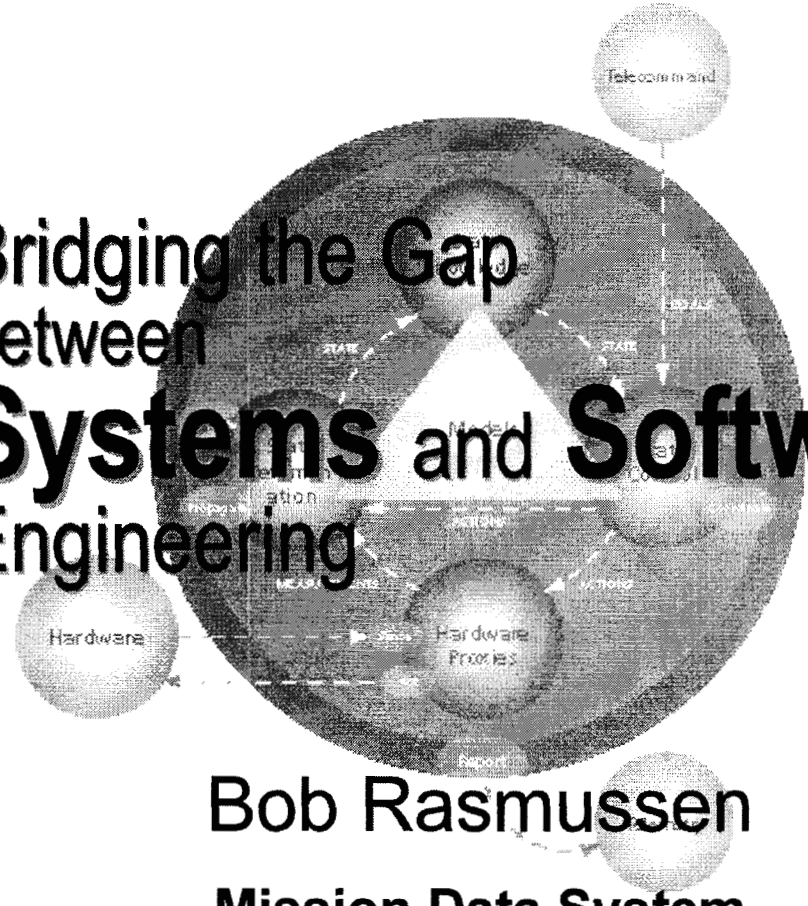
# Bridging the Gap between Systems and Software Engineering




Bob Rasmussen

**Mission Data System**

This document contains Caltech/JPL confidential information.

Copyright 2002, by the California Institute of Technology. ALL RIGHTS RESERVED. United States Government Sponsorship Acknowledged. Any commercial use must be negotiated with the Office of Technology Transfer at the California Institute of Technology.

This document may be subject to US export control laws and regulations. By accepting this document, the user agrees to comply with all applicable U.S. export laws and regulations, including the International Traffic and Arms Regulations, 22 C.F.R. 120-130 and the Export Administration Regulations, 15 C.F.R. 730-744. User has the responsibility to obtain export licenses, or other export authority as may be required before exporting such information to foreign countries or providing access to foreign persons.

# Different Worlds

# There Is a Big Gap Between...

- What **Systems** Engineers do:
  - Define and analyze the capabilities a system must have
  - Establish the decomposition of functionality
  - Provide key algorithms for the accomplishment of these functions
  - Integrate, test, operate, and maintain a system
- And what **Software** Engineers do:
  - Define the software architecture of a system
  - Provide the tools and techniques for software development
  - Design and build and test the software to provide the required functions

# Different Languages, Different Methods

- Systems engineering is ***outward*** looking
  - Mission scenarios
  - Functional decomposition
  - System analysis
  - Performance requirements
  - Resource allocations
  - Command and telemetry dictionaries
  - Flight rules and constraints
  - Control laws
  - Failure modes analysis
  - Fault protection
  - Test procedures
- Software engineering is ***inward*** looking
  - Languages, libraries, operating systems…
  - Concurrent threads, processes, memory management…
  - Real time execution
  - Patterns, abstractions, general algorithms…
  - Data representation, serialization…
  - Interprocess communication
  - Deadlocks, access violations, exceptions…

# What's Wrong With This?

- If these two groups are genuinely doing different unrelated things, maybe not much
  - It has worked — more or less — so far
  - But only when …
    - Systems engineers understand the details of what the software does
    - Software engineers understand the details of what systems engineers want
- The principle risk is **miscommunication**
  - What systems engineers want can be **hard to express**
  - What software engineers build can be **hard to understand**

# This Risk Is Increasing

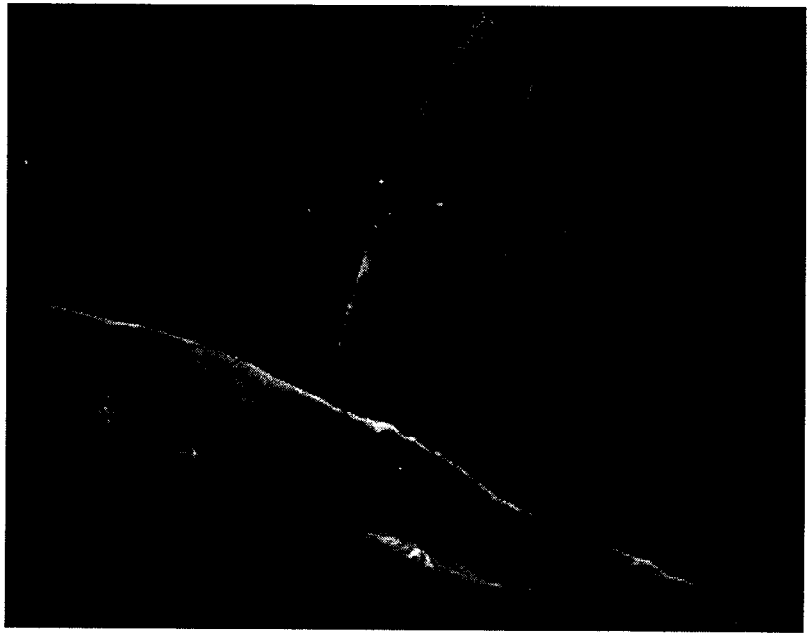
- Trends toward **rapidly increasing complexity** raise serious doubts about the future

- **Interactions** make both **systems** and **software** difficult
  - Elements that work separately often fail to work together
- The **combinatorics** of interaction is staggering
  - So it's not easy to get right
- This is a major source of **unreliability**

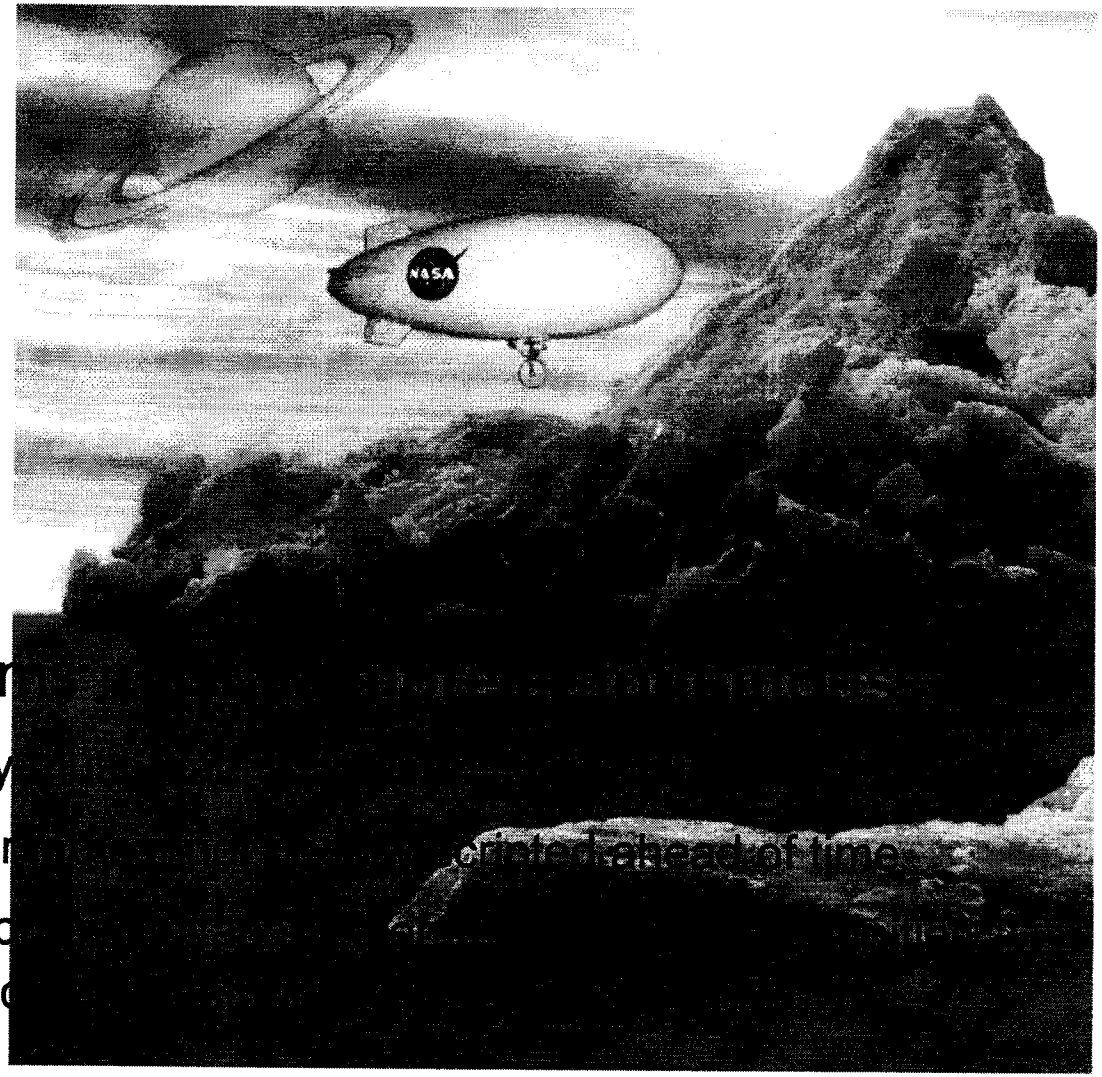
# Forces Driving Change

# Future Missions Will Involve...

- ***In situ* exploration** of the solar system with vehicles on and under surfaces, in atmospheres, and on and around small bodies
- **Multi-vehicle coordination** for formation flying, rendezvous and docking, and so on
- **More complex observatories** with very demanding operational requirements
- An **interplanetary network** requiring multi-project coordination of resources scattered around the solar system



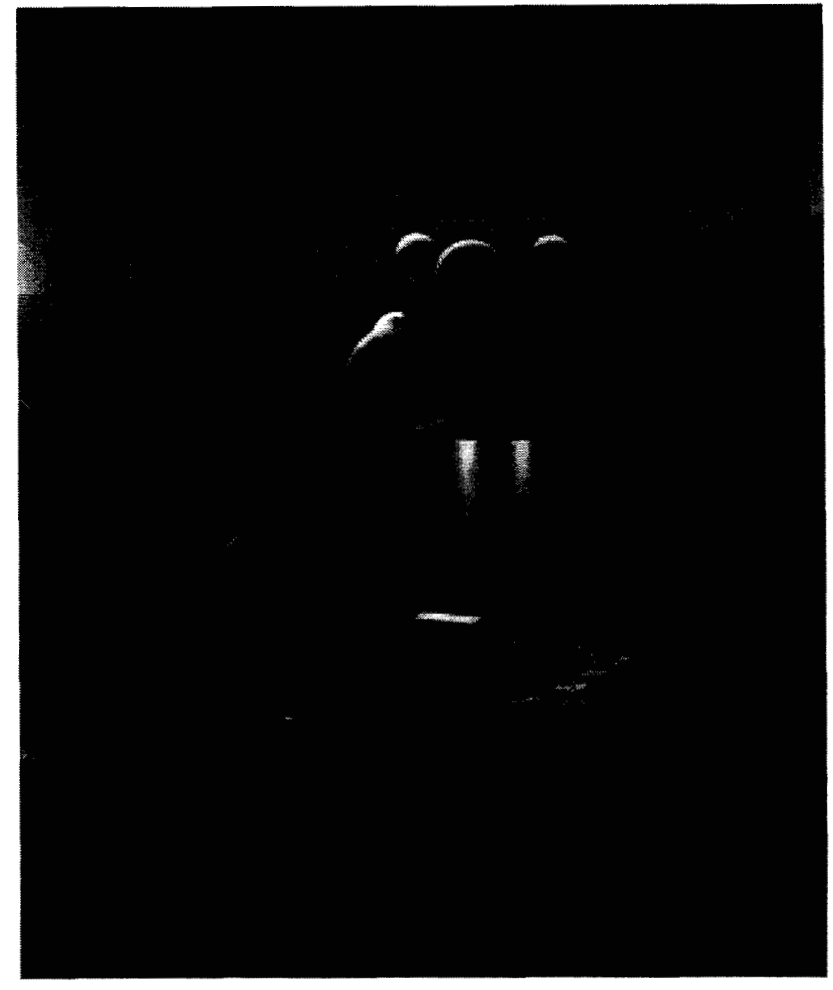
# Missions Are Becoming More Complicated




- Syster
  - They
  - Their
  - Resc
    and c

# Missions Are Becoming More Complicated

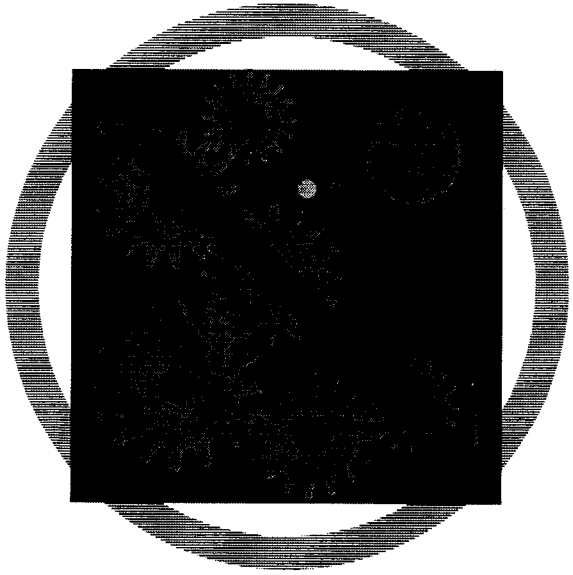
- Systems must do **more complex tasks**
  - They will interact with and alter their environment
  - Multiple activities will compete with system safety for time and resources
  - Opportunistic science will alter plans
  - Data may have to be intelligently screened onboard before transmission



# Systems Engineering Is Getting More Complex

- **Predictable**, modal designs **will not work** as well for the future
  - In situ environments and complex systems won't cooperate
  - Systems must be specified and operated more abstractly

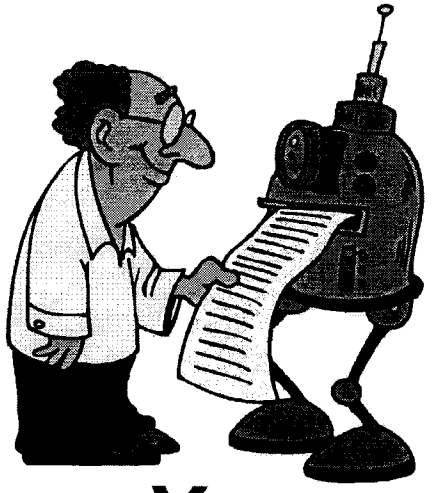
- **Success** must be assured, **despite uncertainty**
  - Software will be called upon to make this happen



**No**

# Systems Engineering Is Getting More Complex

- We are **more dependent on modeling** for system level analysis and verification of designs
  - Systems will be **specified** increasingly in the form of models
  - Correctness will be **verified** increasingly against models of expected performance
- Explicit models are **vital to advanced technologies**
- Systems engineers are responsible for the correct models to use
  - Consistency and interoperability of models will be important
  - They must avoid contradiction, confusion, and confounding constraints



**Yes**

# Software Engineering Is Also Getting More Complex

- Software systems are **increasingly bigger** with **much more functionality**
  - More and more disciplines previously confined to the ground are moving onboard
  - Greater mission complexity drives performance and autonomy
  - Autonomous behaviors require software developments to be integrated much more intimately
- **Reduced schedules and budgets**
  - We must try to realize operational cost savings through more automation
  - We must design for reuse, and then reuse what we design
  - We must standardize and consolidate architectures to share solutions to common problems

# Software Engineering Is Also Getting More Complex

- Small scale, low level implementation stratagems have to give way to more robust methods applied at a **larger scale**
  - Potential interactions grow disproportionately ($\approx n^2$) with size
  - Higher level abstractions must build upon solid foundations
- Complex behaviors depend on **automated reasoning** about the system
  - Reasoning requires understanding
  - Model-based technologies are the key

# Where Is This Leading?

- The line between **specifying** behavior and **designing** behavior is blurring

**System engineers need to know and want to specify …**

**Software engineers want to design software that knows …**

- How a system is put together (connections and other interactions)
- What functions each element performs (models of behavior)
- How system elements might fail (models of faulty behavior)
- What the environment is like and how it affects the system (more models)
- What the system must be able to do (scenarios and their objectives)
- What operating constraints the system must honor (flight rules, etc.)
- What resources the system must manage (power, data storage, etc.)

*And so on*

# Finding a Common Language

# The Common Expression of These Needs Is in Models

- Knowledge in models is ***accessible for inspection***
  - Providing algorithms and rules is not sufficient
  - Assumptions and beliefs must be exposed

*However…*

- Models must be ***verifiable for consistency***
  - Design, analysis, simulation, planning, and so on require different forms
  - Consistency with one another and with the actual system is vital

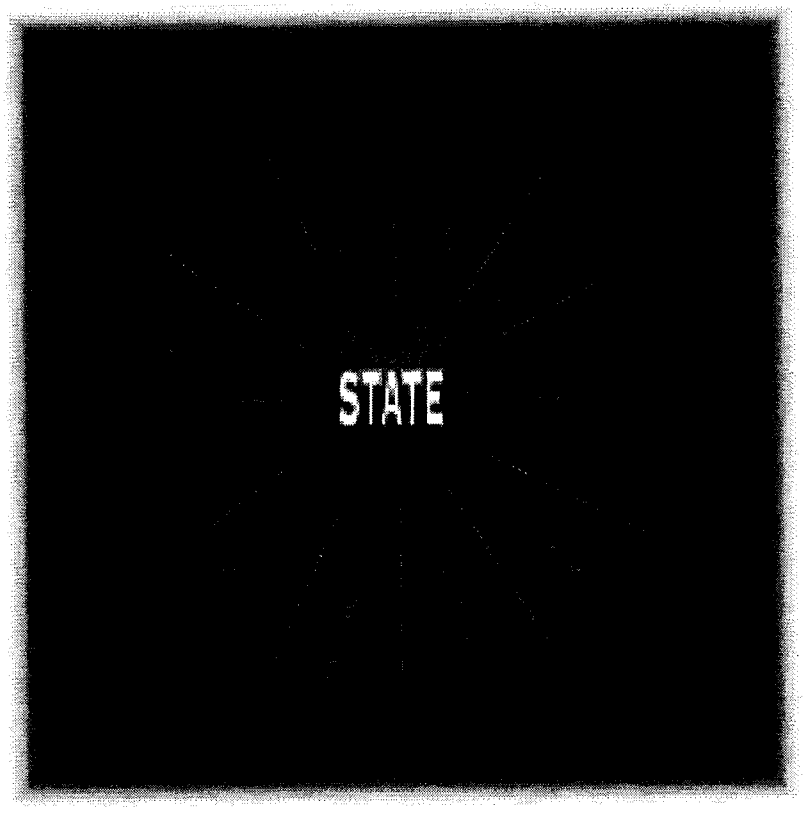
*Therefore…*

**There must be a coherent basis for merging models from around the system**

# State is Central

## Models must share a common definition of system states

- Formally recognize **state** as the key system concept
- Make **state** the central organizing theme for most functions
- Express all knowledge in models of **state**



# An Example

# Imagine Driving Your Car

- As the driver, you…
  - Have destinations and deadlines, and desires for a safe and comfortable trip
  - Plan a route, make sure you have enough fuel or stop along to way to get more, schedule the trip to meet your appointments
  - Rely on gauges and your own senses to put together a comprehensive set of beliefs about what's happening
  - Turn on your headlights or wipers if you have to, observe the traffic laws, navigate around obstacles, and adjust your driving to the weather conditions
  - Have expectations about other drivers, your car's performance, and so on, to inform your moves

# In Other Words...

- You set objectives regarding the **state** of the world
- You monitor the **state** of the world
- You form a coherent notion of the **state** of the world and anticipate its changes
- You adjust your intended actions to achieve a new world **state** that meets your needs
- You control the **state** of the car, which is part of the world

# Examples of Car Driving States

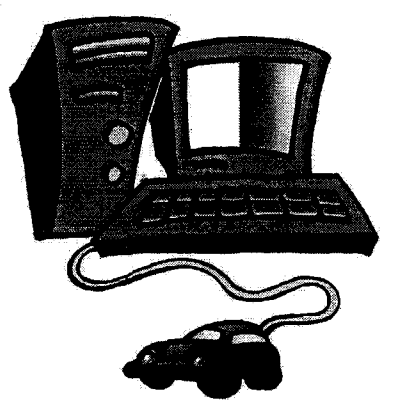
- Where you and neighboring vehicles are and how fast you're driving
- The wear condition of your wiper blades, brakes, tires, and so on
- How much fuel you have
- Ignition and power system status
- Engine speed and transmission gear selection
- Brake and accelerator position, and steering angle
- Temperature, oil pressure, battery charge, and so on
- Headlight status, like on or off, and high or low beam
- Windshield wiper speed
- Where the sun is, and whether it's night or day
- Weather effects like wind, slipperiness, and visibility
- What station your radio is tuned to

# Examples of Car Driving Models

- How the direction of travel depends on steering angle
- How fuel depletion depends on driving speed
- How battery charge varies with engine speed, headlight status, etc.
- How the fuel gauge reading depends on fuel depletion and whether or not the ignition switch is on
- How the pitch of sound from the engine varies with engine speed
- How stopping distance depends on driving speed, pavement slipperiness, and brake pressure
- How the sun moves from east to west every day
- How headlights improve visibility when the sun is down
- How windshield wipers improve visibility when it's raining
- How performance varies with gear selection and driving speed
- How instrument panel gauges don't always read accurately (in my car anyway)
- How the instrument panel gauges sometimes work more reliably when you bang on the dashboard

# Now, Imagine *Software* Driving Your Car

## What would the software have to know? What would it have to do?



- It would have to know…
  - All the states described before — and more
  - All the models described before — and more
- It would have to…
  - Set objectives regarding the **state** of the world
  - Monitor the **state** of the world
  - Form a *coherent* notion of the **state** of the world and anticipate its changes
  - Adjust its intended actions to achieve a new world **state** that meets its needs
  - Control the **state** of the car, which is part of the world

# That Is...

***Software* must know and do the same things *you* do**

- ***Software*** needs models
- These are the *same* models ***systems*** engineers use

*Consistent use*
*of a common notion of* ***state***
*is essential*

# Putting a Common Language to Work

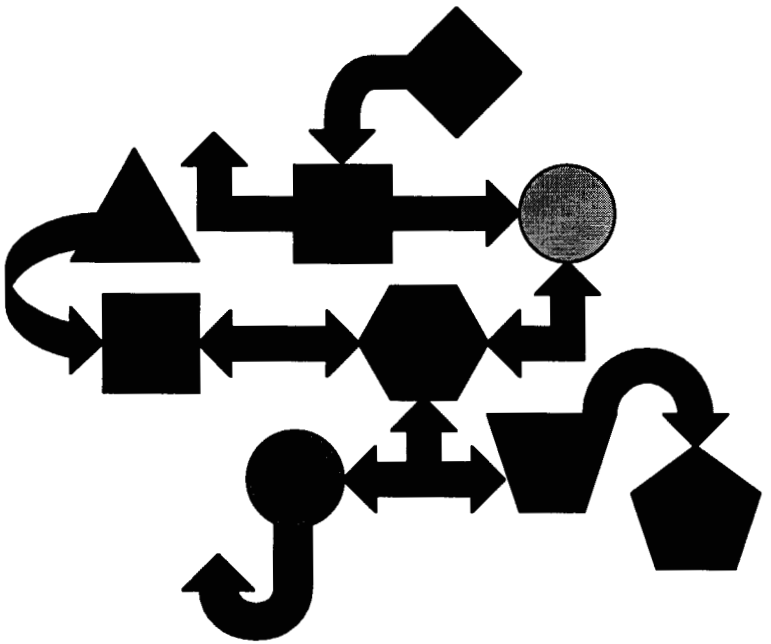
# What's New?

- System Engineers have always analyzed systems in terms of their state
- Software Engineers have always represented system state in software

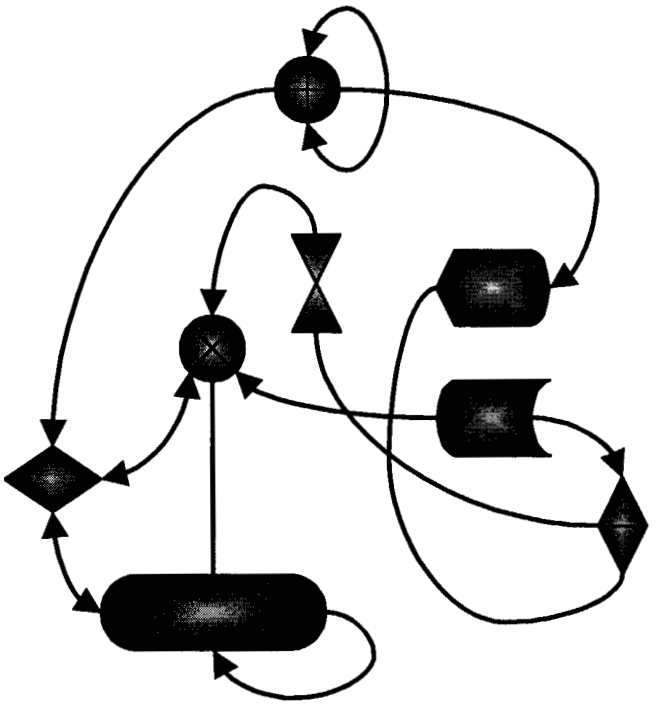
***However ...***

# Different Languages, Different Methods

Systems



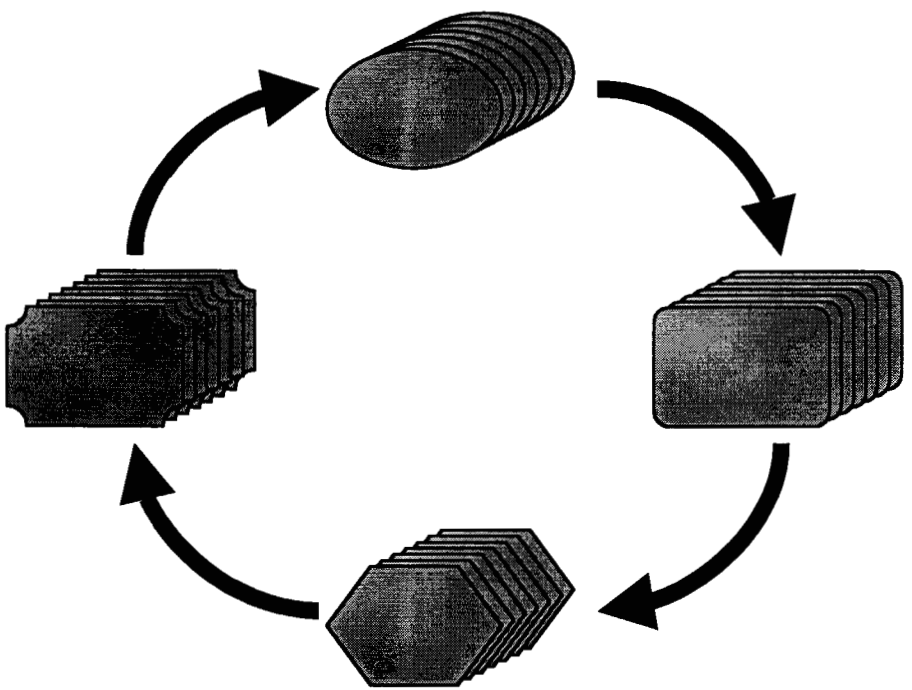
Software



**There has generally been little uniformity within these areas, let alone between them**

# One Language, One Method

Systems *and* Software



**A *common* language enables a *simpler, uniform*, *shared* organization of the effort in *both* areas**

# What's Needed?

- More than just *talk* about state and models
- An concrete, organizing **framework**

**Common building blocks**

**Common interaction methods**

**For both systems and software engineers**

- One method for accomplishing this is called…

# "State Analysis"

# State Analysis:

- A uniform, methodical, and rigorous approach to…
  - Discovering, characterizing, representing, and documenting the **states** of a system
  - Modeling the **behavior** of states and **relationships** among them
  - Capturing the mission **objectives** in detailed **scenarios**
  - Keeping track of system **constraints** and operating **rules**
  - Describing the **methods** by which objectives will be achieved
  - Recording information about hardware **interfaces** and **operation**
- For each of these design aspects, there is a simple but strict **structure** within it is defined
- This structure is comprised of

**Common Framework Elements**

# Common Framework Elements

Goals

Time

Measurements

Estimators

State Constraints

Scenario Fragments

Various Models

Resources

Controllers

Commands

State Variables

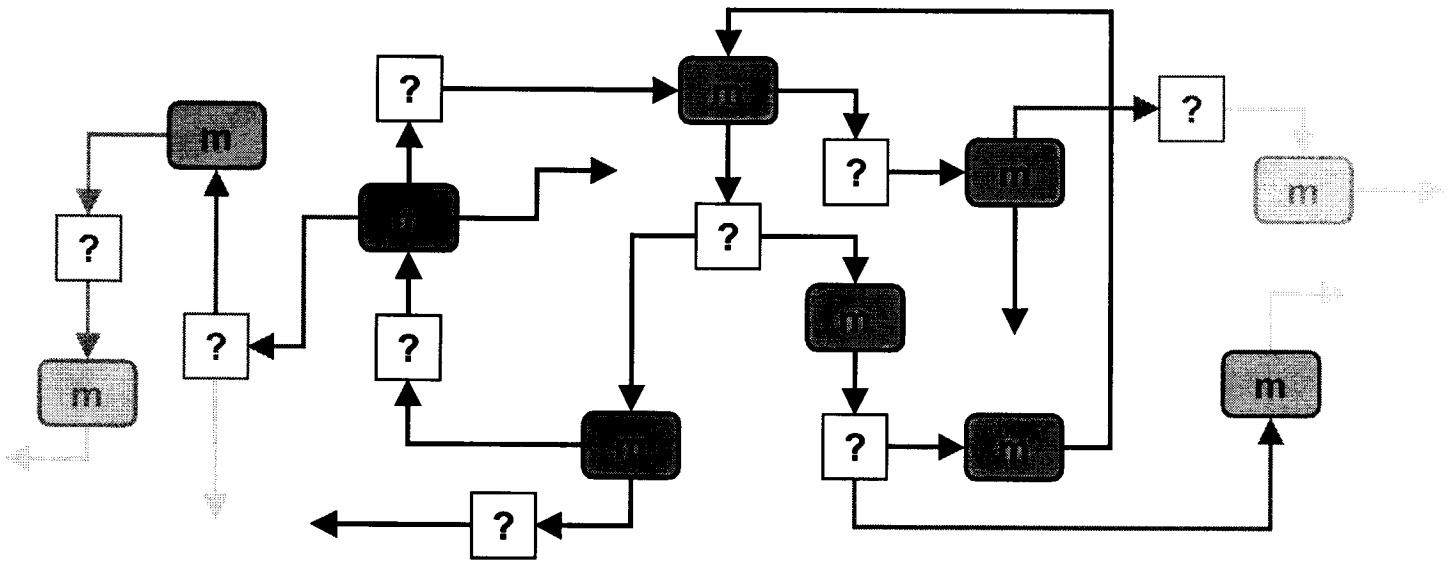
Time Constraints

Allocations

... and a few more

# State Analysis is Recursive

- It is a gradual discovery process, prompted by a **standard** set of basic **questions**
  - The answer to each question is a piece of the model




  - Each answer prompts additional questions, and so on
  - The model unfolds a step at a time in terms of **common framework elements** until all the pieces are identified

# Sample Standard Questions

- What states influence the value of a measurement?
- How will the value of a state be determined?
- How are inconsistencies among various sources of data reconciled?
- How is state knowledge uncertainty represented?
- Under what conditions does a command act as desired?
- What conditions must be true prior to an activity?
- What are the success criteria for an activity?

… and so on

# Following Leads — An Example

## Standard Questions:

What do you want to achieve?
*Move rover to rock*

What's the state to be controlled?
*Rover position relative to rock*

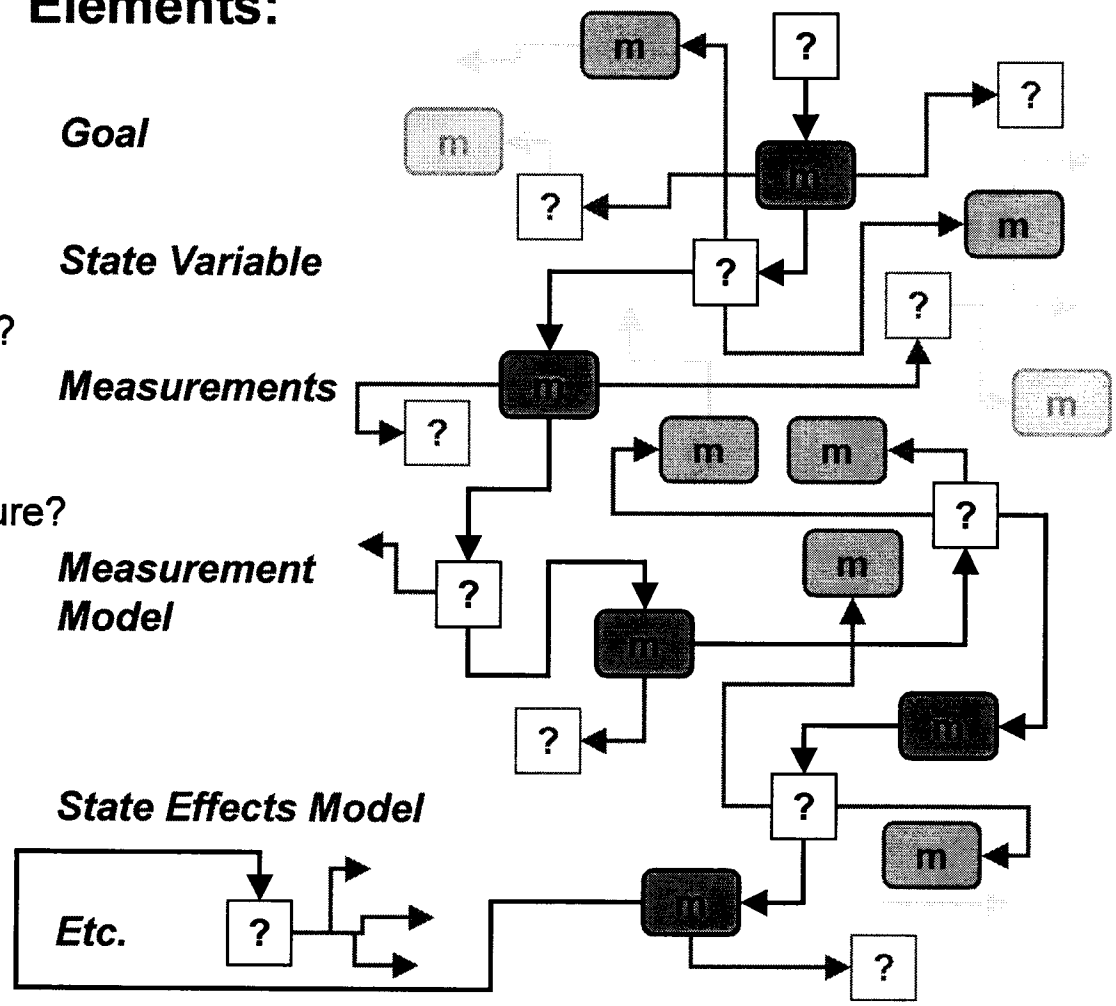
What evidence is there for that state?
*IMU, wheel rotations, sun sensor, stereo camera*

What does the stereo camera measure?
*Distance to terrain features, light level, camera power (ON/OFF), camera health*

How do you raise the light level?
*Wait until the sun is up*

Where is sun relative to horizon?
...

## Common Framework Elements:



# Spacecraft States

- Dynamics
  - Vehicle position & attitude, gimbal angles, wheel rotation, ...
- Environment
  - Ephemeris, light level, atmospheric profiles, terrain, ...
- Device status
  - Configuration, temperature, operating modes, failure modes, ...
- Parameters
  - Mass properties, scale factors, biases, alignments, noise levels, ...
- Resources
  - Power & energy, propellant, data storage, bandwidth, ...
- Data product collections
  - Science data, measurement sets, ...
- DM/DT Policies
  - Compression/deletion, transport priority, ...
- Externally controlled factors
  - Space link schedule & configuration, ...

... and so on

# Spacecraft Models

- Relationships among states
  - *Power varies with solar incidence angle, temperature, & occultation*
- Relationships between measurement values and states
  - *Temperature data depends on temperature, but also on calibration parameters and transducer health*
- Relationships between command values and states
  - *It can take up to half a second from commanding a switch to full on*
- Sequential state machines
  - *Some sequences of valve operations are okay; others are not*
- Dynamical state models
  - *Accelerating to a turn rate takes time*
- Inference rules
  - *If there has been no communication from the ground in a week, assume something in the uplink has failed*
- Conditional behaviors
  - *Pointing performance can't be maintained until rates are low*
- Compatibility rules
  - Reaction wheel momentum cannot be dumped while being used for control

... and so on

# Resulting Software Elements

- Each **State** and **Model** is represented directly in code

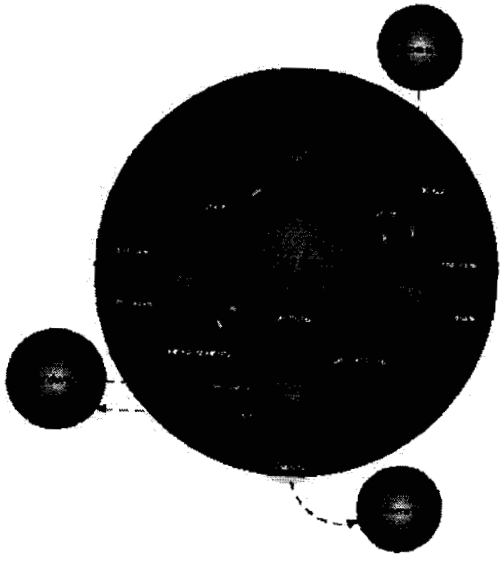
- In addition, there is a ***direct*** **mapping between requirements and implementation** of
  - Algorithms
  - Data structures
  - Scenario elements

... and so on

# Advantages on Many Fronts

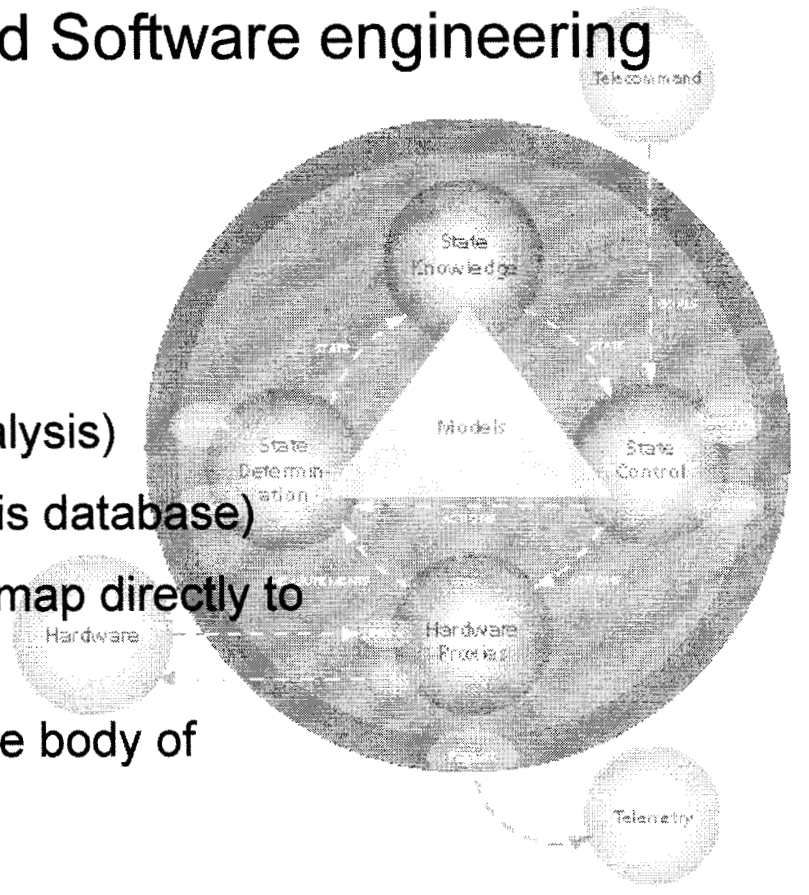
- Captures thorough, unambiguous **requirements**
- Guides a clear **work breakdown**
- Aids collection of **metrics**
- Fosters a **robust design** approach
- Aids cross-checking for **coverage** and **consistency**
- Serves as an **integration** tool at many levels
- Improves **inspectability** and **testability**
- Enables principled **coordination** of the system
- Facilitates increasing **autonomy**
- Assures greater **reusability**

# Putting These Ideas to the Test



# MDS

- State Analysis was created and refined by MDS as part of a joint Systems and Software engineering effort at JPL



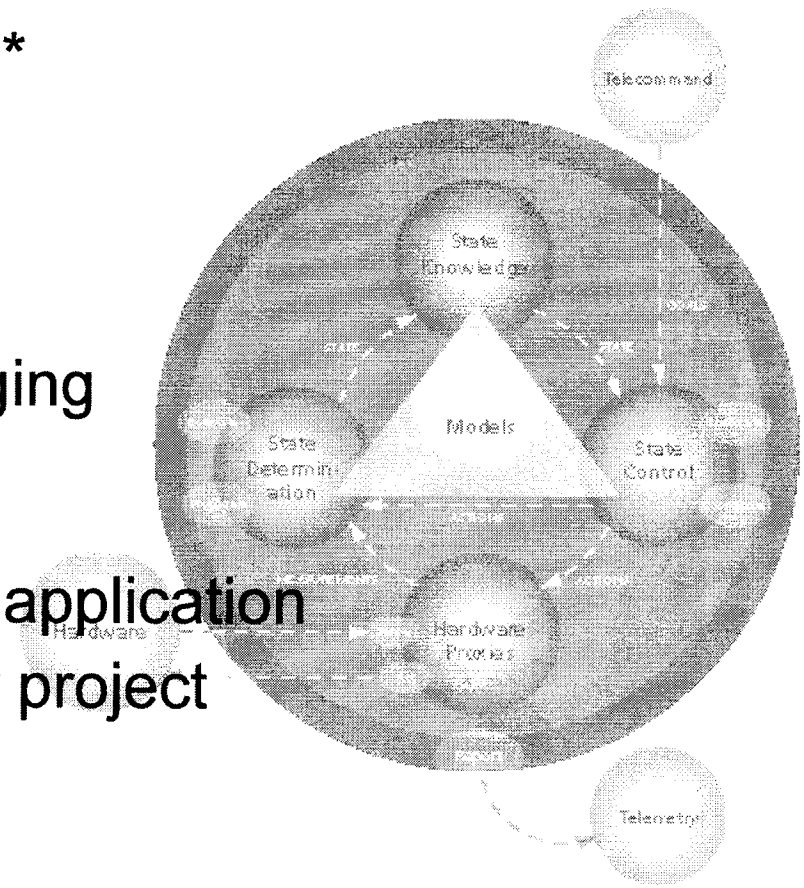

- MDS is comprised of
  - **Processes** (such as State Analysis)
  - **Tools** (such as a State Analysis database)
  - **Architecture** (state-based, to map directly to State Analysis products)
  - **Framework** (a large, extensible body of reusable core software)

# Trial Applications, So Far

- A few interplanetary "cruise" scenarios
- Entry, descent, and landing*
- A surface rover*

- Results so far are encouraging

- MDS is slated for first flight application on the **Mars Smart Lander** project




* In development