

# NESTA: NASA Engineering Shuttle Telemetry Agent

*Glenn S. Semmel, Steven R. Davis, Kurt W. Leucht, Dan A. Rowe, Kevin E. Smith, Ryan O'Farrell, and Ladislau Bölöni*

■ The Electrical Systems Division at the NASA Kennedy Space Center has developed and deployed an agent-based tool to monitor the space shuttle's ground processing telemetry stream. The application, the NASA Engineering Shuttle Telemetry Agent (NESTA), increases situational awareness for system and hardware engineers during ground processing of the shuttle's subsystems. The agent provides autonomous monitoring of the telemetry stream and automatically alerts system engineers when predefined criteria have been met. Efficiency and safety are improved through increased automation.

Sandia National Labs' Java Expert System Shell is employed as the rule engine. The shell's predicate logic lends itself well to capturing the heuristics and specifying the engineering rules of this spaceport domain. The declarative paradigm of the rule-based agent yields a highly modular and scalable design spanning multiple subsystems of the shuttle. Several hundred monitoring rules have been written thus far with corresponding notifications sent to shuttle engineers. This article discusses the rule-based telemetry agent used for space shuttle ground processing and explains the problem domain, development of the agent software, benefits of AI technology, and deployment and sustaining engineering of the product.

NASA Kennedy Space Center (KSC) is responsible for prelaunch ground checkout of the space shuttle. The Launch Processing System (LPS) at KSC provides facilities for NASA shuttle system engineers, con-

tractors, and test conductors to command, control, and monitor space vehicle systems from the start of shuttle interface testing through various phases including terminal countdown, launch, abort, safing, and scrub turnaround.

LPS continually monitors the shuttle and its ground equipment including environmental controls and hardware that loads propellants. Consoles with vehicle responsibilities communicate information directly to and from the shuttle computer systems. Consoles with ground support equipment responsibility communicate information to and from the hardware interface modules that are connected to the numerous ground support systems (see figure 1). Each module is capable of interfacing to approximately 240 sensors or controls. Overall, some 50,000 temperatures, pressures, flow rates, liquid levels, turbine speeds, voltages, currents, valve positions, switch positions, and many other parameters must be controlled and monitored.

Using LPS, NASA shuttle engineers and contractors at KSC are responsible for certifying that ground checkout of the space shuttle has been performed according to program specifications. For more than 25 years, engineers have used LPS to verify space shuttle flight readiness and to control launch countdown. LPS has performed superbly well. Recently, much of the LPS hardware was upgraded assuring its continuance for many more years. However, the

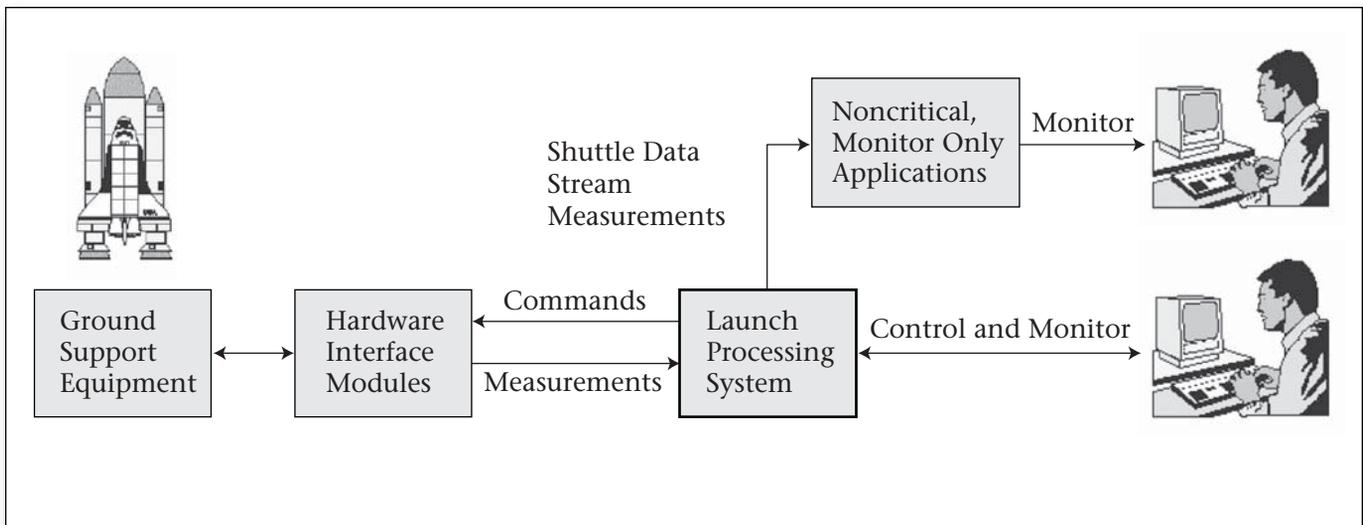


Figure 1: Ground Control and Monitoring at NASA KSC.

system architecture was not changed, and the software remains basically the same. As a result, the level of situational awareness has not increased proportionally to what would otherwise be possible with more modern software technologies.

After the shuttle *Columbia* disaster on February 1, 2003, the *Columbia* Accident Investigation Board (Gehman et al. 2003) proposed recommendations to improve safety from both an organizational and technical perspective. The board indicated the need to adopt “and maintain a shuttle flight schedule that is consistent with available resources.” Also, both management and engineering support staff must maintain an awareness of anomalies and those must not be lost “as engineering risk analyses [move] through the process.” Given two tragic losses of a crew and shuttle, today NASA engineers have an even greater pressure to be more vigilant in identifying problems. At KSC, ground processing of the shuttle is performed by thousands of employees, both contractors and civil servants. Anomalies must be detected and reported to prevent problems with shuttle subsystems, countdown, and launch. The aging LPS hardware has limited resources and precludes the level of automation and notification warranted by this domain.

Contractors at KSC are responsible for the day-to-day operations, checkout, and maintenance of the shuttle. They are the primary users of LPS. NASA shuttle engineers are civil service employees who oversee the contractors. Given the limitations and resource scarcity of LPS, NASA shuttle engineers needed a tool to provide more insight and situational awareness and oversee the work performed by contrac-

tors. An increased insight could help detect anomalies that might otherwise go unnoticed, whether by process error, software or hardware failures in the monitoring equipment, or many other possible causes. A tool was needed to complement LPS that could autonomously and continuously monitor shuttle telemetry data and automatically alert NASA shuttle engineers when predefined criteria have been met. In the latter half of 2003, a software tool was proposed to provide better insight into shuttle ground processing and increase the level of situational awareness. This tool is known as the NASA Engineering Shuttle Telemetry Agent (NESTA).

## Objectives

Data processed by LPS is distributed on a local area network. As shown in figure 1, the distributed data is known as the shuttle data stream (SDS) and contains real-time vehicle ground processing data. It is used by monitor-only applications. The primary objective of NESTA is to provide full-time autonomous monitoring of the SDS and to automatically alert NASA engineers in near real time when predefined criteria have been met. Types of monitoring criteria include expected operational events or milestones (such as vehicle power up, start of launch countdown test, and so on) as well as unexpected events or failures (for example, a large difference between redundant sensor values). NESTA allows shuttle engineers to work on other tasks while minimizing the risk of losing awareness of real-time shuttle processing data and events.

NESTA acts as a software agent for the NASA

engineer. For this discussion, an agent is defined as rule-based, autonomous software that reacts to its environment and communicates results to a human, a NASA engineer in this usage. Agents have been extensively researched (Wooldridge 2000; Russell and Norvig 2003). Agent standards<sup>1</sup> and frameworks<sup>2</sup> (Bölöni and Marinescu 2000) have also been developed. The five primary objectives for NESTA include, first, allowing a NASA engineer to specify rules to be applied to measurements published in the SDS; second, generating near real-time notifications and alerts in the form of e-mails or wireless pages (notifications may include a text message and measurement values and may be sent to multiple users when the rule's premises are satisfied); third, monitoring up to four separate SDS sources, including four control rooms used for checkout and launch of the shuttle and its components; fourth, processing multiple types and subtypes of measurements including discretely (Boolean measurements), analogs (floating-point measurements), and digital patterns (integer measurements); and fifth, allowing users to create and modify multiple monitoring requests without restarting NESTA.

## Why an AI Solution

NESTA leverages various AI technologies within a rule-based paradigm including forward chaining, fast pattern matching by means of the Rete algorithm, declarative programming, predicate logic, and more. AI was a natural fit for monitoring the SDS because pattern recognition and analysis are the primary needs. Although pattern identification could be achieved by employing regular expression libraries within various procedural and object-oriented languages, those paradigms are not specifically intended for this type of application and have less-efficient matching algorithms. The pattern-matching algorithms of rule-based expert system shells are highly specialized and tuned. Also, AI—particularly rule-based languages—lends itself better to this domain because pattern recognition wrapped within a premise-action construct closely mirrors the level of abstraction at which the domain experts work.

The type of data signatures sought by shuttle engineers requires the derivation of rules that are of the same granularity as those typically used in rule-based languages. Fortunately, shuttle engineers were already accustomed to representing knowledge at a fine-grained level. The engineers are adept at either constructing

the rules themselves or expressing the knowledge in pseudocode that lends itself well for translation directly into declarative rules. Many of the rules are either stand-alone or work in conjunction with several other rules, thus suggesting a highly modular system with a rule being a suitably sized working block.

## Other Attempted Solutions

NESTA is a peripheral advisory tool to the real-time control system within LPS. There were three previous projects that attempted to upgrade LPS in the last 15 years. Although those efforts had significantly greater objectives that spanned well beyond just advisory applications, they were advertised to include many of the capabilities that NESTA provides and much more. Approximately half a billion dollars was spent on those efforts and upwards of 600 people worked on the most recent of those upgrade attempts. There were various technical and political hurdles that initially impeded and then ultimately doomed those full-scale replacements of LPS.

NESTA's infusion of state-of-the-art AI technologies and engineering within the legacy launch system, LPS, is particularly notable given the number and size of the preceding attempts to modernize the ground control system at KSC. Those fallen projects, despite having much grander objectives, had little to no spinoffs within the LPS community. In contrast, NESTA is becoming accepted and internalized by members of the launch team and appears to be on its way as a widely used tool. From a business vantage point, NESTA's greatest asset is its development and marketing as a value-added product. That is helping pave its path to acceptance.

## System Components and How They Interact

Figure 2 shows the context diagram for NESTA. The agent process is represented in the middle circle. It communicates with various sources and data stores. A measurement database is used to decode the SDS into usable measurements. The SDS source broadcasts measurements as data packets over local area networks. NESTA monitors this stream for data patterns specified by the shuttle engineers. If a pattern is matched, a notification is sent in the form of an e-mail or wireless page. The Rules data store represents the Jess scripts and knowledge base that defines the rules for the monitoring criteria. All messages and relevant agent activities are also locally logged.

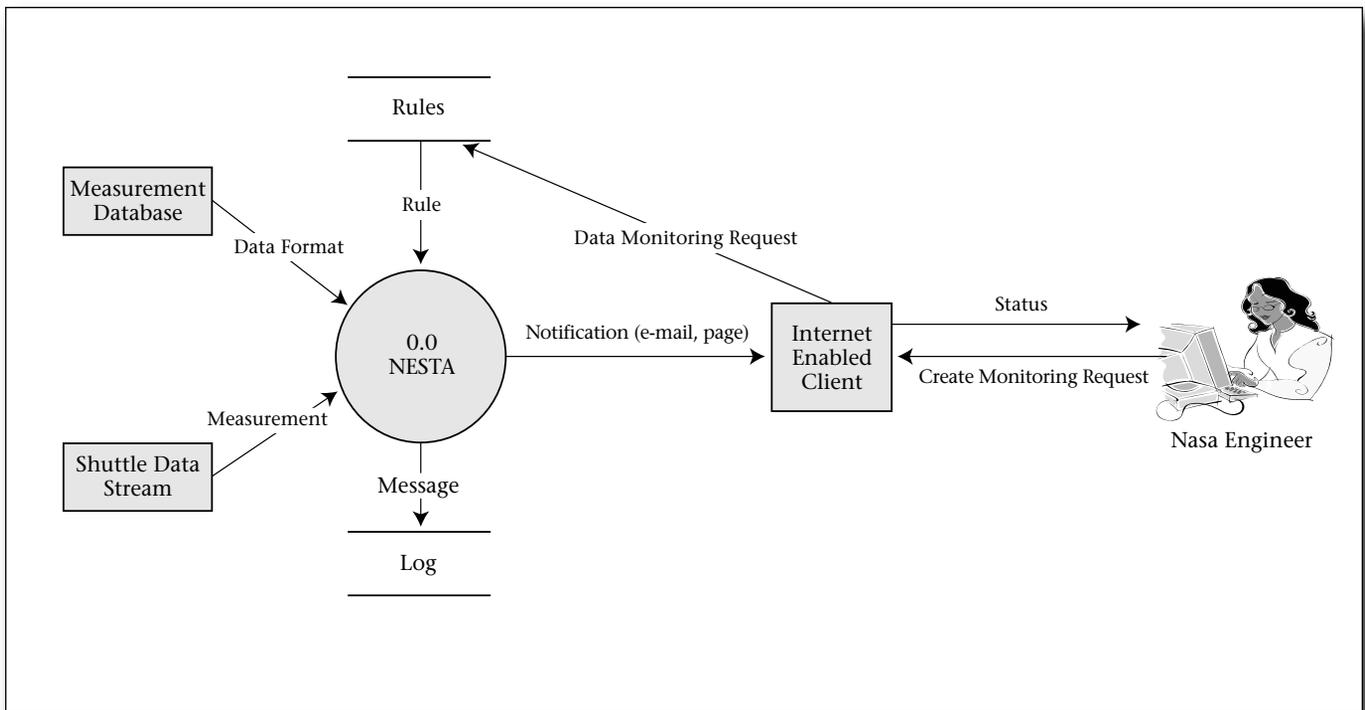


Figure 2: NESTA Context Diagram.

## Languages and AI Tools Used

The Java Expert System Shell (Jess) (Friedman-Hill 2003) was selected as the rule engine. Jess was developed and supported by another government agency, Sandia National Labs. As such, our development team and customer have full usage of the tool through government licensing without any fees. This includes access to all the Jess source code.

Jess's forward-chaining reasoning system was modeled after production systems such as OPS5 (Brownston et al. 1986) and CLIPS (Wygant 1989). It contains a highly efficient and sophisticated pattern matching based on the Rete algorithm (Forgy 1982), which enables its inference engine to process many rules and data rapidly. The engine repeatedly processes through a match-select-act cycle. As a production system, its consequents can be actions. A conflict-resolution strategy determines the precedence of rule firings.

Several hundred monitoring rules have been written thus far for monitoring shuttle ground telemetry. Jess's predicate logic lends itself to capturing and specifying the heuristics and engineering rules of this spaceport domain. The declarative paradigm of this rule-based agent application also makes it highly modular and scalable to span multiple subsystems of the shuttle. Jess also includes a fourth-generation scripting language and interactive command

line that are very conducive for prototyping and testing.

Jess is written entirely in Java and has access to the full Java application programming interface from the scripting language. It provides standard control-flow constructs and supports variables, strings, objects, and function calls. Jess automatically converts between its own types and Java types insulating the developer from manually performing the conversions. Its use as a Java library made Jess's selection more appealing since Java supports multiple platforms with its "write once, run anywhere" paradigm. Beyond that, the need for NESTA to support web-enabled clients also made Java a natural fit given its origins and strong support for developing Internet based applications.

## Design

Java classes were developed to parse and decode the data stream and represent measurements as facts in Jess's working memory. To interface Jess's rule engine with the SDS, each data measurement is modeled and implemented as a Java bean.<sup>3</sup> Java beans provide a component architecture to enable easier integration of applications. A property change notification mechanism is supported that allows one object to become a registered listener of another object. The listener object will then automatically receive changes from the source object. This

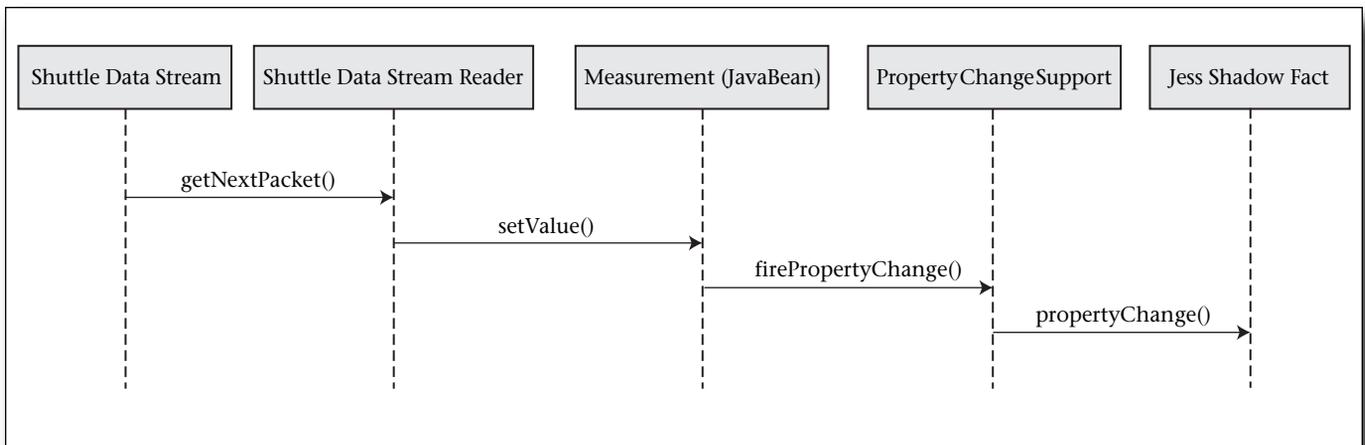


Figure 3. Sequence Diagram Illustrating Update to Jess's Working Memory from Shuttle Data Stream.

is also known as a publish-subscribe or observer pattern (Gamma et al. 1995). Within Jess, each Java bean corresponds to what is known as a shadow fact. A Jess shadow fact is a mirror image of a Java bean, such as a pressure measurement, within Jess's working memory. All shadow facts are registered listeners of their Java bean counterparts. Thus, whenever a measurement changes in the data stream, a property change event is automatically generated for the given measurement, and its sibling shadow fact is updated in Jess's working memory. Figure 3 illustrates this path.

After a shadow fact is updated, the Jess pattern matcher will determine if the premises of any rules match the new or modified facts. Rules are compared to working memory to identify premises that are matched by the data in working memory. For NESTA, this data represents measurements from the SDS and rules represent data monitoring criteria submitted by NASA shuttle and system engineers. Rules with matching premises are activated and placed onto an agenda. Next, the agenda is ordered according to Jess's default conflict-resolution strategy. The highest-priority rule is then fired and executed. This match-select-act cycle repeats until no more rules are available to fire. An action handler class was developed and is used to build and send the notification message to the shuttle engineer whenever a rule fires.

## Knowledge Capture and Representation

Figure 4 shows the knowledge-acquisition workflow for creating or modifying a rule to monitor specific measurements on the shuttle

data stream. The shuttle engineer must specify who is responsible for the rule, the contents of the e-mail notifications, the rule's firing conditions (that is, antecedent, left side), and rearming conditions. That is, some rules may need to have a "one shot" behavior and fire only once when activated the first time. Other rules may need to be rearmed after a given time period or when certain types of conditions are met.

The current version of NESTA does not have a graphical user interface capturing this workflow, but all of the steps are effectively provided within script files. Those files are editable with a plain text editor by the end users. Hundreds of rules have been produced by the customer.

As the rule database grew, patterns of rules began to emerge. Patterns in software design and modeling have been extensively investigated and reported (Gamma et al. 1995). Analogous to those design patterns, the development team and customer began recognizing knowledge patterns for this domain and developed rules following these structures. Some patterns include the following:

*One shot:* The rule fires once regardless of how many times facts cause the premise to re-activate.

*Recurring:* The rule fires each time the premise reactivates.

*Timed:* The rule fires every  $x$  minutes as the premise remains true.

*Queued:* Multiple rules will fire, but notifications are sent to a queue that gets flushed based on a user-configurable amount of time or maximum number of firings. One composite notification is sent when the queue is flushed. That composite notification contains what would have otherwise been multiple e-mails or wireless pages.

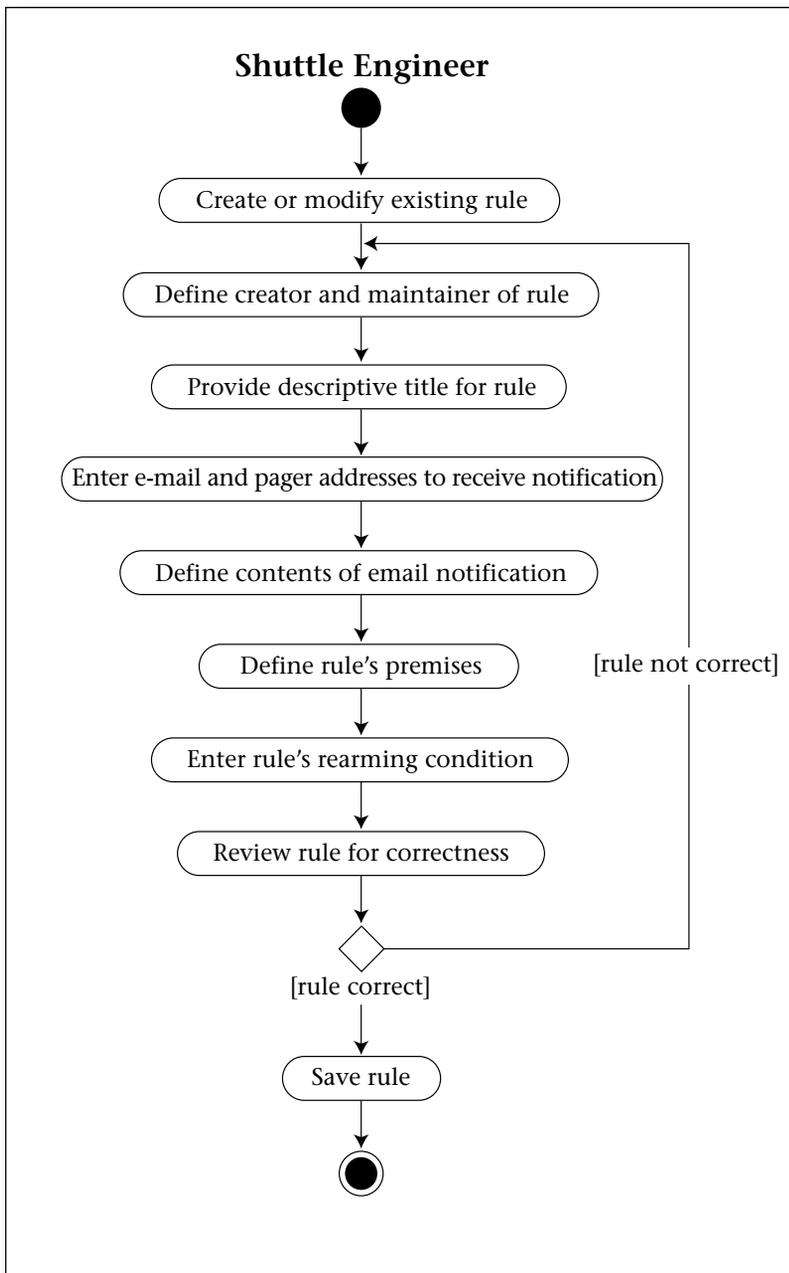


Figure 4. NESTA Knowledge-Acquisition Workflow.

Some sample rules in English prose include the following:

*Notify shuttle engineer when measurement V79S4126E1 or V79S4132E1 or V79S4138E1 or V79S4143E1 equal ON.* This rule indicates that Flight Control Power (ASA 1-4) has been activated.

*Notify shuttle engineer when measurement V90Q8001C1 equals 801.* This rule indicates that a shuttle is in orbit and is preparing to initiate the on-orbit flight-control checkout activity.

*Notify shuttle engineer every 60 minutes with current values of Flight Control launch countdown measurement list when measurement NMA-JORTEST equals 7.* This rule indicates that launch countdown test is occurring. While in launch countdown test, send a current value e-mail containing a list of flight-control measurements every hour.

*Notify shuttle engineer when FD N79IV019D Bit masked 0x0001 equals 1.* This rule indicates that an LPS command and control program has stopped due to a failure and is waiting on the operator for action.

Figure 5 depicts an actual NESTA rule written in the Jess scripting language.

For the rule in figure 5, if all three analog bus voltage measurements, V76V0100A1, V76V0200A1, and V76V0300A1, concurrently exceed 26 volts, the shuttle orbiter is considered to be powered on. Another indicator, SOIADATAV, is used to assure the validity of the incoming data. Data validity is discussed later in this article. Finally, another measurement, NORBTAILNO, is located on the rule's left side. In our terminology, we call this an informational measurement as its specific value has no bearing on whether the rule fires, but it is necessary to include it on the rule's left side so that it becomes part of Jess's activation object and then its value is included in the notification. The action handler parses the fields in the activation object and builds an e-mail with all of the measurements' values that were listed on the left side of the rule. The notifyActionHandler call has two arguments that allow for the notification to be queued. This particular example does not use queuing and simply passes nil arguments in the call. Queuing is also discussed later in the article.

Figure 6 shows an e-mail that was generated for the rule in figure 5. As illustrated, the exact values of all three bus voltages are listed along with the informational measurement showing which of the three Orbiters was powered up. In this case, 103 refers to *Discovery*. The informational measurement proves useful not only in allowing the orbiter reference to be included in the e-mail, but also in not binding the rule to a particular orbiter. That is, NASA shuttle engineers are interested in any orbiter that may become powered up. The rule's pattern matching provides that level of genericity in a very straightforward representation. Of course, the engineer may be interested in being notified only about a specific orbiter. This would require a simple modification to the rule. One additional slot would be referenced in the Digital-PatternFd template narrowing the focus to a particular orbiter. Thus, minor modifications to

```

(defrule vehicle-pwr-on-rule
  "Orbiter electrical power is up."

  (recipient-list (recipient-list-name vehicle-pwr-on-rule))

  ?notPowered <-(vehicle-not-powered)

  (DigitalPatternFd (fdName "NORBTAILNO") )

  (AnalogFd (fdName "V76V0100A1") (valid TRUE) (value ?val1))
  (AnalogFd (fdName "V76V0200A1") (valid TRUE) (value ?val2))
  (AnalogFd (fdName "V76V0300A1") (valid TRUE) (value ?val3))
  (test

    (and
      (> ?val1 26.0)
      (> ?val2 26.0)
      (> ?val3 26.0)

    )
  )

  =>

  (retract ?notPowered)
  (assert (vehicle-powered))
  (notifyActionHandler nil nil)

)

```

Figure 5. An Actual NESTA Rule Written in the Jess Scripting Language.

the rule demonstrate the rich behavior available to the shuttle engineer and show the semantic power of pattern matching.

## The NESTA Hardware and Software Environment

The NESTA application resides on a Dell 1.7 GHz Pentium server. The server includes the necessary user and support files such as the facts scripts, rules scripts, measurement database, logs, and more. Currently, the server executes on a Microsoft Windows 2000 operating system. However, since Java was used exclusively along with its virtual machine, the ability to execute software on other types of servers is readily available. Again, not being bound to a particular hardware platform or operating system was a primary driver in the selection of Java and Jess. Customers receive notification on standard e-mail clients including Windows workstations, wireless pagers, personal digital assistants, cell phones, and more.

## Performance Characteristics of the Shuttle Data Stream

At application startup, NESTA connects to a datastream selected by the user. The datastream includes all measurements at their respective change rates. No data changes will be missing from this stream.

The datastream averages 5 to 10 packets per second and peaks around 50 packets per second at launch. Each SDS data packet can hold up to 360 measurement changes before rolling over to another packet. This calculates to an average of 1,800 changes per second nominally, and 18,000 changes per second peak at launch. During peak data loads, the SDS is throttled at the source and does not maintain true real-time updates. It may lag up to 1 minute or so, but all measurement changes are buffered and none is ever dropped from the data stream. Throttling of the data typically begins at  $T + 1$  second, that is, just after launch. Even though it is the hypothetical peak limit, 18,000 changes per second is the performance load that NESTA is ex-

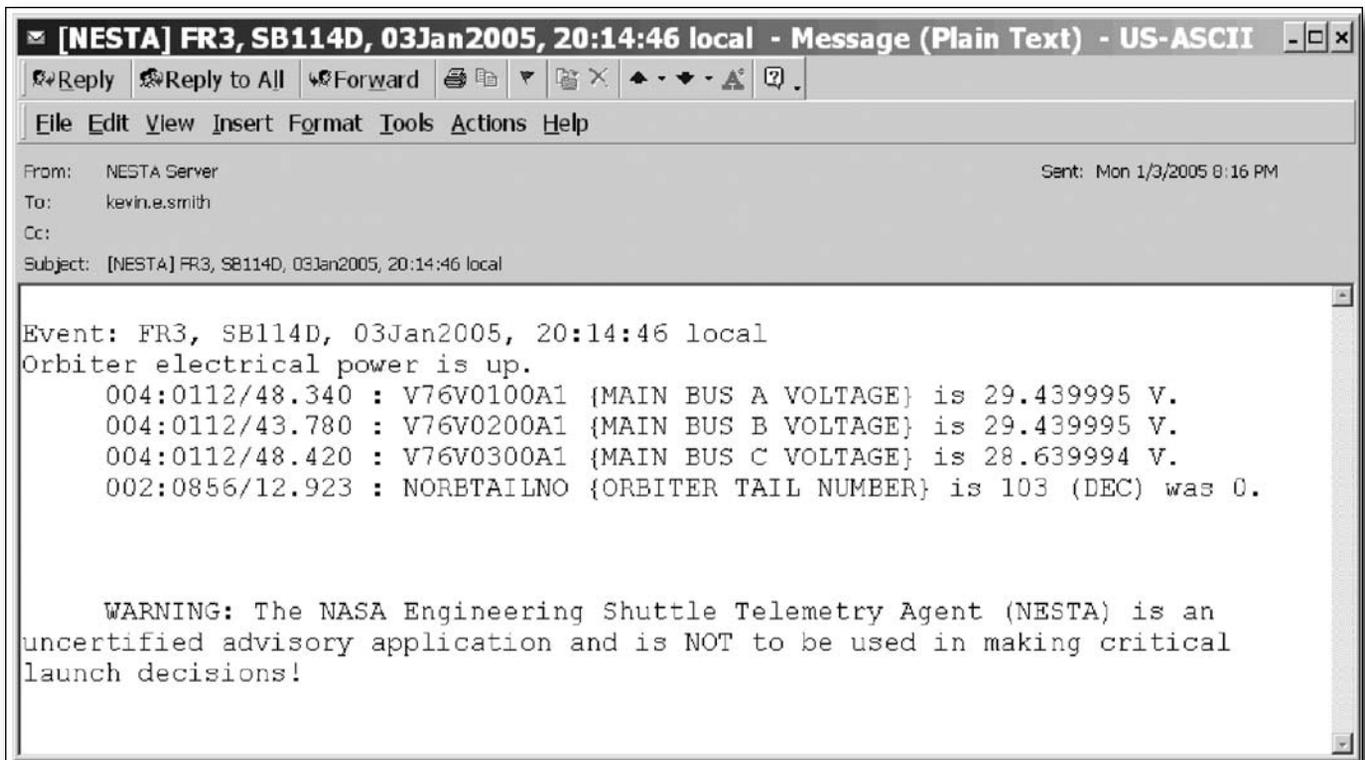


Figure 6. E-mail Generated by NESTA.

pected to meet to avoid missing a measurement change (referring strictly to updating 18,000 facts per second and not indicating how many rules might fire). In fact, only a small percentage of those facts is expected to result in a small percentage of the total rules firing at any given time, even during the peak launch data rates.

The measurement data in the stream is refreshed every three minutes regardless of whether or not it has changed. Since the stream is based on user datagram protocol (UDP), an unreliable datagram packet service is the result. When a packet is dropped on the network, all measurements are marked invalid and the measurements change back to valid one by one as refresh data is received until the completion of a three-minute refresh cycle.

### Performance Testing

Performance testing occurred on an Intel Pentium 4, 1.7 GHz desktop workstation with 768 MB of RAM running Microsoft Windows XP Professional. The SDS reader class in NESTA parses the data stream and updates facts in Jess's working memory. To test the reader class, 12 high-speed analog measurements were selected and instantiated as shadow facts. In the range of 18,000 (nominal) to 36,000 (peak at launch) data changes occurred every second in the test-enhanced data stream and were

processed by the SDS reader class, including various types of measurements such as discretized and analogs. Twelve thousand analog data changes per second were being processed into current values and updated in Jess's working memory by a property change event handler.

Rules were written for 6 of the high speed analog measurements. The other 6 measurements were still relevant to stress the SDS reader class and updating of facts. Five of the 6 rules fired once every minute. The sixth rule fired once for every single measurement change (1,000 per second) for two full seconds sustained out of every minute. Thus, a total of 2,005 rules fired every minute, with 2,000 of them firing within a two-second period. Analog measurements have considerably more processing overhead than the discrete measurements, so it was not possible to sustain thousands of rules containing analogs to fire every second without causing CPU starvation. However, the "fair test" was considered to have only a very small percentage of the measurements that are in the stream actually causing rules to fire. It was considered fair to have short bursts of high-rate rule firings but not long-term sustained high-rate rule firings. NESTA is not intended for users to write rules to notify them by e-mail hundreds or thousands of times each second for a long and sustained period of time.

To summarize, NESTA sustained the above scenario for many cycles on the test-enhanced playback file without CPU starvation and without reporting any packet losses. The CPU utilization on the development workstation was about 90 percent prior to launch and higher than that after  $T - 0$ . It was heavily loaded, but NESTA maintained the pace. NESTA performed well considering that the data stream was stuffed with between one and two times the hypothetical peak load of measurement changes for the performance test. The “long pole” in the process appeared to be the number of rules that actually fired every second sustained. However, even under launch conditions when a heavy data change load exists, there are not expected to be many thousands of rules firing every second. Even several hundred rules firing per minute is considered unrealistically high, but this performance test suggests NESTA could readily handle that load.

## Development and Deployment

At the time of this writing, the customer had used NESTA for more than a year. Hundreds of rules have been written. Along with that, hundreds of NESTA notifications have been generated for multiple NASA engineers. These users have received both e-mails and wireless pages at KSC and other remote sites. Since the customer is a NASA engineer responsible for oversight of contractors, the notifications act as an extra set of eyes that further assure the quality of government oversight.

To better understand NESTA’s payoff, the responsibilities of NASA shuttle engineers must be examined. They include understanding their system and supporting equipment, knowing how their systems are tested and processed, being aware of when their systems are activated, tested, or in use, analyzing performance and data retrievals from any use of a system, and being ready to answer questions about their systems, such as: When was it tested? How did testing proceed? How did the data look? Is it ready to fly?

NESTA has helped shuttle engineers meet these responsibilities in varying degrees. In one recent usage, a shuttle avionics system was powered up over a weekend. The NASA shuttle engineer, being responsible for that system, would not have been aware that the system was powered up except for receiving a NESTA notification. In this case, the avionics user was not part of the shuttle engineer’s immediate organization. Thus, the shuttle engineer did not receive any communiqués regarding the system’s weekend usage. Due to NESTA, the shut-

tle engineer was better prepared to address questions about his system’s usage were they to arise. This has not been an uncommon occurrence. Shuttle engineers utilizing NESTA began realizing that some of their systems were being utilized much more than previously thought. Situational awareness increased markedly.

Aside from increased awareness, NESTA increases efficiency. Some ground operations span 24 hours and include dozens of asynchronous events that are broadcast on the data stream. For example, checkout of flight control hardware in the Orbiter Processing Facility occurred four to six times within the last year. The checkout included long hydraulic operations, powering up different parts of avionics, pressurizing/depressurizing the orbiter, and other work. During a recent flow, the NESTA notifications gave exact times of events of interest to the shuttle engineer. That allowed the shuttle engineer to quickly identify timelines of these lengthy operations. Effectively, a virtual roadmap identifying significant events was automatically generated, and that saved an hour of labor. More-efficient data retrievals resulted.

## Phased Approach to Implementation and Delivery

Multiple releases of NESTA have been delivered to the customer. The development team has four members each working approximately 50 percent of the time on the project. The team works very closely with the customer. Generally, the team meets with the customer at least once per week and has multiple other correspondences by e-mail and phone.

The initial NESTA release required six months. Thereafter, a release occurred approximately every other month. Prior to adopting Java and Jess, some preliminary performance testing was completed to verify that the Java language and Jess rule engine were fast enough to handle the shuttle data stream rates. Concurrent with that coarse performance testing, the initial set of requirements was being developed.

## Development Tools

In addition to Java and Jess, other tools were also used. Eclipse was used as an integrated development environment. Visio 2000 was used to develop Unified Modeling Language models. CVS was employed for configuration management. Ant was used for automating builds. JUnit was used for automated Java unit testing. Emma was used for Java code coverage including measurements and reporting. Finally, OptimizeIt by Borland was employed for profiling performance and detecting and isolating problems.

## Data Validity

As we indicated earlier in the article, the data stream is based on UDP. As such, the connection is not always reliable, and packets may get dropped, which poses problems when rules are waiting for data to arrive. Data health and validity become questionable. If the data stream connection is lost entirely or data becomes stale (that is, not updated), false positives or false negatives may result. That is, notifications of hardware events may never be sent or be sent in error.

To partially address this data validity issue, additional measurements are included in the rules to check for the validity of the stream. Measurements are now marked invalid for a dropped packet or packets or when the source of the measurement becomes bad. There is still a larger problem of false negatives and never receiving an e-mail if the data stream drops packets while a monitored event occurred. Aside from notifying the shuttle engineer of a data loss when it happens, we have not yet identified a mechanism that guarantees all notifications since the data stream is unreliable.

## Measurement Databases Changes

Multiple data streams and control rooms exist. Often, the measurement database, which is used to decode the SDS, dynamically changes on the stream as a result of operations. When that happens, decoding measurements becomes impossible and facts can no longer be updated in Jess's working memory. A short-term fix to this problem was to simply notify the NESTA system administrator when the stream changes. A measurement database Java bean was added and is used within a user rule as a fact. When the measurement database changes, the administrator automatically gets an e-mail and may restart NESTA accordingly. Longer term, automatic restarts of the agent will be provided.

## Flood of E-mails

If an end user incorrectly writes a rule, a possibility existed of flooding the network and servers with hundreds or even thousands of notifications. To prevent that, multiple safeguards, such as user-defined limits, were provided to filter e-mails after a given number have been generated for a particular e-mail account.

Beyond the possibility of user error, there was a separate need to queue e-mails that may be related to some sequence. Queuing provides a mechanism where multiple messages expected to occur within a short time period are grouped together before being e-mailed in bulk. For ex-

ample, four flight-control avionics boxes are often powered up in a short time period. Rather than a user receiving four separate flight-control e-mails that may be interrelated, it was necessary to provide a queuing mechanism that allows a user to tie related e-mails to the same queue and receive one bulk e-mail that was a compilation of what would otherwise be multiple e-mails. Both the queue time and queue length are configurable by the end user.

## Maintenance

New releases are delivered approximately every two months by the development team. Those releases may include bug fixes for problems reported in the former release. However, new releases are generally driven by new functionality as opposed to being driven by software errors.

The design of NESTA is conducive for updating by the end user. That is, the application uses a data-driven approach for the user files. All of the rules and facts are stored in Jess scripts. When rules have to be created or modified, the user has access to several text-based files. A facts file allows a user to add measurements that should be monitored. A rules file allows the entry of new rules. Since these are text-based script files, no compilation is required by the end user. The files are parsed at application startup. This data-driven approach is powerful in that it enables the end users to maintain their own files and not be at the mercy of the development team to add new support for new facts and rules.

## Conclusion and Future Work

NESTA has increased situational awareness of ground processing at NASA KSC. More and more shuttle engineers are relying on NESTA each month and are creating additional rules for monitoring the data stream. The infusion of AI technologies, particularly the Jess rule-based library, has proved very fruitful. Interfacing and integrating these modern AI tools with a legacy launch system demonstrates the scalability and applicability of the tools and paradigm.

The knowledge patterns that are evolving within NESTA will make it easier to train new users. More significantly, these patterns are influencing the design of a web-based, graphical user interface for creating and updating the rules. JavaServer Faces (Geary and Horstmann 2004), a web application framework, is being used to design and implement the web pages. JSF was chosen as it provides a standard com-

ponent application programming interface and targets the separation of presentation from business logic. An open source database management system, MySQL (DuBois 2005), is also being employed to create the database for storing the rules. MySQL supports modern database semantics including stored procedures, triggers, and foreign key relationships.

We are investigating agents that possess the ability to revise previously concluded assertions based on what may now be false or retracted data. Belief revision (de Kleer 1986), also known as truth maintenance, is particularly important when deep reasoning of long inferences is necessary. Jess currently has a very simple form of truth maintenance that we are looking to extend with a full-blown truth-maintenance system.

### Notes

1. The 2002 foundation for intelligent physical agents (FIPA) abstract architecture specification can be found at [www.fipa.org](http://www.fipa.org).
2. For the Java agent development framework (JADE), see [jade.tilab.com](http://jade.tilab.com).
3. For the Java bean specification from Sun Microsystems, see [java.sun.com](http://java.sun.com).

### References

- Bölöni, L., and Marinescu, D. C. 2000. An Object-Oriented Framework for Building Collaborative Network Agents. In *Intelligent Systems and Interfaces*, International Series in Intelligent Technologies, ed. H. Teodorescu, D. Mlynek, A. Kandel, and H. Zimmerman, 3: 31–64. Dordrecht, The Netherlands: Kluwer.
- Brownston, L.; Farrell, R.; Kant, E.; and Martin, N. 1986. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*. Reading, MA: Addison-Wesley.
- de Kleer, J. 1986. An Assumption-based TMS. *Artificial Intelligence* 28(2):127–162.
- DuBois, P. 2005. *MySQL*. Third Edition. Old Tappan, NJ: Pearson Education.
- Forgy, C. L. 1982. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence* 19(1): 17–37.
- Friedman-Hill, E. 2003. *Java Expert System Shell*. Greenwich, CT: Manning Publications.
- Gamma, E.; Helm, R.; Johnson, E.; and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Greenwich, CT: Addison-Wesley.
- Geary, D., and Horstmann, C. 2004. *Core JavaServer Faces*. Upper Saddle River, NJ: Prentice Hall.
- Gehman, H.; Turcotte, S.; Barry, J.; Hess, K.; Hallock, J.; Wallace, S.; Deal, D.; Hubbard, S.; Tetrault, R.; Widnall, S.; Osheroff, D.; Ride, S.; and Logsdon, J. 2003. *Columbia Accident Investigation Board (CAIB), Volume 1*. Washington D.C.: National Aeronautics and Space Administration.
- Russell, S., and Norvig, P. 2003. *Artificial Intelligence: A Modern Approach*. Second edition. Upper Saddle River, NJ: Prentice Hall.
- Wooldridge, M. 2000. *Reasoning about Rational Agents*. Cambridge, MA: MIT Press.
- Wygant, R. M. 1989. Clips: A Powerful Development and Delivery Expert System. *Computers and Industrial Engineering* 17(1): 546–549.
-  **Glenn S. Semmel** is a software lead within the Engineering Development Directorate at NASA KSC, where he heads efforts to infuse AI-based technologies for ground support of the space shuttle and future space vehicles. He is the technical lead of the NASA Engineering Shuttle Telemetry Agent project. His research interests include software agents, autonomy, model-based reasoning, decision-support tools, and automated verification and validation. He received his B.S. in electrical engineering, M.S. in engineering management, and M.S. in computer engineering from the University of Central Florida. He is currently pursuing a Ph.D. in computer engineering. Contact him at NASA Kennedy Space Center, Mail Code DX-E1, KSC, FL 32899; [Glenn.S.Semmel@nasa.gov](mailto:Glenn.S.Semmel@nasa.gov).
-  **Steven R. Davis** is a software engineer in the Engineering Development Directorate at NASA KSC, with an interest in safety-critical control systems. He holds a B.S. in electrical engineering from Auburn University, and is a member of the IEEE. He can be reached at [Steve.Davis@nasa.gov](mailto:Steve.Davis@nasa.gov).
-  **Kurt W. Leucht** is a software and test engineer in the Engineering Development Directorate at NASA KSC. He received his M.S. in space systems from the Florida Institute of Technology and his B.S. in electrical engineering from the University of Mis-
- souri-Rolla. He is a senior member of the IEEE. Contact him at NASA Kennedy Space Center, Mail Code DX-E4, KSC, FL 32899; [Kurt.Leucht@nasa.gov](mailto:Kurt.Leucht@nasa.gov).
-  **Daniel A. Rowe** is a software and test engineer in the Engineering Development Directorate at NASA KSC. He earned a B.S. in aerospace engineering from Iowa State University, an M.S. in mechanical engineering from the University of Central Florida, and an M.B.A. from the University of Central Florida. Contact him at NASA Kennedy Space Center, Mail Code DX-E1, KSC, FL 32899; [Daniel.A.Rowe@nasa.gov](mailto:Daniel.A.Rowe@nasa.gov).
-  **Kevin E. Smith** is the lead flight-control system test engineer within the Shuttle Processing Directorate at NASA KSC. His responsibilities include oversight of the preflight checkout of the space shuttle flight-control subsystems. He received his B.S. in electrical engineering in 1987 from the University of South Florida and his M.S. in electrical engineering in 1993 from the University of Central Florida. Contact him at NASA Kennedy Space Center, Mail Code PH-K2, KSC, FL 32899; [Kevin.E.Smith@nasa.gov](mailto:Kevin.E.Smith@nasa.gov).
-  **Ryan O'Farrell** is a software engineering intern in the Engineering Development Directorate at NASA KSC, and also a student currently pursuing a B.S. in software engineering at Auburn University. He can be reached at NASA Kennedy Space Center, Mail Code DX-E1, KSC, FL 32899; [John.R.Ofarrell@nasa.gov](mailto:John.R.Ofarrell@nasa.gov).
-  **Ladislau Bölöni** is an assistant professor at the Computer Engineering Department of the University of Central Florida. He received a Ph.D. degree from the Computer Sciences Department of Purdue University in May 2000. His research interests include autonomous agents, grid computing, and wireless networking.