NASA's Exploration Agenda and Capability Engineering

Daniel E. Cooke
Texas Tech University

Matt Barry
Jet Propulsion Laboratory

Michael Lowry
NASA Ames Research Center

Cordell Green
Kestrel Institute

NASA is using model-based languages and risk analysis methodologies to raise software development to the level of hardware development. Ultimately, it hopes to achieve a fusion of systems and software engineering by replacing conventional software development techniques with capability engineering, which focuses on a system's full set of functionalities.

Many NASA flight systems have been developed according to an approach that defines hardware first, then effectively retrofits software and human procedures to the hardware systems. Throughout this process, a careful allocation of functionality handles time-sensitive operations onboard, while mission controllers on the ground handle non-time-sensitive operations.

Past missions have been scripted, meaning that engineers developed hardware and software systems to respond mainly to foreseen circumstances, making them less able to handle unforeseen problems or exploration opportunities. The primary responsibility for handling unanticipated situations resided with humans, either onboard the spacecraft or in the mission control center. Clearly, there are times when even time-sensitive concerns are handled on the ground because of the more substantial human and computer resources available to address complex problems.

All the software support for onboard and mission control must be highly dependable, requiring rigorous validation and verification. The US Department of Defense recently presented data showing that the percentage of aircraft functionality provided by software increased from 8 percent for the F-4 in 1960 to 80 percent for the F-22 in 2000.[1]

Although software technologies supporting the development of onboard systems have improved significantly in the past 45 years, whether improvements have kept pace with the growing demand and complexity that result from the increasing software functionality remains arguable. Following current practices, it can take years or months to make software modifications to space-based systems at significant cost. This remains true even though the personnel who develop and maintain International Space Station and Orbiter software have discovered ingenious ways to reduce this time and cost.

FUTURE MISSIONS

Missions that NASA's Exploration Systems Mission Directorate currently is planning will be less predictable than past missions. In addition, they often will be of much longer duration and involve considerably greater distances. Minute-by-minute communication between Mars and mission control will be physically impossible given round-trip communication delays ranging from

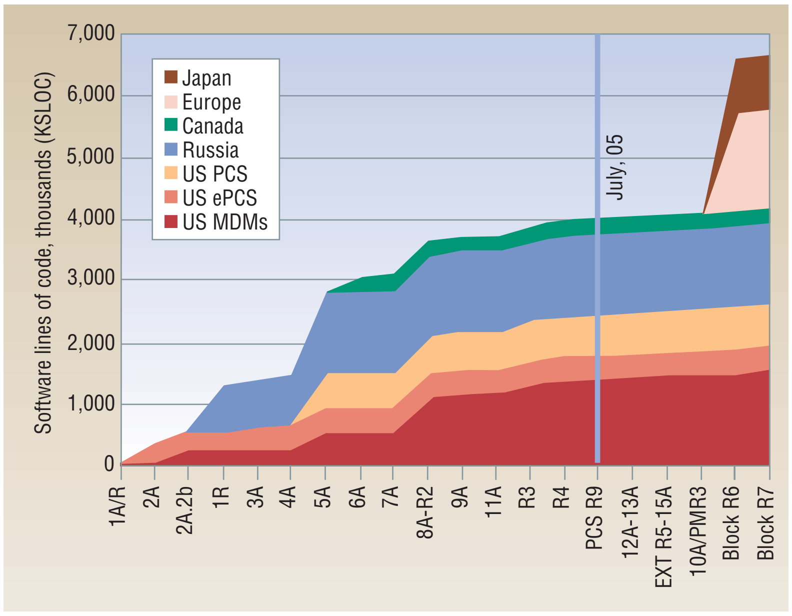


Figure 1. Growth in International Space Station software. Since 1998, more than 4 million source lines of code (SLOC) have been added to the ISS onboard systems. International partners have provided half of the new content. Although the new content developed outside the US does not figure into the US portion of the ISS program costs, integrating the new content from non-US partners does. Current projections indicate that another 3 million SLOC will be developed by international partners and integrated by the US ISS offices.

six to 41 minutes, not counting a solar conjunction between Mars and Earth during which communication might be impossible for up to three weeks.

Communication delays could drastically impact the allocation of functionality in future missions. Some of the intellectual safety net that mission controllers and their computers currently embody will need to accompany the astronauts onboard. For example, more intelligent software that assists the astronauts in discovering workarounds when systems fail will help in developing quick onboard solutions to unforeseen problems or in taking advantage of unforeseen exploration opportunities.

Because of increased software functionality, hardware developed for future missions will likely be of greater utility. Thus, systems will be more adaptable for varied mission scenarios. However, modifying capabilities between and during missions will require revolutionary software development approaches.

These observations led to the primary conclusion reached at a NASA Software Engineering Technology Workshop held in Houston in April 2004. Participants in this workshop concluded that model-based software development—or *capability engineering*—must replace conventional software development techniques. In this approach, engineers and programmers provide direct, declarative descriptions of problem solutions and modeled systems rather than developing instructions that implement these algorithms and models.

In its purest form, capability engineering focuses on a system's full set of capabilities or functionalities, independent of whether humans, software, or hardware support the capabilities. Clearly, this changes the engineers' view: They declare capabilities, study alternative capabilities, and perform risk analysis at a functional level, free from the burden of hand-coding the correct program statements or human procedures.

ISS SOFTWARE BASELINE REVIEW

Although the International Space Station does not involve the long distances planned for future missions, it is at the very least a long-duration mission. In addition to correcting and modifying original capabilities, the ISS software support team is continually adding new components as well as testing and integrating software provided by our international partners. Figure 1 shows the growth in ISS software since 1998.

The ISS software team might have been ahead of the curve when it comes to capability engineering and intelligent software design. The blending of systems engineering and software engineering figures prominently in the overall design of ISS systems. From the systems engineering viewpoint, a careful mapping of data elements from sensors to effectors helps manage every piece of data. Currently, there are 250,000 named and managed data elements.

ISS architecture

Figure 2 outlines the software architecture approach taken in developing the ISS onboard systems. The architecture isolates the more fundamental onboard capabilities from the display systems, and the displays from the crew's laptops. These laptops, called Station Support Computers, provide task procedures, e-mail access, and other office capabilities for each crew member. These are basically noncritical systems running Microsoft Windows.

Tier I of the architecture provides interfaces for command-and-nontrol mission controllers on the ground and the crew interfaces to onboard computer software. Personal computer systems for crew interfaces provide a separate set of laptops that supply 6,000 display interfaces for the ISS crew. These displays were developed using a high-level interface specification language. The isolation of the interface software from the onboard systems separates a significant portion of the changes to systems and indicates a careful analysis of information flow throughout the system—mirroring the systems engineering efforts to manage control data from sensors to effectors.

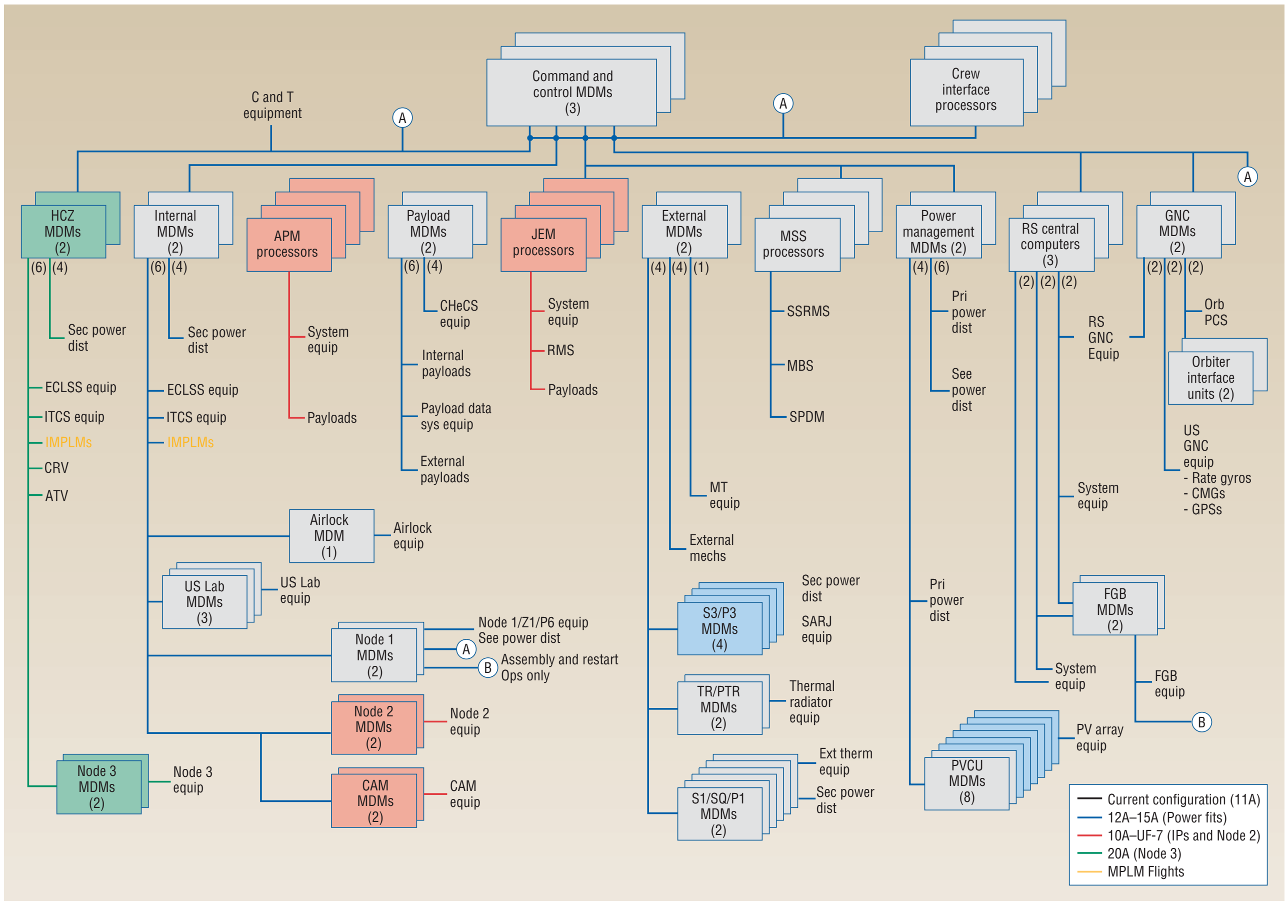


Figure 2. Software architecture outline for ISS onboard systems. The architecture isolates the more fundamental onboard capabilities from the display systems and the displays from the crew's laptops.

Tier II identifies the major onboard systems, written primarily in Ada. They include

- *internal systems* to monitor and control the cabin environment;
- *external systems* to monitor and control the robotic arm and a mobile transporter that moves items along the station's external spine;
- *power management* to monitor and control power distribution and provide fault detection and correction;
- *guidance, navigation, and control (GN&C) systems* to support navigation, attitude determination, and control;
- *payload management* to provide support for managing, controlling, and coordinating onboard payloads; and
- *hub control* to support command and data handling and control of the node 3 caution and warning system, secondary electrical power system, remote power controller modules, common berthing mechanisms, the thermal control system's active and passive thermal functions, and environmental control and life support.

Tier III provides component-level software supporting the Tier II system elements. These multiplexer/demultiplexer radiation-hardened 386 microprocessors manage the data paths from onboard sensors and effectors. All told, the Tier II and III software runs into the neighborhood of 7 million source lines of code (SLOC).

Its complexity, combined with its criticality, translates into a daunting effort to maintain and sustain an ever-changing system. That requirement formulation is not reflected in the software development costs offers more evidence that the ISS approach has effectively integrated systems and software engineering.

ISS change process and costs

Software change requests (SCRs) initiate the three types of ISS software functionality changes.

One form of change involves the addition of new functionality, including changes required when adding new station modules. Another change source involves making corrections due to code nonperformance—that is, when requirements are not met. Both of these change types result in the actual modification of source code.

The third type of change is based on flexibility designed into the system in the form of pre-position loads. The 1,473 PPLs serve as parameters that software developers can use to tweak performance and modify some forms of functionality. Handling changes with the PPLs reduces the need for traditional modifications that require code changes.

Since deployment of the onboard ISS software, an estimated 30,000 SCRs have been processed. All told, roughly 865 software change requests were processed last year, costing $47 million in development costs and $21 million in software integration costs. Thus, on average, a software change, including a PPL, costs around $78,000. This amount does not cover the SLOC produced by international partners—or, more precisely, it includes the cost of integration but not the cost of producing the new content.

Production-quality program synthesis is the keystone for full life-cycle model-based programming.

ISS EXPERIENCE WITH MODEL-BASED PROGRAMMING

Aerospace applications use model-based prototyping extensively. In particular, developers often prototype control applications in commercial environments such as Matlab/ Simulink, then validate them in simulation environments. These model-based prototyping environments enjoy widespread use in part because they provide an intuitive language interface for aerospace engineers. This interface is often a block-and-arrow visual input format similar to traditional block-diagram notations used in control and systems engineering.

Production-quality program synthesis is the keystone for full life-cycle model-based programming. Without this capability, model-based programming is limited to being a prototyping tool whose utility ends after detailed design, when the production code is developed manually. Further, any subsequent upgrade or maintenance fix must be made manually, directly on the software.

In our experience, developers seldom make the additional effort to maintain the prototype. They are reluctant to retain more than one artifact during maintenance upgrades; even maintaining documentation in synchronization with the executable code is time-consuming.

Some deployed aerospace software systems, including a portion of the ISS's GN&C software, have used program synthesis, also called *autocoding*. ISS programmers report that the experience with the current generation of commercial autocoding technology has been mixed. During the ISS software development, engineers cordoned off two portions of the ISS software for an autocoder in hopes of saving time and money: attitude control, specifically pointing, control, and guidance-monitoring of the gyros; and truss software, including power.

For these functions, GN&C-knowledgeable ISS engineers developed and maintained the code in Ada. In this case, autocoding seemed like the dream solution, potentially reducing staff and development time. Unfortunately, technology limitations kept autocoding from meeting expectations.

For the front-end design, the selected commercial autocoder had benefits similar to Matlab, such as intuitive input notation for control engineers. However, the quality of the generated code had limitations that technology being developed in research laboratories at NASA and elsewhere might some day overcome.

Limitations in the autocoder-generated software caused particular difficulties in the subsequent maintenance phase, which is still ongoing for ISS. These problems include the following:

- Engineers found it difficult to navigate through the generated code from one hierarchical level to the next to find a function specified in the input. Due to the deep nesting, they even struggled to determine the code's inputs and outputs.
- Maintenance proved particularly difficult. Estimated time for updating code for the autocoder was at least a factor of five compared to updating functions manually coded in Ada. Adding comments to the code was also difficult, which further hampered maintenance.
- The autocoder produced voluminous nonoptimized code—as much as three times the number of source lines—compared to manually developed code. The code expansion factor poses a particular problem in space applications. Making commercial computers robust for radiation tolerance takes five years or more; it also typically makes them an order of magnitude less powerful in terms of speed and memory than the workstations on which developers prototype the software.
- The initial concern that autocoding's opacity would make it difficult to achieve a human rating of the software, which is in essence a tolerance for two successive faults, did not turn out to be a limiting factor except for the understanding and maintenance issues.

Model-based approaches have improved significantly since the ISS software's initial development several years ago, and software architecture and design are now much better understood. Although NASA and aerospace researchers are often at the forefront in developing advanced software technologies, there remains a healthy and probably appropriate tension between the desire to take advantage of new technologies on aircraft or spacecraft and the time- and safety-critical requirements of the onboard systems. Once space-based systems are deployed and trust in them is gained, wholesale changes to them often become infeasible.

Developing a cautiously aggressive approach to the use of model-based software development in the deployment of onboard systems remains an important goal. Only through sound and assured approaches that permit the rapid change of software will NASA be able to modify capabilities to address a potentially rapidly changing exploration environment.

FUTURE MODEL-BASED APPROACHES

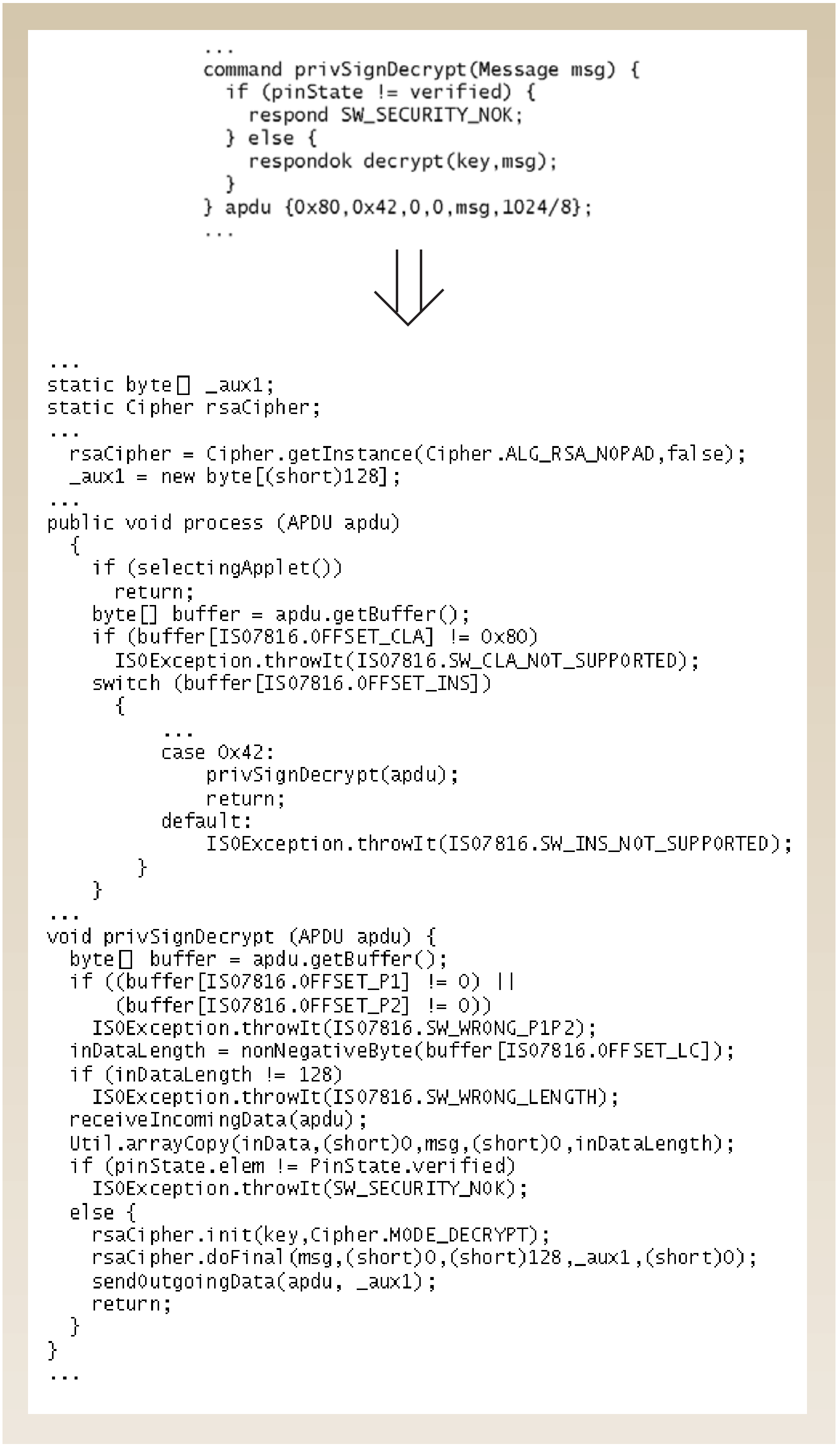
Rapidly and dependably modifying onboard capabilities for future missions requires fully developed model-based development approaches. Engineers can use the approaches that NASA is currently studying and testing to prototype and develop less time-critical systems on the ground and onboard.

In the domain-specific approach to model-based software development, domain-specific languages play a crucial role. Two examples help illustrate this approach: the AutoSmart system for developing Java Card applications using the SmartSlang domain-specific language (www.kestrel.edu/jcapplets)[2,3] and a domain-specific approach to flight-software development based on the state analysis modeling approach developed by JPL using the Mission Data Language (MDL), a domain-specific language for formalizing state analysis.

AutoSmart

This effort includes a code generator that produces code along with a proof that the code is correct. Researchers are now leveraging the technology developed for AutoSmart and applying it to flight software.

Developed by Alessandro Coglio and colleagues at the Kestrel Institute, AutoSmart illustrates a very high-assurance, semantic, and proof-oriented approach to model-based software generation. At the AutoSmart system's heart is SmartSlang, a domain-specific language for smart card applications.



```
...
command privSignDecrypt(Message msg) {
  if (pinState != verified) {
    respond SW_SECURITY_NOK;
  } else {
    respondok decrypt(key,msg);
  }
} apdu {0x80,0x42,0,0,msg,1024/8};
...
```

```
...
static byte[] _aux1;
static Cipher rsaCipher;
...
  rsaCipher = Cipher.getInstance(Cipher.ALG_RSA_NOPAD,false);
  _aux1 = new byte[(short)128];
...
public void process (APDU apdu)
  {
    if (selectingApplet())
      return;
    byte[] buffer = apdu.getBuffer();
    if (buffer[ISO7816.OFFSET_CLA] != 0x80)
      ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
    switch (buffer[ISO7816.OFFSET_INS])
      {
        ...
        case 0x42:
            privSignDecrypt(apdu);
            return;
        default:
            ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
      }
    }
...
void privSignDecrypt (APDU apdu) {
  byte[] buffer = apdu.getBuffer();
  if ((buffer[ISO7816.OFFSET_P1] != 0) ||
      (buffer[ISO7816.OFFSET_P2] != 0))
    ISOException.throwIt(ISO7816.SW_WRONG_P1P2);
  inDataLength = nonNegativeByte(buffer[ISO7816.OFFSET_LC]);
  if (inDataLength != 128)
    ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
  receiveIncomingData(apdu);
  Util.arrayCopy(inData,(short)0,msg,(short)0,inDataLength);
  if (pinState.elem != PinState.verified)
    ISOException.throwIt(SW_SECURITY_NOK);
  else {
    rsaCipher.init(key,Cipher.MODE_DECRYPT);
    rsaCipher.doFinal(msg,(short)0,(short)128,_aux1,(short)0);
    sendOutgoingData(apdu, _aux1);
    return;
  }
}
...
```

Figure 3. SmartSlang spec and corresponding Java Card code. The fragment of a SmartSlang spec above the arrow shows specialized syntax for commands, responses, cryptography, and application protocol data units (APDUs)—all typical smart card concepts.

Figure 3 shows a fragment of a SmartSlang spec that has specialized syntax for commands, responses, cryptography, and application protocol data units (APDUs)—all typical smart card concepts. APDUs are an ISO-standardized, low-level, byte-oriented format for encoding commands to and responses from a smart card.

The fragment declares a command that takes a message as argument and returns the result of applying an

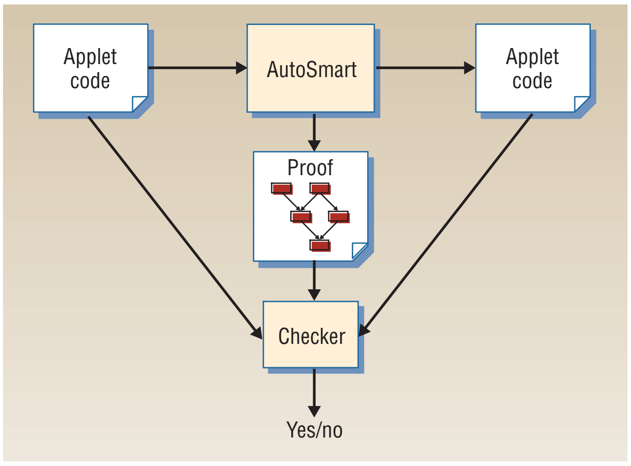


Figure 4. Provable correctness of AutoSmart transformations. A series of correctness-preserving transformation steps achieve AutoSmart's translation of SmartSlang into Java Card.

RSA private key to it—whose declaration is not shown—for signature or decryption. The command performs this operation only if an earlier verification of a PIN via another command—not shown—succeeded; otherwise, it returns an error response. The apdu line specifies how the command is encoded as an APDU.

AutoSmart automatically generates Java Card applets from SmartSlang specs. The term applet here does not refer to Web-browser applets, but means any application installed on a smart card. AutoSmart first performs several consistency checks on the input SmartSlang spec involving automated reasoning such as linear arithmetic, then it translates the spec into Java Card code.

In Figure 3, the code corresponding to the SmartSlang fragment is below the arrow. The code performs the inverse of the encoding specified in the spec's apdu line—that is, it decodes the command from the APDU, which is accessible as a byte array via the Java Card API. Based on the first two bytes, it dispatches control to the method privSignDecrypt. The method retrieves the rest of the APDU and checks its conformance to the encoding specified in the apdu line. The cryptographic operation is realized via calls to the Java Card API.

Given that smart cards have limited memory and that the Java Card does not support garbage collection, applets should not allocate new objects dynamically during normal computation: Applets should instead allocate all objects statically in advance and reuse them during normal computation. The first few lines of the generated code in Figure 3 show the declaration and the preallocation of the objects needed to realize the privSignDecrypt command.

The code that AutoSmart generates is not artificially verbose; on the contrary, AutoSmart's developers carefully engineered it to produce code close to what a human developer would write. This is possible because of SmartSlang's domain specificity: SmartSlang captures smart card concepts that the AutoSmart system translates to the Java Card coding idioms typically used in handwritten code. The result is a three- to four-times expansion of code size from SmartSlang to Java Card.

Figure 4 shows the series of correctness-preserving transformation steps that AutoSmart uses to translate SmartSlang into Java Card. The source and target languages' semantics, SmartSlang and Java Card, have been formalized so that each SmartSlang spec and Java Card program has a representation in the same logical language.

The two representations can be formally compared to prove that the SmartSlang spec and the Java Card program have the same observable behavior, where the observable behavior of a smart card application is the set of all possible traces of command-response exchanges in time. Specware,[4] Kestrel's system for formal software development, formalizes the semantics of both SmartSlang and Java Card. Metaslang, the Specware specification language (www.specware.org), is based on higher-order logic, and has similarities to the functional language ML and the languages of the theorem provers PVS and HOL.

For each transformation refinement that AutoSmart carries out, the system automatically generates an associated correctness proof. This proof establishes that the output is correct with respect to the input, which applies to intermediate stages in the translation, not just to the end points. Kestrel is completing the process of having AutoSmart generate a proof of the code's correctness with respect to the spec, along with an applet's Java Card code, as Figure 4 shows. AutoSmart constructs the proof stepwise as it applies transformations. Thus, the fully automatic transformation process guides the proof's construction.

The proof is expressed in Metaslang, is machine-amenable, and can be checked by a proof checker for the Metaslang logic. Thus, we can be confident about the correctness of an applet that AutoSmart generates without necessarily having confidence in AutoSmart itself.

As Figure 4 shows, the simple checker, which is much smaller and simpler—and therefore easier to trust—than AutoSmart, can check the proof, spec, and code. The checker's core is the Metaslang proof checker, complemented by two simple encoders that map the SmartSlang spec and the Java Card program into their Metaslang representations.

The checker first checks the proof for validity, then uses a simple syntactic comparison to verify that the conclusion of the proof is indeed the formula stating the behavioral equivalence of the given SmartSlang spec and Java Card program. If the proof for a particular applet is valid, then the applet is correct, no matter how many bugs the generator may have. Moreover, the validity of a proof does not depend on the party that produced it.

Mission Data Language

The underlying technology used for SmartSlang is now being applied to MDL, which was developed under the joint leadership of the Kestrel Institute's Lindsay Errington and Kestrel Technology LLC's Allen Goldberg. State analysis is an approach to designing integrated flight, ground, and test systems defined within the Mission Data System project at JPL. MDS seeks to provide an architectural framework that captures best software and system engineering practice supported by a reusable code base.

MDS, a state-based architecture, has the state capture relevant time-varying properties of a system under control, such as position, temperature, pressure, resource levels, device modes, and health status, as well as external environmental conditions. State variables represent components of the state—for example, the state of a battery's charge. The value held in a state variable is a function that gives an estimate of the state variable's value in the past, present, and future, along with an estimate of its certainty of accuracy. Such a time-varying function is called a *state function*.

Separation of estimation and control is a key Mission Data Systems architectural principle.

Separation of estimation and control is a key MDS architectural principle. Traditional flight architectures achieve control by issuing sequences of commands to actuators that, hopefully, result in attaining some desirable goal.

MDS achieves control by directly stating goals—that is, constraints on state variables and, in particular, constraints on their future values. Such goals are achieved through goal elaboration—the breakdown of a goal into a set of lower-level goals that represents one possible way of achieving the high-level goal.

The lowest level goals, those a controller can directly achieve without further decomposition, are known as *executable goals* (XGoals). For example, specifying a location for a planetary rover to move to is a high-level goal, requiring route planning, resource management, and so forth; specifying a desired rotational velocity for a wheel is an XGoal.

MDL source language. With MDL, a formal domain-specific language for expressing the objects and products of state analysis, users can define appropriate types and instances. For example, to model a six-wheeled rover, a user may specify a type such as a wheel-rotation estimator and then define six instances of that type. Initially, in conjunction with high-level state analysis, a user might just specify entities and their dependencies. For example, the wheel-rotation estimator for one wheel could depend on the state variables for the other wheels and measurements from hardware adapters for the wheels.

Users can state such dependencies without specifying the precise algorithm for producing the state function the estimator uses. The code generated then contains stubs that invoke the estimation algorithm where needed.

MDL provides a Matlab-like language for describing the behavior of estimators and controllers. The language also allows the declaration of physical units such as those for mass, velocity, or voltage; design-time checking of units for conformity; and automatic design-time those coercions—for example, conversions between radians and degrees. In the future, we will extend this notion to conversions between reference frames used to describe positions on or near planets and other bodies and in deep space.

The architectural concepts behind MDL have no counterparts in Matlab, Simulink, Stateflow, or UML, although some could be naturally represented in Simulink. MDL anticipates the use of other modeling notations to describe the behavior of estimators and controllers and is designed so that Matlab models can be integrated. Thus, MDL combines the familiarity of Matlab with the advantage of having a formal semantics.

The appropriate modeling notations for estimators and controllers depend on the system being modeled, but various common modeling notations are used for flight software: state machines, which are often used for discrete devices; differential equations; Kalman filters; and so forth. We expect future versions of MDL to support these modeling notations.

MDL operational semantics. MDL has an executable operational semantics. Thus, developers can simulate and experimentally validate an MDL specification. MDL semantics, based on a hybrid message-passing and a synchronous architecture, run estimators, controllers, and hardware adapters periodically. When a component executes, it examines message queues for messages sent to it by other components. It can then generate a message for another component, further driving the simulation.

MDL code generation. Java, the target language for code generation from MDL, is not the traditional language of choice for flight software. On the positive side, Java offers productivity advantages and a growing set of trained programmers, libraries, middleware, and tools that make it an attractive development platform. However, until recently, its weak semantics for real-time computation, lack of support for low-level device control, and unpredictable and potentially large time delays introduced by garbage collection, all made Java unsuitable for hard real-time applications.

The Real-Time Specification for Java has addressed these shortcomings. The RTSJ JVM significantly extends the common Java virtual machine, and it includes a preemptive, priority-based scheduler with mechanisms to prevent priority inversion; asynchronous event handling; high-resolution timers; and an explicit memory-management scheme that lets Java execute safely without a garbage collector. Currently available commercial RTSJ implementations, with extensive RTSJ APIs, support many styles of real-time computation.

For hard real-time applications such as MDS control loops, garbage collections introduce unacceptable delays. Thus, they must use scoped memory—RTSJ's explicit memory management feature. Initial experience with scoped memory has shown that programmers find it subtle and difficult to use. However, the MDL code generator hides this complexity, defining all needed memory scopes and performing the required copying of data from memory scopes with different lifetimes.

The MDL code generator targets a straightforward runtime architecture. Each component instance runs as an RTSJ no-heap real-time thread. Rate-monotonic analysis based on specified periods determines thread priority, and each thread has a scratch memory scope. Long-lived state information—for example, state variables—is stored in the RTSJ JVM's immortal memory.

NASA is studying new general-purpose languages for the rapid development and deployment of onboard systems.

The generated code conforms to efforts to define a safety-critical subset of RTSJ that restricts language use to the common idioms used within the safety-critical community, mainly commercial avionics. This will facilitate bootstrapping a community of Java users and enable FAA certification for reasonable cost. We estimate the code expansion from MDL to Java to be a factor of from 3 to 4.

Code correctness. The approach used to show the correctness of the transformation from SmartSlang to Java Card can also establish the correctness of the code generation from MDL to RTSJ.

Specifically, this involves describing the semantics of MDL and RTSJ in Metaslang, leveraging the substantial commonalities with the Java Card semantics already developed. The common aspects include object and bytecode representation, JVM instruction semantics, and internal JVM data structures.

We have defined a notion of behavioral equivalence between the MDL source spec and the Java target code: They are behaviorally equivalent if they generate the same set of commands to hardware actuators within simulated time. We can at best give conditional guarantees for real-time performance, since this depends on the implementation of RTSJ.

While many modeling languages serve to produce throwaway prototypes, the code produced for SmartSlang and MDL is intended for production and should be as efficient as handcrafted code. Both DSLs remove the effort of dealing with fairly low-level but nontrivial problems, such as data marshalling, scheduling, and memory management—all well-known error-prone tasks when done by programmers. The code generators utilize schedulability analysis, symbolic arithmetic reasoning, and other analyses that go beyond conventional compiler technology.

Indeed, using a well-tested generator to generate code automatically provides greater assurance than manual coding, and having the generator produce an independently checkable proof provides even greater assurance, overcoming any concerns about the generator's soundness.

GENERAL-PURPOSE LANGUAGES

To address future needs, NASA is also studying new general-purpose languages such as SequenceL for the rapid development and deployment of onboard systems. SequenceL is a small, Turing-complete, and executable language comprised of eight grammar rules.[5-8]

In SequenceL, a simple semantic called the Normalize-Transpose-Distribute (NTD) discovers and automatically generates most iterative and nested iterative algorithmic structures. Because SequenceL synthesizes the structures in a consistent and provably correct manner, it effectively avoids many programming errors and saves considerable time and expense. Although SequenceL is a general-purpose language, it has proven to be similar to the languages NASA engineers use when developing the requirements for GN&C.

As a proof of concept, a SequenceL version of a GN&C decision-support system—the Shuttle Abort Flight Management (SAFM) system—will be tested on the Shuttle Engineering Simulation Environment at Johnson Space Center later this year. It took six months of full-time effort for one GN&C engineer to complete the original SAFM prototype, which was designed by a GN&C team at JSC. Once the system was validated, the team turned over the requirements document to General Dynamics to develop the flight-certified code, which took roughly two years to complete and cost approximately $8 million.

Future missions will require quick and dependable means to modify systems—means that current software development approaches do not fully afford. Mission capabilities will require rapid modifications between and, more importantly, during missions. To address these concerns, we are currently demonstrating that SequenceL can be used for the identification and validation of GN&C requirements.

Working mainly with the requirements documents, and with minimal contact with the JSC GN&C engineers, SequenceL specifiers have developed an understanding of the requirements and elaborated and validated them in SequenceL with six weeks of one person's full-time effort. Thus, in our initial experiment, we completed the requirements review and produced a working prototype

3.7.4.13.1 Functional Requirements
3.7.4.13.1.1 The signature of the Earth Fixed to Runway Transformation utility shall be as follows:

M_EFTo_Rw = EF_TO_RUNWAY(Lat, Lon, RW_Azimuth)

3.7.4.13.1.2 The Earth Fixed to Runway Transformation utility shall perform the following algorithm:

$$\mathbf{M} = \begin{pmatrix} \text{Cos(RW\_Azimuth)}, & \text{Sin(RW\_Azimuth)}, & 0 \\ \text{-Sin(RW\_Azimuth)}, & \text{Cos(RW\_Azimuth)}, & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{MEFTopdet} = \begin{pmatrix} \text{-Sin(Lat) * Cos(Lon)}, & \text{-Sin(Lat) * Sin(Lon)}, & \text{Cos(Lat)} \\ \text{-Sin(Lon)}, & \text{Cos(Lon)}, & 0 \\ \text{-Cos(Lat) * Cos(Lon)}, & \text{-Cos(Lat) * Sin(Lon)}, & \text{-Sin(Lat)} \end{pmatrix}$$

M_EF_To_Rw = (**M**) • (**MEFTopdet**)
*Rationale: **M** is the Topodetic to RW matrix.*

(a)

```
M_EF_To_Rw(Lat,Lon,Rw_Azimuth) ::= mmrow,
 (            (Cos(RW_Azimuth),          Sin(RW_Azimuth),          0),

              (-Sin(RW_Azimuth),         Cos(RW_Azimuth),          0),

              (0                         0                         1)

              ),
(             (-Sin(Lat) * Cos(Lon),     -Sin(Lat) * Sin(Lon),     Cos(Lat)   ),

              (-Sin(Lon),                Cos(Lon),                 0),

              (-Cos(Lat) * Cos(Lon),     -Cos(Lat) * Sin(Lon),     -Sin(Lat))
              )
```

When one adds a two line SequenceL definition for Matrix Multiply:

```
              mmrow: [s] * [[s]] -> [[s]]      mmrow(a,b) ::= dotProd(a,transpose(b))
              dotProd: [s] * [s] -> s           dotProd(x,y) ::= sum(x * y)
```

(b)

Figure 5. (a) Example requirement from the Shuttle Abort Flight Management system requirements document. (b) Executable SequenceL version of the requirement.

in SequenceL in one-fourth the time it took the GN&C engineers for the original SAFM development.

As Figure 5 shows, the major change in the SequenceL version of the requirement is adding the definition of matrix multiply, mmrow, and dotProd, and adding nested ()'s to denote the rows of the matrix. Since matrix computations (including matrix multiply) are the gist of the requirement, the SequenceL NTD semantic performs all the work in terms of generating the procedural aspects of the problem solution. Using SequenceL, the specifier does not have to engage in the error-prone efforts involved in coding the nested iterative control structures, nor is it necessary to manage the matrix subscripts—also an error-prone activity.

With languages like SequenceL, engineers can focus their attention on requirements without manufacturing the procedural algorithms that satisfy those requirements. Languages that correctly synthesize codes will be essential for realizing rapid development and dependable deployment capabilities.

RISK MANAGEMENT FOCUS

A focus on capability engineering must also improve the engineering discipline of software project management to meet or exceed that of hardware project management. Software is almost entirely a design artifact, yet today few tools are available to support risk analysis of that design, simulate design tradeoffs, implement safety design factors, and analyze design scalability. Cost and schedule estimation tools are often inaccurate and, owing to historical calibrations, fail to keep pace with the rapidly increasing allocation of system functionality to software.

Defect estimation models similarly are grounded in historical practice, not accounting for technologies inserted at each step in the life cycle to either decrease defect introduction or increase defect removal. Designing quality in at the development cycle's beginning is far more effective in terms of cost and dependability than testing in quality at the development cycle's end.

Model-based approaches that synthesize correct codes reduce the risks associated with hand-coding systems.

Tools employing domain-specific languages to facilitate communication, enforce proper requirements elicitation, examine coupling in design features, produce provably correct models, and link verification tests to specifications will increase quality at the development cycle's beginning. Formal models of specifications, safety requirements, domain knowledge, and implementation knowledge enable rigorous analysis to enhance upfront quality.

Practices that put software on an equal footing with hardware, such as JPL's state analysis practice, ensure that software and hardware designs evolve together and trace decisions leading to functional allocation without the context loss of traditional hierarchical decompositions.[9]

Developers need quantitative measures of design quality and runtime performance to demonstrate the system candidate's trustworthiness and make trades against measures that stakeholders deem important for safety-critical systems. Such measures also support a statistical statement of system risk exposure given the inherent uncertainty in the exploration environment.

To summarize the differences between hardware and software development, we offer the following points:

- *Software does not wear out.* Therefore, safety factors for software are not measured as easily as they are for hardware—for example, if a mechanical switch needs to function 1,000 times, we can develop a switch with a factor of safety to double or triple its lifetime.
- *Hardware risk analysis can be done early.* It can be performed at the requirements or design level because manufacturing of the hardware can be accomplished with degrees of certainty that exceed those involved in software manufacturing.
- *Software's incremental production costs are virtually nil.* This is the one factor already in software's favor—particularly with the current promise of automatic prototyping. Although we can learn how to improve hardware design through experimentation and experience in the production phase, we learn nothing about how to improve a software design by creating one more copy.

Software development will not achieve the level of hardware development without the introduction and use of model-based languages. Further, model-based development will not be widely accepted without risk analysis methodologies on a par with those employed in hardware development.

Current software practices will not admit risk analysis strictly at the design phase because too much risk and uncertainty remain in the implementation and verification of the software—the manufacturing phase.

Model-based approaches that synthesize correct codes vastly reduce, if not eliminate, the risks associated with hand-coding the systems to be deployed. Thus, risk analysis can be focused at the requirements level because the requirements themselves reside in the software.

Once model-based approaches demonstrate an ability to perform risk analysis at the same level as hardware risk analysis, they will likely be more widely accepted. When hardware and software are on a somewhat equal footing, the ultimate fusing of software and systems engineering can be accomplished, and we can focus more on capabilities and less on whether a human, program, or piece of hardware provides a capability.

Finally, we will most likely see safety factors realized in model-based approaches. For example, designers of a system component that performs some action can either assume that the action has been performed or they can measure and specify the costs involved in instrumenting the environment to verify the effects of the action. ■

Acknowledgments

Parts of this work were supported by NASA-NNG05GP48G and NASA-NNA05CR24C. We thank Susan Creasy, Kevin Window, Marcia Kerr, Howard Hu, Alessandro Coglio, Lambert Meertens, Allen Goldberg, and Matthias Anlauff for contributions to various sections of the article.

References

1. Defense Science Board, "Report of the Defense Science Board Task Force on Defense Software," Nov. 2000, Office of the Under Secretary of Defense for Acquisition and Technology, Washington, D.C.
2. A. Coglio and C. Green, "A Constructive Approach to Correctness, Exemplified by a Generator for Certified Java Card Applets," *Proc. IFIP Working Conf. Verified Software: Tools, Techniques, and Experiments*, 2006, to appear; vstte.ethz.ch/Files/coglio-green.pdf.
3. A. Coglio, "Toward Automatic Generation of Provably Correct Java Card Applets," *Proc. 5th ECOOP Workshop Formal Techniques for Java-like Programs*, 2003, tech. report 408, ETH Zürich, n.pag.; www.cs.ru.nl/ftfjp/2003/15.pdf.
4. "Specware User Manual," Kestrel Institute and Kestrel Technology LLC, 2004; www.Specware.org/doc.html.
5. D.E. Cooke and V. Kreinovich, "Automatic Concurrency in SequenceL," *Science of Computer Programming*, vol. 42, no. 1, 2002, pp. 115-128.
6. D.E. Cooke and J.N. Rushton, "Normalize, Transpose, and Distribute: A Basis for the Decomposition and Parallel Evaluation of Nonscalars," to be published in *ACM Trans. Programming Languages and Systems*, 2006.
7. D.E. Cooke and J.N. Rushton, "SequenceL—An Overview of a Simple Language," *Proc. Int'l Conf. Programming Languages and Compilers* (PLC 05), 2005, pp. 64-70.
8. D. Cooke et al., "Application of Model-Based Technology Systems for Autonomous Systems," *Proc. Infotech@Aerospace*, AIAA-2005-7063, Infotech@Aerospace, 2005.
9. D. Dvorak, R. Rasmussen, and T. Starbird, "State Knowledge Representation in the Mission Data System," *Proc. 2002 IEEE Aerospace Conf.*, IEEE Press, 2002.

***Daniel E. Cooke** is a professor in the Department of Computer Science, Texas Tech University. His research interests include declarative languages, program synthesis, and program verification. Cooke received a PhD in computer science from the University of Texas at Arlington. Contact him at dcooke@coe.ttu.edu.*

***Matt Barry** is the program director for Surface Support Systems at the Jet Propulsion Laboratory. His research interests include technology selection, artificial intelligence, and systems engineering. Barry received a PhD in computer science from Rice University. Contact him at matthew.r.barry@nasa.gov.*

***Michael Lowry** is the head of the Robust Software Engineering Group at the NASA Ames Research Center. His research interests include program synthesis, automated verification and validation, and automated software engineering. Lowry received a PhD in computer science from Stanford University. Contact him at michael.r.lowry@nasa.gov.*

***Cordell Green** is the director of the Kestrel Institute. His research interests include knowledge-based tools for software engineering and automated software design and synthesis. Green received a PhD in electrical engineering from Stanford University. He is a Fellow of the ACM and a member of the IEEE. Contact him at green@kestrel.edu.*