within the time duration. If the condition becomes true within its duration, then the function returns immediately without waiting for the duration to expire. The function returns true or false depending on whether the property succeeds verification or not. Assuming the condition and reaction from the previous example, the following example specifies a test that the condition becomes true within 10 s:

```
verify_within(C, 10, R)
```

3. *verify_within_voting(N, C_list, D_list, [R], [S])*

The argument *C_list* is a list of conditions and *D_list* is a list of durations. This construct verifies that at least *N* of the conditions in the condition list become true within their corresponding time durations in the duration list. The reaction *R* indicates the response that should be taken if this verification fails. If enough conditions evaluate to false (such that *N* conditions cannot possibly evaluate to true), the construct will execute the reaction immediately. As an example, consider three conditions *C1*, *C2* and *C3*, as well as a reaction *R* have been defined:

```
def C1(): return read.PRESSURE1 >= 300
def C2(): return read.PRESSURE2 >= 350
def C3(): return read.PRESSURE3 >= 375
def R(): print ("Pressure checks failed")
```

The following call verifies that at least two of the following conditions become true: *C1* within 5 s, *C2* within 10 s, or *C3* within 15 s; if this check fails, the reaction *R* is executed:

```
verify_within_voting(2, [C1,C2,C3], [5,10,15], R)
```

4. *assert_constraint(S, C, R, [D], [F])*

This non-blocking construct registers the condition *C* to be continuously monitored "in the background", after which control returns immediately to the calling application. If the condition at some point evaluates to false, the specified reaction is executed in a separate thread, without disrupting execution of the calling application. If the flag *F* is true (or is not provided), then the monitoring continues after failure is detected, until expiration of *D* if provided. If a duration *D* is provided, the constraint is automatically removed after that duration. The constraint can also be removed explicitly with the function *remove_constraint(S)*. As an example, consider the following program text:

```
assert_constraint("C_One", C1, R1, 25)
verify_within("C_Two", C2, 10, R2)
```

This code will register the constraint *C1* to be monitored in the background, and will then immediately continue execution of the *verify_within* call. The condition *C2* will now be checked to become true within 10 s, whereas the condition *C1* is continuously being checked to be true during the 25 s. If constraint *C1* ever evaluates to false during the 25-s interval, reaction *R1* gets executed on a separate thread while the checking of constraint *C1* continues on new updates (because the optional reactivation flag *F* defaults to true), until the 25 s have elapsed and the monitor is removed.

5. *conditional_interrupt(S, C, R, [D], [F])*

This non-blocking construct is a variant of the former *assert_constraint* construct in that it registers the condition to be continuously monitored in the background, while control returns immediately to the calling application. However, if the condition at some point evaluates to true, the calling application is interrupted (temporarily stopped) while the reaction is executed. The monitor (also referred to as an interrupt handler) is by default removed if a provided duration *D* expires, or after the reaction is executed (unless the flag *F* is provided and is true). The interrupt handler can also be explicitly removed with the function *remove_interrupt(S)*. The library also contains a function *timed_interrupt(S,D,R)*, which interrupts the calling application after the specified duration and executes the specified reaction.

C. Controlling Functions

The LCS DSL library also contains functions for submitting information into the system, such as explicitly updating derived ("application-controlled") measurements, sending commands to end items, sending messages to displays, and initiating user dialogs via the displays. Two functions exist for issuing derived measurements from an application, enabling different applications to communicate via shared variables. The function *set(S,E)* submits a single update of the measurement named by the string *S* with the value returned by the expression *E* (where *E* represents an expression involving one or more measurements). The function *derive(S,E)* defines a new measurement named *S* to be derived from other measurements: querying *S* will henceforth return the value of expression *E*. This means that when any measurement referred to in *E* is updated, *S* gets updated to denote the new value of *E*. As an example, the following code snippet will ideally print the value 500:

```
derive(X, lambda: Y * 100)
set(Y, 5)
print X
```

Note, however, that the set and derive functions are not guaranteed to have immediate effect since they invoke the publish/subscribe middleware services. These constructs send a message to update the measurement, rather than instantaneously assigning a value to a local variable as in a traditional programming language. Hence, in the preceding example, some time delay would have to be inserted before the print statement to ensure that 500 gets printed.

The *send_command(K)* function submits a command object *K*, sub-classing a pre-defined Command class in the library. This is one of the few places where the user has to be aware of OO features. A command's destination is defined and fixed in the LCS information model and cannot be changed by an application. Discussion of the LCS information model is beyond the scope of this paper, but it can be thought of as a database providing system configuration information. As an example, the following statement defines a reaction function that, when applied, submits a command to open a valve. The command is a so-called discrete command requesting an end item to assume a value from a discrete value space {OPEN,CLOSE}. The reaction function occurs as a parameter to a call of the *verify_within* function that checks whether the valve opens within 10 s, and commands the valve open in case it is not observed to happen.

```
open_cmd = Discrete("valve42", OPEN)
def R(): send_command(open_cmd)
verify_within(lambda: read.valve42 == OPEN, 10, R)
```

Finally, the DSL defines functions for sending messages to displays:

```
send_message(text, role, [criticality])
```

and initiating user dialogs via displays:

```
prompt(type, text, role, [listener_role], [timeout], [responses])
```

Such messages are targeted to subscribers (displays) that assume certain roles, hence following the publish/subscribe concept. A function also exists for spawning a new Python program, for either blocking or non-blocking execution:

```
perform(name, args, blocking)
```

D. Tabular DSL

One concern with using a DSL that includes the richness of a general-purpose programming language like Python is the potential for introducing complexity (and thus greater potential for errors) into control applications. This is of particular concern considering that LCS application developers are intended to be systems engineers. For this reason, LCS proof-of-concept team members at KSC developed a prototype Tabular DSL that was interpreted using the Python DSL as execution engine [7]. A tabular program consists of a sequence of lines, each line representing a DSL construct. The first column defines the construct, and subsequent columns define arguments. The tabular approach allows a customized editor to check input arguments and otherwise suggest alternatives, much like a syntax-directed editor.

V. Prototype Implementation

The prototype execution environment for the LCS DSL is implemented as a layered set of libraries in Python and C++, as illustrated in Fig. 2. The messaging services were provided by a commercial implementation of the DDS [6]. The common services layer adapts the messaging services for the particular message data structures used in the LCS system. In keeping with the decision to adopt an internal DSL, the logic implementing the various language constructs described in Sec. IV is coded in Python. This section presents the implementation of the common services and DSL layers of the LCS architecture, and discusses some interesting multi-threading issues that were encountered during the implementation and testing of these layers.

A. C++ Common Services

As shown in Fig. 2, the LCS architecture provides a set of common services for use by supervisory control applications and the command and telemetry gateways. The common services layer is implemented as a set of C++ API libraries. A complementary set of services were implemented in Java to interface with the display applications, as shown in Fig. 2. Figure 3 presents a more detailed view of the common services design, including the interfaces and mechanisms it provides between the DSL layer and the DDS middleware.

A defining feature of the DDS messaging service is that it must be adapted for the specific message data structures it will manage. This is accomplished using tools provided by the vendor that compile message structure specifications defined in an interface definition language (IDL) into C++ source code that will then implement the message class along with associated marshalling support and interface adapters. DDS defines a message topic in terms of one of these specific data structures.

To isolate some of the DDS-specific implementation details from higher-level libraries and applications, the C++ common services libraries also define their own distinct classes for each message type. This was found to be necessary because the DDS IDL translation process does not support the definition of methods in the message types—it interprets them as pure data structures. From a language design perspective, it was important to be able to support type-specific operations (e.g., comparisons) as member methods. So the C++ common services libraries provide interface adapters for each message type, which translate between the message classes used at the Python interface, and corresponding message structures generated from the IDL code used at the DDS interface.

In addition, a few other services are implemented in this layer for convenience. For example, since DDS messaging is asynchronous, the dialog service (see Fig. 3) provides a single blocking dialog method that issues a prompt message and waits for a matching response. Implementing this service here helped to minimize the complexity of the interface between C++ and Python.

The measurement service provides a query interface by which the application may request the "current" (or most recently received) measurement sample for a named sensor (this interface is invoked through a

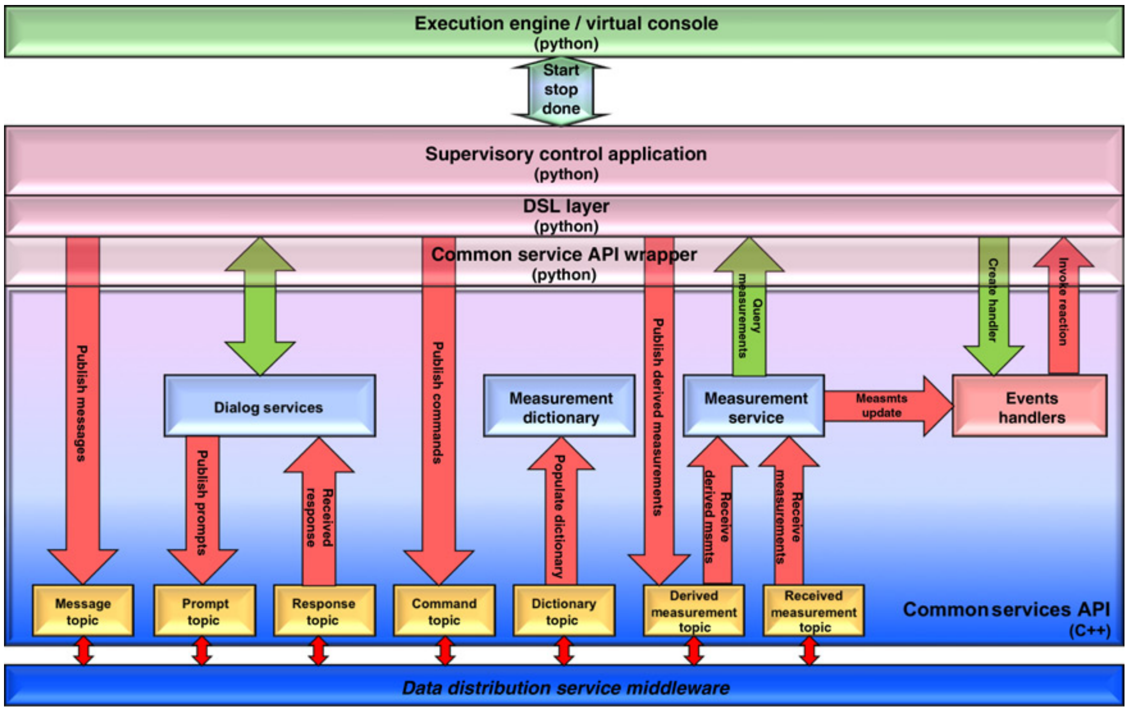


Fig. 3 Common services design.

`read.measurement` call in a DSL application). Because of the potential race condition between measurement samples arriving and when the application queries for samples, an additional measurement event service was required to enable the implementation of monitoring logic that would be sure to see every sample in a given sensor stream. The event service allows the application to register a callback handler that will be used to deliver every new sample associated with the named sensor as it becomes available. This mechanism is used in the implementation of more complex monitor functions in the DSL, including *verify_within*, *verify_within_voting*, *assert_constraint*, and *conditional_interrupt*. The measurement dictionary service provides a dictionary of known measurement (sensor) names and value types needed to allow the event service to verify requests before any data samples have been received.

The interface between C++ and Python is implemented using the Boost library [8], which supports exporting C++ classes and interfaces into Python. One of the key benefits of Python as the implementation language for the prototype LCS DSL was the existence of a well-defined extension API, and several distinct tools supporting interfacing between Python and other C++ tools. In particular, some of these tools support the mapping between Python's object model and that of C++. Of the several tools evaluated, the Boost library (specifically, the Boost.Python library [9]) was found to provide the most complete support for symmetrical OO interactions between C++ and Python. Other tools could export C++ functions or class methods into Python, but Boost was able to export a C++ interface into Python so that Python could implement a class implementing the interface that could then be called from the C++ layer. As will be shown, this interaction is key to the callback mechanism used to implement the event service.

Event services in the C++ common services layer are implemented using listener threads (usually one for each topic) whose job it is to notice when new messages have arrived in a given topic, match those messages against the set of sensor names for which notification callbacks have been requested, and then call each of the registered handlers, which are typically implemented in Python. This requires that these listener threads call through cross-language interfaces to deliver a data object that must also be translated from a C++ structure into a Python object.

To implement these interfaces, a C++ interface (abstract base class) defining the abstract callback method was exported to Python using Boost. This allowed the Python layer to then implement callback handlers derived from this interface, and to pass references to these to the callback registration methods which were also exported from C++ to Python (Fig. 4). Boost provides C++ templates that can be used to articulate an export wrapper for each class

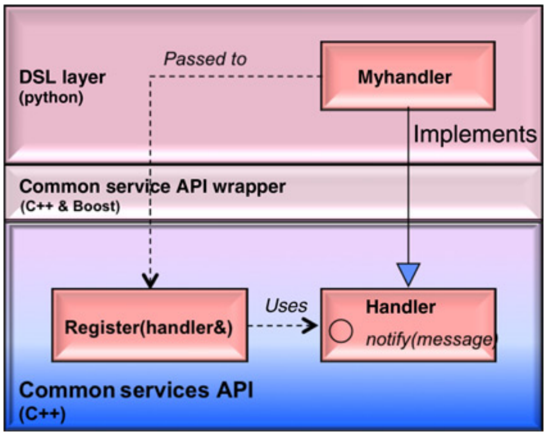


Fig. 4 Event service design.

to be exported from C++ to Python. An export wrapper is a C++ class defined in terms of these Boost templates, which identify the Python-relevant properties of the class, and name the methods and members to be exported. The templates then implement all of the details needed to translate the C++ semantics and interfaces into the Python object model and API, which is implemented in C.

One key function that Boost did not provide, and that required significant additional work in the interface layer, was the synchronization of threads. The Python interpreter is designed to be thread safe, and the Python language can support thread safe operations (thread support can be dynamically enabled in a Python program so that single-threaded Python scripts do not pay the overhead cost of synchronization). Thread safety in the interpreter is achieved through the use of the Global Interpreter Lock (GIL), which is essentially a mutex lock on the entire execution engine. Since the threads used to implement the event callbacks originate in the C++ service library, and not inside of Python, these non-native threads need to interrupt the Python interpreter in order to call handlers in Python. Thus, additional code was required in the callback invocation methods to acquire the GIL before entering Python, and release it after returning. Apart from this detail, all of the interfacing details were handled by Boost templates.

B. Python DSL

As already mentioned, the Python DSL is built on top of the C++ common services. A control application interacts with the common services though a set of objects of C++ classes, wrapped as Python objects/classes in the DSL layer. These include classes for creating objects containing various kinds of information, e.g., measurement objects. There are similar classes for creating command objects, message objects and dialog objects. In addition, the DSL layer includes classes that contain methods for publishing such objects to the middleware, that is, methods for publishing measurements, commands, messages, and dialogs. As an example, the measurement service class offers specifically three methods (the code examples presented in the following are abstractions of the real code, and in some cases Python syntax is not strictly followed, in order to make the presentation more succinct):

```
class MeasurementService:
  def publish(name, value)
  def addHandler(name, handler)
  def removeHandler(name, handler)
  ...
```

The method *publish(name,value)* publishes the value as a new measurement value for the measurement with that name. The two other methods are used for associating and un-associating measurement names with handlers that will get executed when the measurements are updated, typically due to state changes in the end items. A handler is an object of a class that subclasses the following *Handler* class:

```
class Handler:
  def update(measurement)
```

The *update* method of a handler object will execute code specific for a particular condition to be verified. Since the control application can execute while new measurements arrive in the measurement service, the control application and the measurement service execute in separate threads, resulting in a multi-threaded program. The measurement service thread reacts to the publishing of measurements on the message bus and activates handlers if required. In parallel, the control application thread reads measurements, publishes measurements, and registers and removes handlers.

The most interesting DSL functions are the monitoring functions that verify some property over a time interval, and which therefore need to register corresponding handlers with the measurement service. The handlers are all objects of a class *Constraint*, which subclasses the *Handler* class mentioned earlier, and hence overrides the *update* method. The *Constraint* class specifically contains the following data beyond the *update* method:

- *measurement name list*: list of measurement names referred to in the condition.
- *condition*: condition to be continuously monitored.
- *condition reaction*: reaction to be executed when the condition gets violated (becomes false).
- *duration*: duration over which the condition must be checked.
- *duration reaction*: reaction to be executed after this duration has expired.
- *reactivate flag*: true if the constraint should persist after a condition violation while the duration time period has not yet expired.

The *update* method has the following form:

```
def update(self, measurement):
  if not self.condition():
    spawn(self.condition_reaction)
```

The method is here somewhat simplified, in particular code has been left out which manipulates a state machine in order to ensure a correct interaction with the associated timer thread when it comes to removing the constraint. This (somewhat simplified) *update* method just checks the condition and spawns the reaction in a new thread in case it is violated.

In the measurement service class a method exists for adding a constraint (note that the Python-specific "self" argument is omitted here for the sake of clarity in the presentation.):

```
def addConstraint(C, R_c, D, R_d, F):
  constraint = Constraint(C, R_c, D, R_d, F)
  for m in measurements(C):
    ms.addHandler(m, constraint)
  constraint.startTimer()
```

The method creates a constraint from its arguments, and registers the constraint with each measurement in the measurement list (i.e., creates a handler for each measurement in the measurement list). This list is computed from the condition being evaluated and contains all measurement names mentioned in the condition. In the real implementation this measurement list is, however, not extracted automatically as shown. Instead this list is given explicitly as argument to all DSL functions taking a condition as argument. Automatically extracting the list of measurement names requires parsing of Python expressions. This will be discussed in more detail in Sec. VIII of the paper. Registration of the constraint has the following effect: "*check condition C during the period D; execute reaction R_c if C gets violated; execute reaction R_d if duration D expires; repeat verification on condition failure during the period if F is true*".

The LCS DSLs monitoring functions are all defined using a generic non-blocking function *monitor*, which is based on the functionality of *addConstraint*. A call of the form:

```
monitor(C, R_c, D, R_d, F)
```

invokes *addConstraint(C, R_c, D, R_d, F)*, after first having evaluated the condition to check its immediate value. Note that this function is given a reaction to execute when the condition is violated as well as a reaction to execute when the timer expires. After registration, the function terminates and program execution continues, with the constraint now active in the background.

The specific monitoring functions can now be implemented as follows. The translation of the *assert_constraint* function is straightforward:

```
def assert_constraint(C, R_c, D, F):
  monitor(C, R_c, D, None, F)
```

A constraint will be registered, which checks the condition *C* continuously in the background (relative to the control application thread), and executes the reaction *R_c* if it gets violated. The constraint condition is monitored until the duration *D* expires. No special reaction is executed when the time period expires beyond removing the constraint (this default timeout behavior is built into the *monitor* function).

The *conditional_interrupt* function behaves similarly to *assert_constraint*, but differs in two significant ways:

1) *condition negation*: the condition passed as argument triggers the reaction to be executed when it becomes true (in contrast to false).
2) *interrupt*: the reaction to be executed has to interrupt and stop the main DSL control application while executing.

The first objective is simply achieved by negating the condition that is stored in the constraint. The second objective is achieved by letting the control application thread execute these interrupt reactions, which are stored in a global queue with the function *addInterrupt*:

```
def conditional_interrupt(C, R_c, D, F):
  R_i = lambda: addInterrupt(R_c)
  monitor(negate(C), R_i, D, None, F)
```

As can be observed, the reaction to be executed when the negated condition goes false (i.e., when the original condition goes true) is a reaction that adds the argument reaction to the interrupt queue. No special reaction is executed when the time period expires beyond removing the constraint. Interrupts are now executed by a special function *executeInterrupts()*, which is called as the first statement in all DSL functions. This function iterates through the interrupt queue, and executes them one by one. Note that in this manner only DSL functions (and not the basic Python constructs) can be interrupted, so there may be small delays in seeing the main thread get interrupted, until the main thread begins execution of the next DSL construct in the flow of the application.

Timed interrupts share the interrupt characteristic with conditional interrupts (they stop the main thread), but they are purely triggered by the expiration of a timer. Technically, a timer (Python's library provides timers as threads) is started that triggers after the duration expires and executes in a separate thread a reaction that inserts the interrupt in the interrupt queue. This is achieved with the following function, requesting the reaction *R_d* to get executed as an interrupt after *D* time units:

```
def timed_interrupt(D, R_d):
  R_i = lambda: addInterrupt(R_d)
  monitor(None, None, D, R_i, False)
```

The call of *monitor* has a "*None*" as condition (meaning: no condition), a "*None*" as condition-reaction *R_c*, the duration *D*, and a reaction *R_i* that adds the argument duration-reaction *R_d* to the interrupt queue. The reactivate flag is false, which in fact is irrelevant since no condition is monitored.

The semantics of the *verify_within(C, D, R_d)* construct is to block until the condition *C* becomes true, or until the duration *D* expires, at which point *R_d* gets executed. To support the blocking wait on the condition evaluation (which occurs in a separate thread), a new class *ConstraintStatus* is introduced. Objects of this class serve to establish communication between the caller and the condition evaluator. The class has the following interface:

```
class ConstraintStatus:
  def wait()
  def signalSuccess()
  def signalFailure()
  def satisfied()
```

An object of this class contains a Boolean flag indicating whether the condition has been satisfied or not. The method *signalSuccess* sets this flag to true and signals a semaphore that the *wait* method is waiting on. The *signalFailure* method sets the flag to false and signals the semaphore. The function satisfied thereafter returns true in the first case and false in the second case (the value of the satisfaction flag). To implement the *verify_within* construct, the functions *signalSuccess* and *signalFailure* are passed as arguments to the generic *monitor* function, representing condition reaction and duration reaction, respectively, as follows (note also that the condition is negated to model the "eventually true" check):

```
def verify_within(C, D, R_d):
  cs = ConstraintStatus()
  monitor(negate(C), cs.signalSuccess, D, cs.signalFailure, False)
  cs.wait()
  if cs.satisfied():
    return TrueCode
  else:
    R_d
    return FalseCode
```

The implementation of the *verify_within_voting* function is very similar to that of the *verify_within* function. However, it uses a slightly more complicated variant of the *ConstraintStatus* class with the same public interface. The new class contains two additional counters: the number of conditions that minimally must be satisfied, and the number of candidate conditions that have not yet evaluated true. The class constructor is called with two natural numbers: the number of conditions that minimally must evaluate to true, and the number of initial candidate conditions. The method *signalSuccess* is called when a condition becomes true within its time period and it decrements the number of conditions required to still become true, and also the number of candidate conditions. Similarly, the method *signalFailure* is called when a time period for a condition expires without the condition having become true, in which case only the number of candidate conditions is decremented. The method *satisfied()* returns true if the number of required conditions is non-positive. For each condition-duration pair a constraint is registered which calls *signalSuccess* when the condition becomes true (it monitors the negation being true and reacts when the negation becomes false), and which calls *signalFailure* when the time period expires without the condition having become true.

C. Multi-Threading Issues

As outlined previously, the implementation is multi-threaded due to the fact that constraints, reactions and timers execute in parallel with the main application. This raises a number of issues. One issue is the general notion of program suspension, as required by interrupts, external suspension commands (the *pause()* construct, not discussed in Sec. IV) and hard program termination (normal as well as abnormal). Our approach to interrupts is to let them execute on the main thread as described earlier: the main thread executes queued interrupts. In a single-threaded application, program termination simply means terminating that thread in a safe place, after cleaning up various data structures. If the termination is abnormal and deep inside a nested series of function calls, throwing an exception (after cleanup) may be the most convenient manner in which to terminate the program. In the presence of multi-threading, however, the situation becomes more complicated. When a thread must terminate the entire application, it needs to inform the other threads about this intention. Note that throwing an exception in one thread does not terminate the other threads: exceptions are local to threads. Furthermore, a terminated thread must not just clean up and delete shared data structures, such as the common services. When the main application terminates, what normally happens is that all common services are closed down. However, if there are still threads running (e.g., other applications), one must wait for these to terminate. Design of a more robust and satisfactory long-term solution to this problem was left as future work for the LCS team.

A second issue is data races and deadlocks. As part of a unit testing framework, an algorithm was implemented for detecting cycles in lock graphs from normal (non-deadlocking) runs [10]. This became highly useful for ensuring the absence of deadlock potentials. One particularly interesting deadlock occurred due to the interaction between the Python interpreter and C++. The Python interpreter takes what is called the GIL on the entire virtual machine each time it executes an instruction. The unexpected deadlock scenario was as follows: the Python program executes an instruction (takes the GIL) that attempts to take a lock L on the C++ side; in parallel, a thread on the C++ side takes the same lock L and then makes a call-back of a Python function (a condition evaluator), causing the GIL to be requested. Now the Python thread holds GIL and the C++ thread holds L — a deadlock has occurred. To resolve this potential deadlock problem, additional code was required in the measurement service callback invocation methods to acquire this lock before entering Python, and release it after returning.

A third issue is the removal of constraints. A timer can expire while a constraint reaction is being executed, in which case the timer and the reaction have to agree who removes the constraint. In the current design, a shared state machine keeps track of occurred events and hands removal permission to the right thread.

A fourth issue is a dangling pointer problem that arises when a Python thread passes a Python object reference from the Python data area over to the C++ data area, and then later exits the scope where that object is introduced. In this case, the object gets garbage-collected on the Python side, and the C++ side now holds a dangling pointer (pointing to a garbage-collected data area). This issue was resolved by always making sure on the Python side that such objects were not garbage-collected, by inserting them into a list of "garbage objects" (although certainly not garbage on the C++ side).

VI. Evaluation

An evaluation was performed of the implemented monitor and control system. The monitoring and control system consists of the Python DSL implemented in Python, the underlying common services layer implemented in C++, an information architecture (database) containing persistent monitor and control meta-data, display and GUI software implemented in Java, and the two COTS tools providing the gateway and publish/subscribe middleware functions. In addition, the evaluation also involved writing control applications in the Tabular DSL.

A. Case Study

The monitoring and control system was evaluated by application to two different scenarios for the existing Space Shuttle system, both part of the current Shuttle launch countdown sequence: i) a Liquid Hydrogen Fast Fill application (LH2), and ii) a Main Propulsion System (MPS) pneumatic decay test. A high-fidelity software simulator replaced the real Shuttle end item during this evaluation.

The LH2 scenario consists of monitoring the filling of the Space Shuttle external fuel tank with liquid hydrogen, which is transported from a container positioned in some distance from the Shuttle and connected via a complicated system of pipes, valves, etc. The LH2 process begins at launch minus 6 h and 50 min and lasts for about 40 min, nominally. Its purpose is to fill the shuttle's LH2 external tank from a level of 5% full to 98% full. During that time liquid oxygen is also transported to the Shuttle (called the LO2 process). Sample code representative of a small portion of the LH2 control application is shown in Fig. 5. The MPS process measures the pressure decay and flow rates in the MPS's LH2 and LO2 pneumatic system lines. The MPS application was implemented to demonstrate to the Shuttle operations community that the DSL could be used to implement applications with more algorithmic and numeric computations than the LH2 application.

During the tests, a monitoring display showed the current state of affairs on the (simulated) launch site. Information was provided to this display application (written in Java) via subscriptions on the message bus, and directives from the operators (e.g., initiation of the LH2 application) were input into the LCS system via publication on the message bus. A snapshot of the LH2 display is shown in Fig. 6, showing the liquid hydrogen storage tank on the left, connected to the Space Shuttle external tank on the right.

The DSL applications send sequenced commands, verify their successful execution by checking specific measurements, send messages, and prompts to operators on displays like the one in Fig. 6, monitor measurements for condition violations, and automatically take actions in response to such violations.

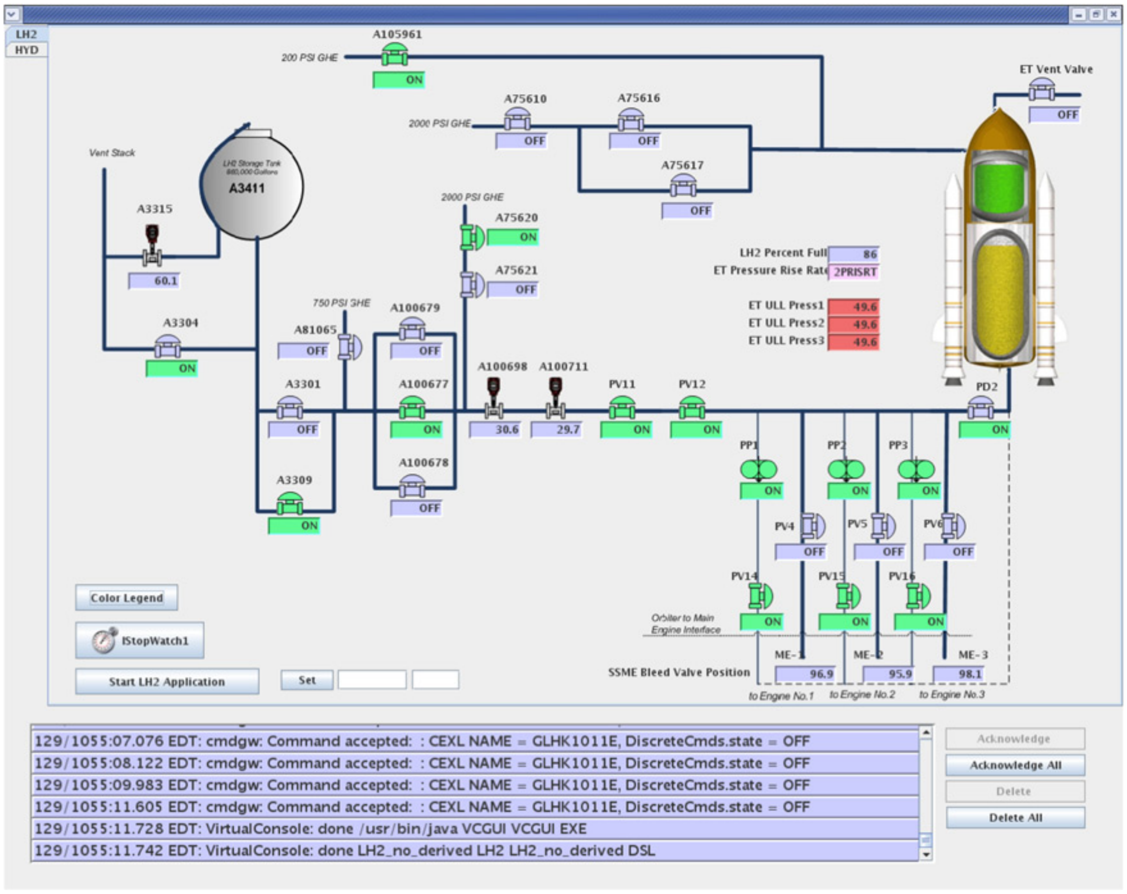
The DSL applications were developed using the Eclipse PyDev development environment [11]. The programs were designed to meet requirements and specifications provided by Shuttle system engineers. The control applications were also reviewed by the system engineers to assess their correctness, completeness, and readability.

B. Result of Evaluation

Implementation of the control applications in the Python DSL was relatively straightforward. Eclipse and the PyDev development environment were useful in quickly turning around code changes and bug fixes. Application

```
# A3301 PRIMARY OPEN COMMAND
send_command( discrete( "VALVE1", "ON" ) )

# TEST INDICATORS
if verify_within_voting ( 2, [lambda : read.INDICATOR1 == OFF, # VLV1 CLOSED #1 IND
                              lambda : read.INDICATOR2 == ON], # VLV1 OPEN #1 IND
                              ["INDICATOR1", "INDICATOR2"],
                              [8, 26], # 8 seconds, 26 seconds
                              DIALOG,
                              "INDICATOR1 and INDICATOR2"
                         ) > 0 :

                         # A FAILURE OCCURRED.  ERROR MESSAGE
                         send_message ( "VALVE1 PRIMARY COMMAND FAILED", LH2, WARNING )

                         # USE SECONDARY COMMAND AND SECONDARY INDICATORS
                         perform( "VLV1_SEC_OPEN", [], BLOCKING)

# SUCCESS MESSAGE
send_message ( "VALVE1 OPENED SUCCESSFULLY", LH2, INFORMATION )
```

Fig. 5 DSL code to open valve and check for appropriate measurements within specified times.



```
129/1055:07.076 EDT: cmdgw: Command accepted:  : CEXL NAME = GLHK1011E, DiscreteCmds.state = OFF
129/1055:08.122 EDT: cmdgw: Command accepted:  : CEXL NAME = GLHK1011E, DiscreteCmds.state = OFF
129/1055:09.983 EDT: cmdgw: Command accepted:  : CEXL NAME = GLHK1011E, DiscreteCmds.state = OFF
129/1055:11.605 EDT: cmdgw: Command accepted:  : CEXL NAME = GLHK1011E, DiscreteCmds.state = OFF
129/1055:11.728 EDT: VirtualConsole: done /usr/bin/java VCGUI VCGUI EXE
129/1055:11.742 EDT: VirtualConsole: done LH2_no_derived LH2 LH2_no_derived DSL
```

Fig. 6 Display for the LH2 application tests.

software developed using the Python DSL successfully executed the LH2 scenario, demonstrating the expected functionality and satisfying the requirements for the proof-of-concept activity.

One significant V&V challenge was that there was no way to actually run and test the application programs without having the full LCS system and simulator up and running, or without intermingling supervisory control code with simple simulation "stubs" within the control application. This highlights the general need for compositional software development and unit testing, where each component can be tested in isolation. It furthermore illustrates the complication resulting from integration of several software artifacts either written in different languages or written by different parties.

Since Python is dynamically typed, the static checkers PyLint [12] and PyChecker [13] were used to statically check Python programs before their execution. None of these checkers, however, perform type checking, and only catch some of the errors that would be otherwise caught in a statically typed programming language. To catch type errors, one could potentially write a specialized static checker for the Python DSL. Python's meta-programming features enable the development of specialized checker tools for Python. The alternative is to use testing techniques, in which case a coverage tool like Coverage [14] seems to be crucial in constructing good test suites, in combination with a unit testing tool like PyUnit [15]. Based on the LCS proof-of-concept team's experience, these are useful tools, and testing the program by running it still provides the best chance of finding type errors in Python. For this purpose, simulators and stubs for other software components in the architecture are of critical importance.

In summary, Python proved to be a convenient and functionally effective language for the purposes of implementing the monitor and control DSL for the presented scenarios. However, its dynamic typing presents a serious challenge from a verification point of view. Further analysis would be required to determine whether development risk can be adequately reduced, perhaps through the use of a specialized IDE that guides application developers and prevents them from making certain classes of errors.

VII. Language Design in Scala

At the time the work described here was carried out (late 2006 – late 2007), Scala [3] was emerging as a new programming language. Scala was first released in 2003, and a second redesigned version was released in 2006 as version 2.0. Since 2006, the language has steadily gained popularity, and is by some considered as a potential replacement for Java. In 2006, however, Scala was too new to be taken into consideration as an implementation language for the LCS DSL. The language, however, appears to have several advantages for DSL definition. This section briefly discusses, as an after-thought, the pros and cons of using Scala (version 2.8.0) for defining the DSL.

Scala combines OO programming and functional programming (FP), including FP concepts such as higher order functions (functions taking functions as arguments, as in Python) and pattern matching over values of what corresponds to algebraic types. Uniformity is achieved by considering all data as objects (with methods). The Scala compiler compiles Scala source code to class files for the Java virtual machine (JVM), and is hence 100% compatible with Java: Java libraries can be used within a Scala program, and Scala can be called from a Java program. Scala is a statically typed language with a sophisticated type system. Types are inferred automatically if not provided by the programmer. Also semicolons (statement separators) are inferred automatically if omitted by the programmer. These two features add to the scripting feel of the language, making it less verbose than Java, and approximately as succinct as Python. Scala's primary concurrency construct is that of an actor. Actors are concurrent processes that communicate by exchanging messages. The implementation of actors is based on Java's concurrency concepts, and hence is more powerful than Python's concurrency model, which only allows the Python VM to execute one instruction at a time. Scala actors can execute in true parallel on multiple cores, whereas Python threads must acquire a lock (the GIL) on the interpreter for each instruction executed, hence preventing instructions from different threads to execute in true parallel on different cores.

Scala provides, as described in [3], all the features Python supports relevant for definition of the DSL, such as higher order functions and default arguments (introduced in Scala version 2.8.0). However, in Scala the body of a lambda abstraction can contain any number of statements and expressions. This is in contrast to a lambda abstraction in Python, the body of which can only contain one expression and no statements. In addition to this, Scala offers some features that turn out to further ease definition of DSLs:

- *Call-by-name*, also referred to as automatic closure construction. This permits omission of lambdas in lambda abstractions. That is, generally it is possible to apply a function (appropriately typed, as we shall see) to an expression or statement with call-by-name semantics, meaning that the expression or statement is only evaluated inside the function when referred to by name.

- *Curried functions*, allowing arguments to be naturally separated without having to appear in a comma-separated argument list.
- *Overloaded functions*, allowing minor variants of a DSL construct to have the same name (for major variants one might want to have different names). Function overloading to some extent can be used to avoid default arguments, and thereby a comma-separated list of arguments.
- *Infix notation for method calls*, permitting definition of DSL constructs that appear as built-in language constructs.
- *Implicit conversions*, through definition of so-called *implicit* functions that convert values of one type to values of another type. Such functions are applied automatically in places where it will correct a type problem.

In the following discussion, these concepts will be demonstrated, using the *verify_within* construct as an example. Several approaches will be demonstrated illustrating the different DSL supporting features of Scala. First, however, it will be demonstrated how measurements can be modeled, allowing for maximal notational inconvenience.

A. Measurements

Measurements are defined in a class similar to the Python equivalent:

```
class Measurement(id : Symbol, value : Int) {
  private val time : Long = ...
  def getId() : Symbol = id
  def getValue() : Int = value
  def getTime() : Long = time
}
```

A main difference is that values and methods are now typed. Measurement ids are represented as symbols in the *Symbol* data type, which contains quoted identifiers such as `'Msmt2`. Symbols are slightly more convenient to write than their string counterparts: `"Msmt2"`. Second, measurements are published in a mapping from symbols to measurements in the measurement service object *ms* (we only show the methods):

```
object ms {
  ...
  def publish(name : Symbol, value : Int) = {...}
  def getByName(name : Symbol) : Measurement = {...}
  ...
}
```

Without any additional definitions, a measurement would be accessed as follows:

```
ms.getByName('Msmt2).getValue() > 0
```

or, if this is defined as function named *read*:

```
def read(s : Symbol) = {
  ms.getByName(s).getValue()
}
```

we can write:

```
read('Msmt2) > 0
```

To avoid such still slightly heavy notation, two implicit conversion functions are defined, one from symbols to integers and one from symbols to measurements.

```
implicit def conv1(s : Symbol) : Int = {ms.getByName(s).getValue()}
implicit def conv2(s : Symbol) : Measurement = {ms.getByName(s)}
```

The names of these functions are unimportant since they will be applied by the Scala compiler under the hood to make expressions type check. That is, these functions allow us to write expressions such as:

```
'Msmt2 > 0
```

Scala will automatically apply the appropriate conversion function (*conv1*) to obtain an integer from `'Msmt2`. Hence the expression is equivalent to:

```
conv1('Msmt2) > 0
```

Furthermore, with a Scala value declaration (defining a constant) of the form:

```
val Msmt2 = 'Msmt2
```

it is possible to now refer to `Msmt2` without quotes, as in:

```
Msmt2 > 0
```

This is to be compared to the notation needed in Python:

```
read.Msmt2 > 0
```

or alternatively:

```
getByName("Msmt2") > 0
```

Derived values can be defined as follows using Scala's `def` construct:

```
def DMsmt = Msmt2 * 100
```

Each time `DMsmt` is now referenced, the expression `Msmt2 * 100` is evaluated according to the rules already described, including application of an implicit conversion function. The corresponding Python notation was:

```
derive(DMsmt, lambda: read.Msmt2 * 100)
```

Implicit conversion from symbols to integers in addition gives us all the operators on integers to work on measurements, without having to define them as methods in the *Measurement* class, as we did in Python.

B. verify_within construct

In the following, we shall present three approaches to define the *verify_within* construct, illustrating various techniques for defining DSLs available in Scala. Recall the signature of the *verify_within* function:

```
verify_within(C, D, [R], [S])
```

The function takes as the third argument a reaction (optional). Since Python is not statically typed we can just pass either a *DIALOG* constant or a lambda abstraction. In Scala, we have to declare the type of a reaction explicitly. This can be done as follows using Scala's notation for what is often referred to as *algebraic types* in FP languages, namely *case classes*:

```
abstract class Reaction
  case class CODE(code : () => Unit) extends Reaction
  case object DIALOG extends Reaction
```

The first definition introduces an abstract class (type) *Reaction*. The next two definitions introduce the sub-classes *CODE* and *DIALOG*. The *CODE* subclass is parameterized with a function (formal parameter name is *code*), which has the type: "`() => Unit`". In general, the type of a function is represented as $T_1 \Rightarrow T_2$, indicating that the function takes arguments of type T_1 and returns results of type T_2, and potentially also having side effects on variables in scope. In this case, the formal parameter *code* is required to be a function that when applied to the value `()` returns a value in type `Unit`. The `Unit` type contains the single value "`()`", and corresponds to the `void` type in languages such as Java and C. Hence, this is a function that will be executed only for its side effects.

Assume that f is a function of type "`() => Unit`", that is: a function taking no arguments and returning no result, used purely for its side effect. Then the following values are members of the type *Reaction*: `CODE(`f`)` and `DIALOG`. In other words, `CODE` and `DIALOG` are the value generators of the *Reaction* type.

C. verify_within construct : Solution 1

We can define a function with the following signature:

```
def verify_within(cond : () => Boolean, time : Int,
reaction : Reaction = DIALOG, name : String = "") = {...}
```

This definition looks much like its corresponding definition in Python, except for the added type information. With such a definition we would be able to write the following two calls of *verify_within*:

```
verify_within(() => Msmt2 > 0, 10, DIALOG)
verify_within(() => Msmt2 > 0, 10, CODE(() => println("Msmt2 violation")))
```

Note the required application of the `CODE(...)` value generator. If, however, we define an implicit conversion function:

```
implicit def conv3(code : () => Unit) : Reaction = {new CODE(code)}
```

We can now, instead of the last call of *verify_within*, write:

```
verify_within(() => Msmt2 > 0, 10, () => println("Msmt2 violation"))
```

This now has the same appearance as the Python code, but has the advantage of being statically typed.

D. verify_within construct : Solution 2

In the second version, we define a function that is curried, and which uses a call-by-name formal parameter for the condition:

```
def verify_within(cond : => Boolean)(time : Int)(reaction : Reaction) = {
  ...
}
```

Function currying is a technique for transforming a function that takes multiple arguments, as in $f(x, y, z)$, into a function f_c, that takes a single argument, and returns a function that applies to the rest of the arguments, possible also in a curried manner: $f_c(x)(y)(z)$. In this case (ignoring the property name argument), the *verify_within* function is transformed into a curried function that takes its arguments one at a time. This will make applications of the function more readable as we shall see.

The first parameter to the function, the condition, has the type: "`=> Boolean`", representing a call-by-name Boolean parameter type. When applying the function to a Boolean expression, this expression is not evaluated until it is referred to inside the definition of *verify_within*. For example, we can write:

```
verify_within (Msmt2 > 0) ...
```

without the expression `Msmt2 > 0` being evaluated at call time. In addition, we can define an implicit conversion function from call-by-name statements to reactions, which just inserts a lambda abstraction "`() =>`":

```
implicit def conv4(code : => Unit) : Reaction = {new CODE(() => code)}
```

With these definitions we can now write:

```
verify_within (Msmt2 > 0) (10) (DIALOG)
```

and:

```
verify_within (Msmt2 > 0) (10) {
  println("Msmt2 violation")
}
```

In this last call, the implicit conversion function `conv4` is applied to the last argument. This notation appears more user-friendly than solution 1. The solution is also safe in the sense that should one forget some of the arguments, as in:

```
verify_within (Msmt2 > 0) (10)
```

The compiler will emit the error message: "missing arguments for method *verify_within*". It is possible to omit arguments to a curried function if the call appears in a context where the type of the partial application matches or if the call is followed by the symbol "_". This is, however, not the case when the call occurs as a general statement.

E. verify_within construct : Solution 3

The final solution consists of attempting to give a programming language feel to the syntax. Instead of the preceding solution we would want to write:

```
verify (Msmt2 > 0) within 10 onfail DIALOG
```

and:

```
verify (Msmt2 > 0) within 10 onfail {
  println("Msmt2 violation")
}
```

Consider that such a construct were to be defined as grammar with a non-terminal for each phrase starting with a keyword, using $\langle N \rangle$ to indicate a non-terminal N and [...] to indicate optional:

$$\begin{aligned}
\langle Verify \rangle &\leftarrow \textbf{verify}\ [\langle Name \rangle]\ \langle Cond \rangle\ \langle Within \rangle \\
\langle Within \rangle &\leftarrow \textbf{within}\ \langle Int \rangle\ \langle OnFail \rangle \\
\langle OnFail \rangle &\leftarrow \textbf{onfail}\ \langle Reaction \rangle
\end{aligned}$$

This grammar structure can be emulated in an OO language, as illustrated by the following Scala object, which defines the grammar and its semantics described earlier:

```
object Verify {
  def verify(cond : => Boolean) : Within = {new Within("", cond)}
  def verify(name : String)(cond : => Boolean) : Within =
    {new Within(name, cond)}

  protected class Within(name : String, cond : => Boolean) {
    def within(time : Int) : OnFail =  {new OnFail(name, cond, time)}
  }

  protected class OnFail(name : String, cond : => Boolean, time : Int) {
    def onfail(reaction : Reaction) {/* implementation of DSL construct*/}
  }

  implicit def conv5(code : => Unit) : Reaction = {new CODE(() => code)}
}
```

The object defines two overloaded *verify* methods, corresponding to the optional nature of the property name, one method not taking a name as argument and one taking a name. Each of these methods return an object of the nested class *Within*. The *Within* class itself defines a *within* method, which when applied to a time value returns an object of the class *OnFail*. This class finally defines the *onfail* method, which at this point has access to all information (name, condition, time, and reaction), and hence can execute the semantics of this construct. The implicit conversion method converts code to reactions. With this definition we can write:

```
verify(Msmt2 > 0).within(10).onfail({
  println("Msmt2 violation")
})
```

However, Scala allows method calls using infix notation. That is, given an object o and a method m in o, instead of writing $o.m(a)$ as in traditional OO languages, it is possible to write: "$o\ m\ a$", omitting the dot (".") and the parentheses around the argument. We can therefore write:

```
verify (Msmt2 > 0) within 10 onfail {
  println("Msmt2 violation")
}
```

The construct now appears like any other programming construct and from a DSL design point of view looks ideal. This approach, however, has a couple of problems associated with it that make programming unsafe in the current version of Scala (version 2.8.0). First, it is, for example, possible to write only part of the construct, as in:

```
verify (Msmt2 > 0) within 10
```

leaving out the `onfail {...}` part. The compiler will not complain. Furthermore, if no special runtime checking is performed to detect this (for example, by ensuring that all begun constructs are ended before a new is begun), no warnings will be issued during runtime, either. However, a solution for this problem might be implemented in a future version of the Scala compiler, as communicated by Scala's designer Martin Odersky[††]. An analysis will be provided for checking whether a function call has side effects or not. Odersky[††] suggests that with such an analysis it is possible to disallow pure expressions as statements, essentially disallowing the partially instantiated DSL construct described earlier. Another problem is due to Scala's semicolon inference. It is not possible to write for example:

```
verify (Msmt2 > 0) within 10
onfail {
  println("Msmt2 violation")
}
```

where the `onfail` "keyword" has been moved to a line for itself. The compiler will infer a semicolon after the first line, and will subsequently not be able to associate the name `onfail` with the method defined in the *OnFail* object which is the result of the first line. There is no obvious solution to this problem beyond avoiding such line breaks. Note, however, that it is possible to write:

```
verify (Msmt2 > 0) within 10 onfail {
  println("Msmt2 violation")
}
```

F. Analysis of DSL Approaches

Earlier we have seen three approaches defining a DSL construct in Scala. The first approach corresponds to the way a DSL is defined in Python. The notation is just as succinct as in Python, and more succinct than it would be in Java due to Java's lack of function values. As an added advantage, in contrast to the Python solution, the Scala solution is statically typed, which will prevent many programmer mistakes. The second approach, using overloading, currying and call-by-name, yields a solution that from a notational point of view is even more succinct and clear. This solution is still as safe as the previous solution with respect to the compiler being able to detect programming errors. The third approach, on the other hand, was demonstrated unsafe in the current version 2.8.0 of Scala but might become safe in future versions.

Scala seems to have qualities that make it a good platform for DSL development compared to the combination of Python and C++ on the one hand and Java on the other hand. It offers language concepts that make it possible

[††] Personal email communication with Scala's designer Martin Odersky, 5 Jan. 2010.

to construct a DSL optimized for ease of use. It is statically typed, with a notationally succinct flavor comparable with a scripting language like Python (amongst other things due to type and semicolon inference). It is compatible with Java, which would make it easy to integrate with other parts of a system, for example, the message bus. It has a concurrency model more appropriate for the DSL than Python's, which only allows one thread to execute at any point in time, preventing utilization of multi-core machines.

However, Scala is a sophisticated language, so one can debate whether systems engineers (usually not programmers) should program in it, even if only using a subset of the language.

VIII. Discussion

Python is clearly an easy-to-learn and convenient language, due to its succinct syntax, abstract constructs and extensive library, including meta-programming features. It supports OO as well as a limited form of FP. In particular, the special lambda abstraction allows one to write anonymous (nameless) functions, which are convenient when implementing a DSL. Python provided a powerful tool for the short-fuse and resource-constrained LCS proof-of-concept activity. In addition, the Boost.Python interface library proved to be extremely useful for implementing the interfaces between Python and the C++ messaging frameworks.

The LH2 example application demonstrated that the required control idioms could be implemented in a programming language interface to the new distributed LCS architecture. The program was able to express both the sequential logic needed to implement test procedures (issuing commands, and verifying responses through sensor measurements), and the reactive logic needed to implement safety constraints. It further demonstrated the use of multiple programming languages specialized to support the implementation of the specific control idioms, and user interfaces.

A main issue of concern is the lack of static typing in the language. Python is dynamically typed, meaning that many kinds of errors are not caught at compilation time. There do exist static checker tools for Python, such as PyChecker and Pylint, but they do not seem to catch quite fundamental problems, such as, for example, a wrong number of arguments in a function call. The Tabular DSL demonstrated some mitigation of these risks. Although the risks were not entirely eliminated in the proof-of-concept task, it did demonstrate that tools were available to support the implementation of this verification at several points in the process, including in the language, or in the integrated development environment, or both. It is clear, however, that the use of a statically typed language such as Scala would improve safety as well as notational convenience.

The second issue of concern identified during the initial DSL survey was the uncertainty associated with the ability of Python to satisfy performance requirements. The ability of the applications to meet real-time deadlines—primarily, the latency of reactions to changes in sensor values—will depend to a large extent on the latency of the delivery of the measurements to a gateway, and from there across the message bus to the monitoring application. That is to say, the performance issue is more of an architectural concern for LCS, considering the adoption of a distributed architecture. With this said, the language allows applications to react to measurement changes as they arrive via the callback interface; there will be a maximum rate at which these update events can be delivered and processed that will depend on how many conditional expressions the application uses. It is not expected that the post-proof-of-concept DSL and common services implementation would represent a performance bottleneck in the LCS system, but formal benchmarking remains to be performed. It is noted that a Scala implementation would have better performance due to its rooting in the JVM. Scala is even potentially faster than Java due to its strong static typing, which allows dynamic checks to be removed.

One of the decision points in the design of the DSL API was whether it should have an OO flavor in the spirit of Python, or whether it should have a more classical procedural flavor like the C programming language. Python (as well as Scala) allows both forms. In the OO approach, the DSL programmer would have to be aware of classes and objects and would work with dot-notation. In the procedural approach, a program consists of a sequence of function definitions. It was decided that the OO of Python should be downplayed in order to minimize the number of Python concepts a system engineer would have to master: stand-alone functions resemble the syntactic constructs in familiar programming languages, and may also lead to more succinct control application implementations. This decision was largely influenced by the desire for the new language to reflect the capabilities of the GOAL language used as a requirements model, and to make the language intuitive to the system engineers who would be using it. The resulting syntax, however, exhibits idioms of both programming styles.

Several of the Python constructs take as parameters a lambda abstraction (or a function reference) as condition and are supposed to evaluate this condition each time a variable occurring in that expression (or referenced by that function) is updated. This concerns the constructs *verify_within*, *verify_within_voting*, *assert_constraint*, and *conditional_interrupt*. These constructs cause update handlers to be registered with common services and take measurement name lists as explicit arguments so that update handlers can be registered for those measurement names. These measurement name list arguments are an annoyance and would preferably be inferred automatically from the conditions themselves. This would require parsing and extraction of variable names from the condition expressions (lambda abstractions or functions). This could be non-trivial if the conditions contain nested function calls, leading to arbitrarily "deep" Python code, in which case a free variable analysis of all of Python is required. There are currently no obvious simple solutions to this problem. Alternatively, one could specify a way of writing conditions where a condition is not a general Python expression. For now, the design assumes that the user specifies the list of measurements, any update of which should trigger the check of the condition. The problem is the same for Scala.

There is currently an asymmetry in the design of the DSL from a linguistic point of view. For example, consider the blocking function *verify_within* and the non-blocking *assert_constraint*: *verify_within* checks that a condition becomes true within a time period, whereas *assert_constraint* checks that a condition stays true during a time period. To be completely symmetric, the language would need to offer additional constructs, such as a blocking *verify_during* function, which checks that a condition stays true during a time period, and a non-blocking *assert_constraint_within* that checks that a condition becomes true within a time period. The prototype DSL design was driven by requirements provided by experts at KSC in a very direct manner, hence the asymmetric design. Such a requirements-driven design might subsequently be followed by a "*linguistic cleansing*" procedure, introducing symmetry as appropriate, and providing other desirable language characteristics.

In general, the DSL can be regarded as a temporal logic embedded into an imperative programming language, a concept that can be given further theoretical attention. The *monitor* function described in Sec. V in fact represents a very general temporal operator that could be embedded into any programming language.

A final issue to be discussed is the notion of time. In the current DSL implementation, time is measured on the clock local to the control application. An alternative would be to measure time in terms of time stamps in measurements, or some other global time-keeping mechanism. This issue is of particular importance given the distributed nature of the LCS architecture.

IX. Conclusions

Experience with the current Space Shuttle test and checkout system suggests that significant lifecycle costs are associated with the development and maintenance of software for monitor and control applications. Particularly expensive is the process by which system engineers express requirements for test procedures in prose, software developers translate these requirements into code, and then both sets of experts are engaged in verification of the resulting application's correctness. The Constellation LCS will have to significantly reduce these lifecycle costs while assuring a consistent high level of safety and security.

To this end, the LCS project has completed a proof-of-concept demonstration of an approach that has the potential to ultimately lower the lifecycle costs of monitor and control applications for launch processing, test, and checkout. This paper has presented one of the important contributions from this proof-of-concept activity, namely the homegrown Python DSL and an associated common services framework design.

In summary, the use of Python as the base language for the homegrown LCS DSL was successful. The prototype demonstrated that the implemented DSL design could satisfy the requirements of a realistic monitor and control problem, working effectively with the other elements of the distributed LCS architecture. The KSC system engineers readily understood the DSL logic, supporting its development throughout the prototype. Along with Python's strengths as a popular, well-supported language, the Python-based DSL seems to live up to its promise as a low-cost solution.

However, given the paramount importance of safety and reliability, Python's weak static typing remains a concern. Also, the DSL is not optimal, from a succinctness and ease-of-use perspective. Specifically, the need to write lambda abstractions and long comma-separated argument lists to function calls seems unpleasant. For this reason, as a post-project experiment, Scala was studied as an alternative DSL implementation language. The use of Scala would solve several problems with the Python implementation, and would better satisfy the desired safety, syntactic simplicity, and scalability requirements.

Acknowledgments

The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The authors would like to acknowledge the contributions of the rest of the LCS Proof-of-Concept team at NASA Kennedy Space Center.

References

[1] The White House, Office of the President, "A Renewed Spirit of Discovery: The President's Vision for U.S. Space Exploration," Released to accompany the President's NASA FY 2005 Budget, Jan. 2004, web site: http://www.nasa.gov/pdf/55583main_vision_space_exploration2.pdf.

[2] Python web site: http://python.org.

[3] Scala web site: http://www.scala-lang.org.

[4] Bennett, M., Borgen, R., Havelund, K., Ingham, M., and Wagner, D., "Development of a Prototype Domain-Specific Language for Monitor and Control Systems," *Proceedings of the 2008 IEEE Aerospace Conference*, May 2008.

[5] Mitchell, T., "A Standard Test Language—GOAL (Ground Operations Aerospace Language)," *Proceedings of the 10th Workshop on Design Automation at the Annual ACM IEEE Design Automation Conference*, pp. 87–96, 1973.

[6] Object Management Group, Inc., "Data Distribution Service for Real-time Systems, v1.2," Jan. 2007, web site: http://www.omg.org/technology/documents/formal/data_distribution.htm.

[7] Leucht, K. W., and Semmel, G. S., "Automated Translation of Safety Critical Application Software Specifications into PLC Ladder Logic," *Proceedings of the 2008 IEEE Aerospace Conference*, May 2008.

[8] Boost web site: http://boost.org.

[9] Boost.Python web site: http://boost.org/libs/python/doc/index.html.

[10] Harrow, J., "Runtime Checking of Multithreaded Applications with Visual Threads," *SPIN Model Checking and Software Verification, Proceedings of the 7th International SPIN Workshop*, Lecture Notes in Computer Science Vol. 1885, Springer, Berlin, pp. 331-342, 2000.

[11] PyDev web site: http://pydev.org.

[12] Pylint web site: http://www.logilab.org/857.

[13] PyChecker web site: http://pychecker.sourceforge.net.

[14] Coverage web site: http://nedbatchelder.com/code/coverage.

[15] PyUnit web site: http://pyunit.sourceforge.net.

Shlomi Dolev
Associate Editor