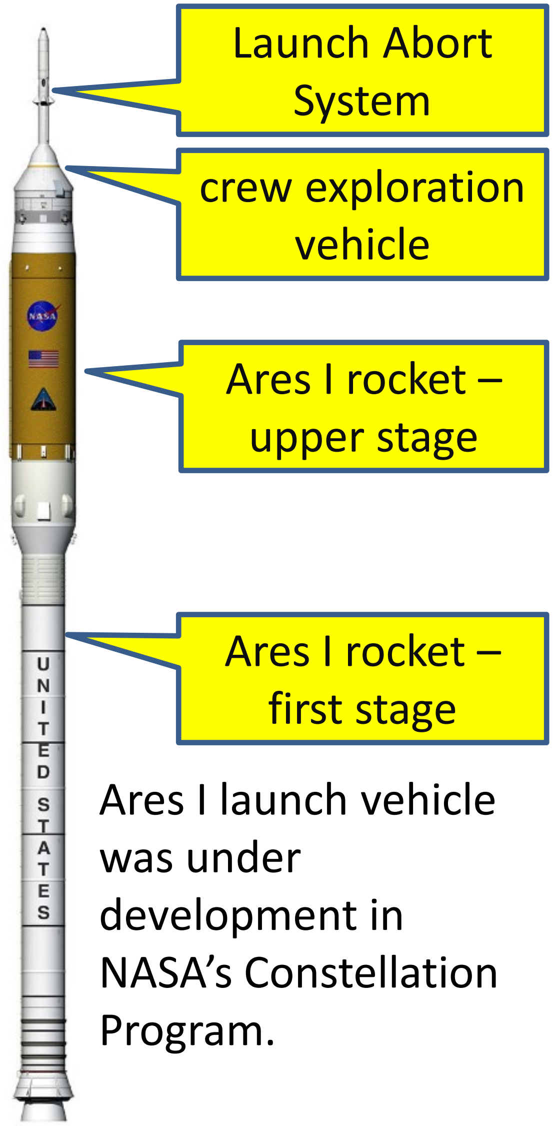
Architecting and Generalizing a Safety Case for Critical Condition Detection Software

Martin S. Feather , Jet Propulsion Laboratory, California Institute of Technology
Lawrence Z. Markosian, SGT, Inc, NASA Ames Research Center

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, and at NASA Ames Research Center, under a contract with the National Aeronautics and Space Administration.
The work was sponsored by the NASA Office of Safety and Mission Assurance under the Software Assurance Research Program. This activity is managed locally at JPL through the Assurance and Technology Program Office.

NASA's Launch Abort System for the crew exploration vehicle

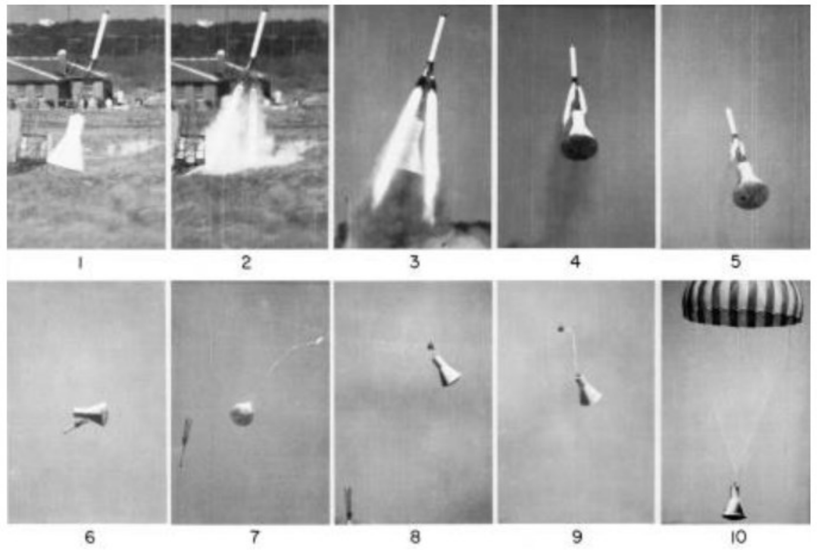


Ares I launch vehicle was under development in NASA's Constellation Program.

In the event of an emergency on the launch pad or during ascent, the Launch Abort System separates the crew exploration vehicle so it can parachute down to ground.

Similar capability planned for NASA's Space Launch System (currently in development).

Launch abort systems not new – images below of abort test of Mercury capsule



NASA Identifier: L59-2768

What IS new: the role of **software** in helping determine the need for an abort

Critical Condition Detection

Key requirements:

- Determine when a critical condition is present (typically within some time bound). Failure to do so is termed a "false negative" (in statistics, a "type II error").
- When a critical condition is not present, don't mistakenly determine it is.
  Such a mistaken determination is termed a "false positive" (in statistics, a "type I error"; informally, a "false alarm")
  (both important; we focus on "false +ve)

Commonplace examples of "false positives" in critical condition detection:

- Determination of the need to deploy a car's airbag – only when necessary!
- Medical device alarms in hospitals:
  "…It is estimated that between 85 and 99 percent of alarm signals do not require clinical intervention, such as when alarm conditions are set too tight; default settings are not adjusted for the individual patient or for the patient population; ECG electrodes have dried out; or sensors are mispositioned. ..." [Medical device alarm safety in hospitals, Sentinel Event Alert, Issue 50, The Joint Commission, April 2013]
- Alarm clock goes off early (OK, OK, maybe it's not all that critical in most cases)

Challenging* aspects of Critical Condition Detection for Launch Abort

- Multiplicity of “abort conditions” (not just one)
- Timeliness: some abort conditions progress rapidly from their onset, leaving little time for detection and the response that follows
- Detection of abort conditions:
  - Constrained by what sensors can be utilized
  - Transients – addressed by affirmation of the abort condition by measuring similar sensors multiple times
  - Localized/misleading phenomena – addressed by affirmation of the abort condition by measuring dissimilar sensors or assessing the health status of other vehicle components or subsystems
- Imperfect sensors – addressed by sensor data qualification to “monitor a network of related sensors to determine the health of individual sensors within that network”

*See “An Abort Failure Detection, Notification, & Response System: Overview of an ISHM Development Process”, Pisanich et al., 2008 IEEE Aerospace Conference, and “Sensor Data Qualification for Autonomous Operation of Space Systems”, Maul et al., *AAAI Fall Symposium on Spacecraft Autonomy: Using AI to Expand Human Space Exploration,* Arlington, VA, October 2006

Launch Abort System's Software

Abort Failure Detection, Notification, and Response (AFDNR)

The AFDNR function is a part of the Ares I Flight Computer flight software designed to recognize Ares I conditions that require Orion to manually or automatically issue the abort command to initiate the abort sequence to separate Orion from the Ares I vehicle, and notify Orion of the abort recommendation with sufficient time to safely separate the Orion from the Ares I.

…

Upon recognition of a the need for an abort, AFDNR notifies Orion (including the crew), Mission Systems and Ground Systems (pre-launch only). Furthermore, if the abort condition is time-critical, AFDNR contains logic to autosafe the system.

Our work on Safety Cases

We were developing a Safety Case for AFDNR's "False Positive" requirement (an upper limit on the probability of False Positives), and draft training material based on this.

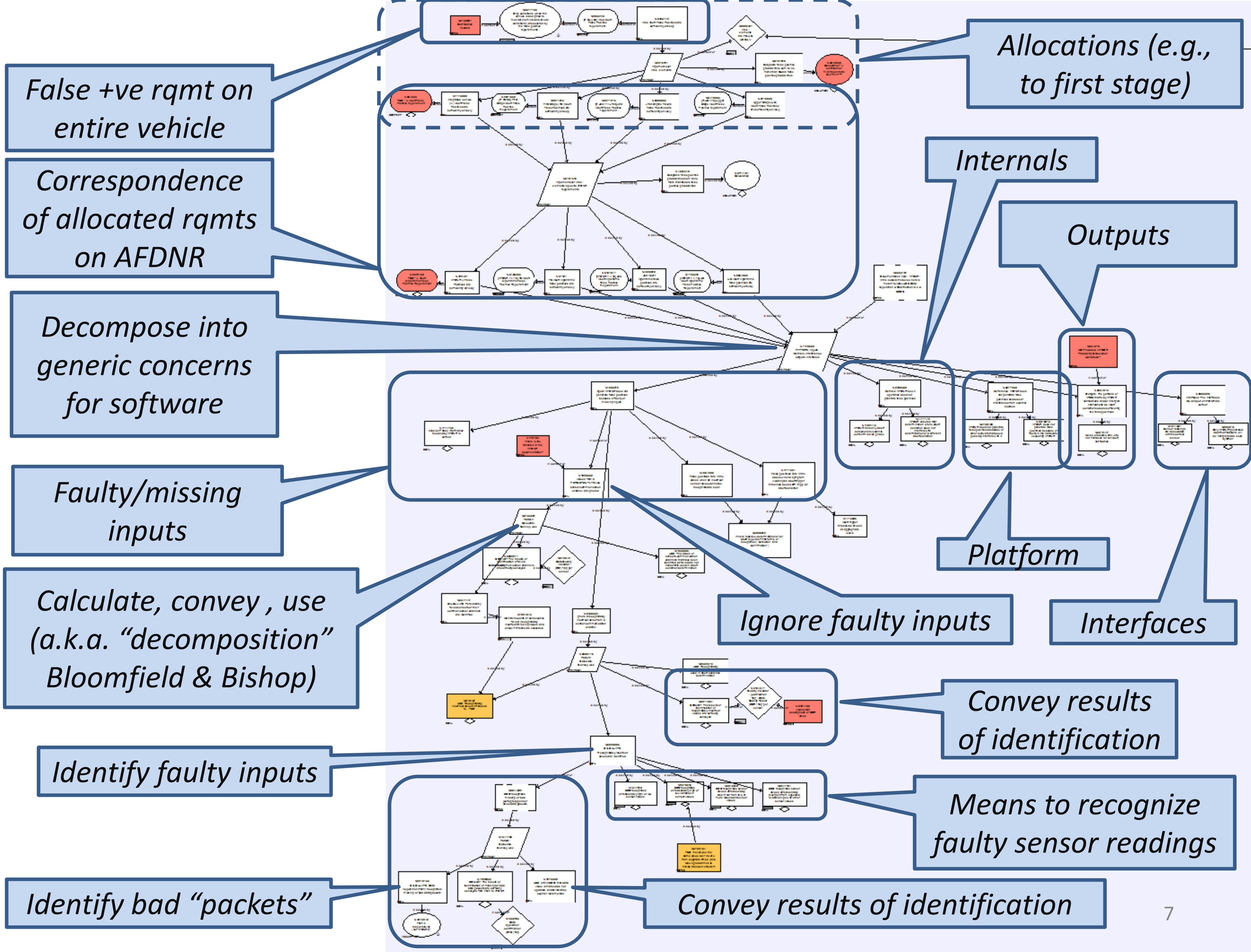
- A **pilot project** to investigate safety cases for NASA software
- Obviously safety critical
- "False Positive" requirement our focus
- Access to developers [to whom we express our thanks]

"Risk-Informed Safety Case" in NASA System Safety Handbook, Nov. 2011

But… NASA's Constellation program cancelled partway through so AFDNR progressed only through to design, as did our safety case.

Our initial AFDNR Safety Case

- We designed the AFDNR safety case based on:
  - the explicit safety requirements allocated to AFDNR,
  - the implicit requirements on AFDNR components that we extracted from the FDNR System Design Document (SDD), &
  - evidence, primarily from testing of an executable prototype.
- Our focus: AFDNR algorithms
  - Not flight software
- The availability of documentation:
  - A thorough SDD was available
  - Software safety analysis documentation was <u>un</u>available


False +ve rqmt on entire vehicle
Allocations (e.g., to first stage)
Correspondence of allocated rqmts on AFDNR
Internals
Outputs
Decompose into generic concerns for software
Faulty/missing inputs
Platform
Calculate, convey , use (a.k.a. “decomposition” Bloomfield & Bishop)
Ignore faulty inputs
Interfaces
Convey results of identification
Identify faulty inputs
Means to recognize faulty sensor readings
Identify bad “packets”
Convey results of identification

In Progress – generalizing and re-architecting the case

AFDNR details restricted, **yet basic issues of critical condition detection are generic**.

We now appreciate how the organization of our safety case could be improved – had most everything we THINK is needed somewhere, but disorganized

Hence our work to re-architect (“refactor”?) and generalize our safety case, and the training material based upon it.

Overall Architecture

Generic software hazards

Allocation pattern

False +ves < N%

Subsystem$_S$ allocations of False +ves < N$_S$%

Allocation pattern

Condition$_C$ allocations of False +ves < N$_{SC}$%

Detection Hazards & their Mitigation pattern

Transients

Localized phenomena

Etc.

Faulty sensor readings

Transients escape identification < N$_{sct}$%

Identify & ignore

False **-ve** requirement on identification subsystem

Detection specific

+

External interference with software hazards

Software-system interface hazards

Inter-software hazards

Intra-software hazards

Software development waterfall

N-in-a-row Mitigation **requirement/spec**

N-in-a-row Mitigation **design**

N-in-a-row Mitigation **source code**

N-in-a-row Mitigation **executable**

Calculate-convey-use pattern

Conclusions

AFDNR detection role an instance of a potential widely useful "software safety case architecture template" for critical condition detection
(e.g., Nguyen & Ellis's "Safe Mode" – "Experiences with Assurance Cases for Spacecraft Safing" ISSRE 2011)
More such templates a good thing?

Safety case structure parallels the NASA (and other) software safety processes – is there are generic super-pattern?

Not sure how to accommodate quantitative risk into the software portions
- Software reliability estimation itself
- Software hazards that aren't readily localized and/or are addressed globally

Design rationale was present in the System Design Document, but deep within the 600 page document.
Some interesting insights emerged, e.g., the safety case for freedom from false positives case places reliance on a subsystem's freedom from false negatives!