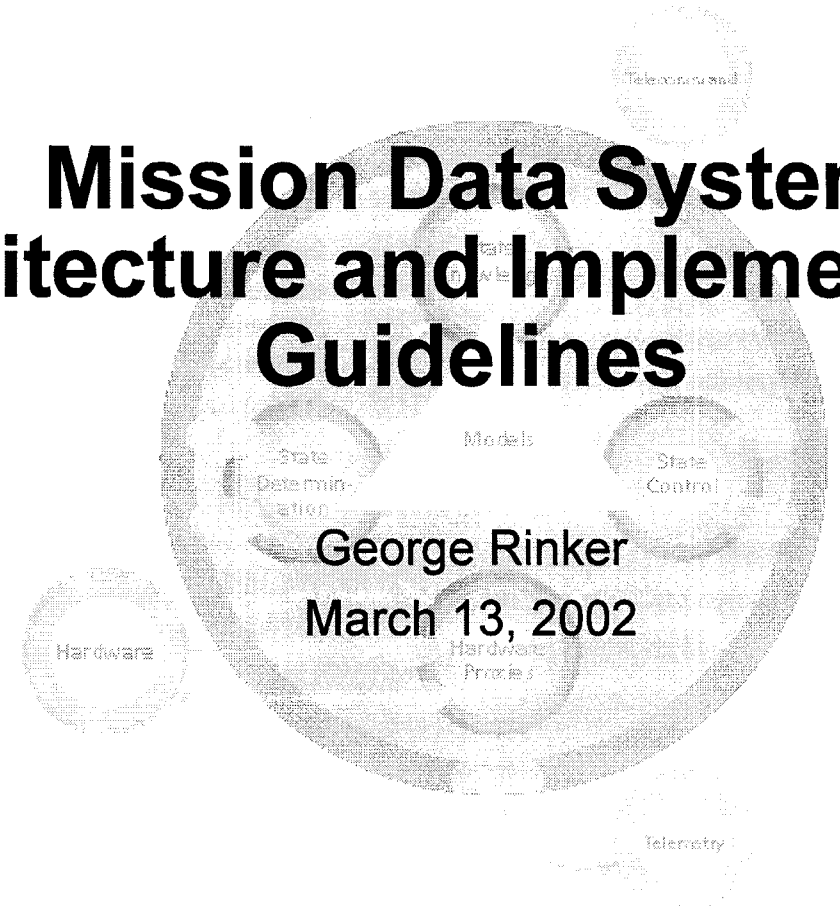


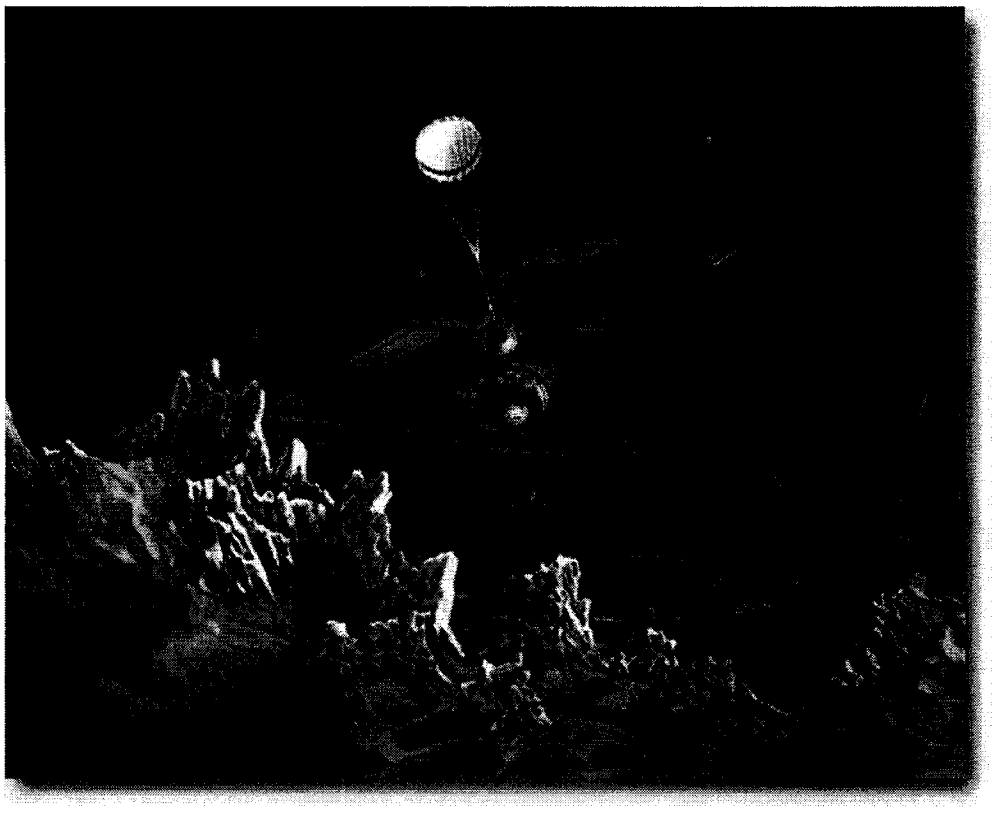
Mission Data System Architecture and Implementation Guidelines

George Rinker

March 13, 2002

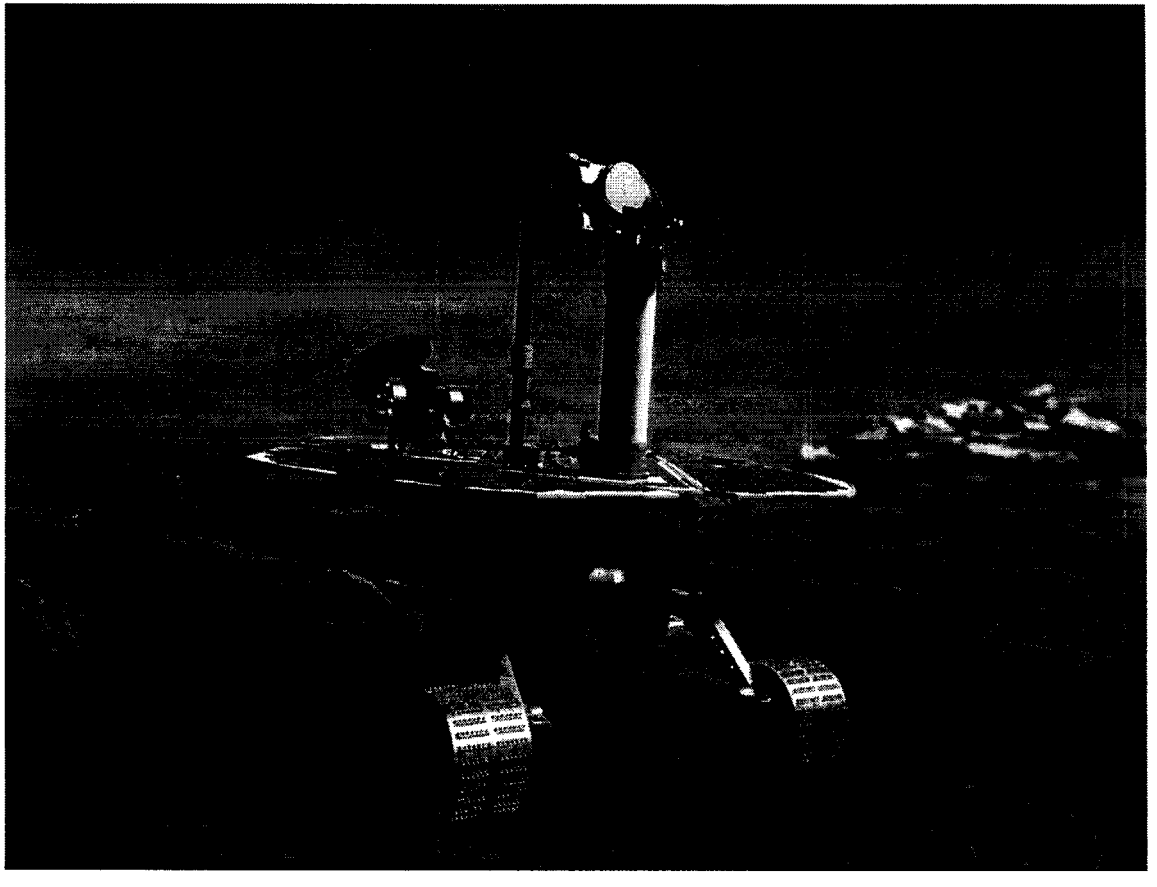
Historical Context

- Until recently, JPL missions were one-of-a-kind, spaced many years apart
- Missions have been designed for human control from Earth
- Flight software has used relatively simple time-based sequencing
- Very little autonomy except for fault protection and a few "critical sequences"
- There is a big gap between systems engineering and software engineering



Pressures for Change

- New era of frequent launches
- More *in situ* operations in uncertain environments
- More constrained communication with Earth demands more onboard decision-making
- Specter of mission-ending failures due to errors in software



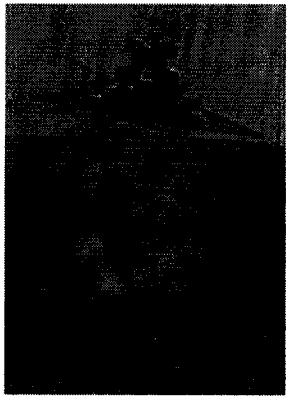
The MDS Vision

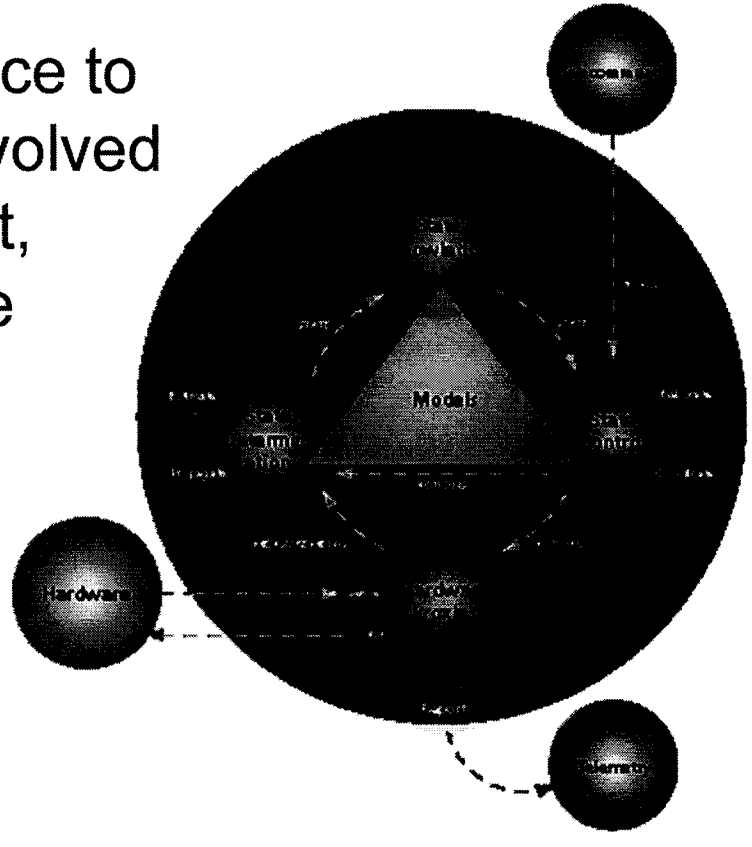
A unified architecture for flight, ground, and test systems that enables missions requiring reliable, advanced software

- Build a **highly reusable core software system** for a wide variety of space mission applications
- Promote modern, synergistic **processes for systems and software engineering**
- Establish an improved development life cycle for **more reliable mission software**
- **Reduce development cycle time and cost**
- **Reduce operations cost** with **increased autonomy**
  - Satisfy complex mission requirements (e.g., robust *in situ* exploration)
  - Express operational intent through goals

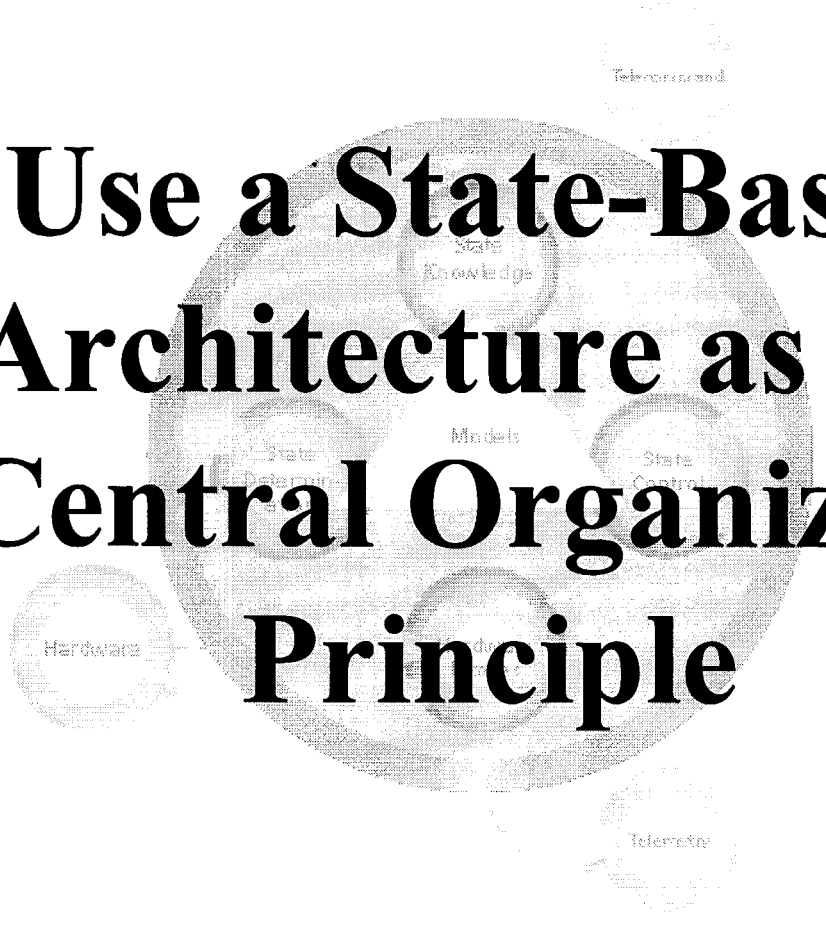
MDS Guidelines

- Based on the MDS experience to date, a set guidelines has evolved for developing a unified flight, ground, and test architecture



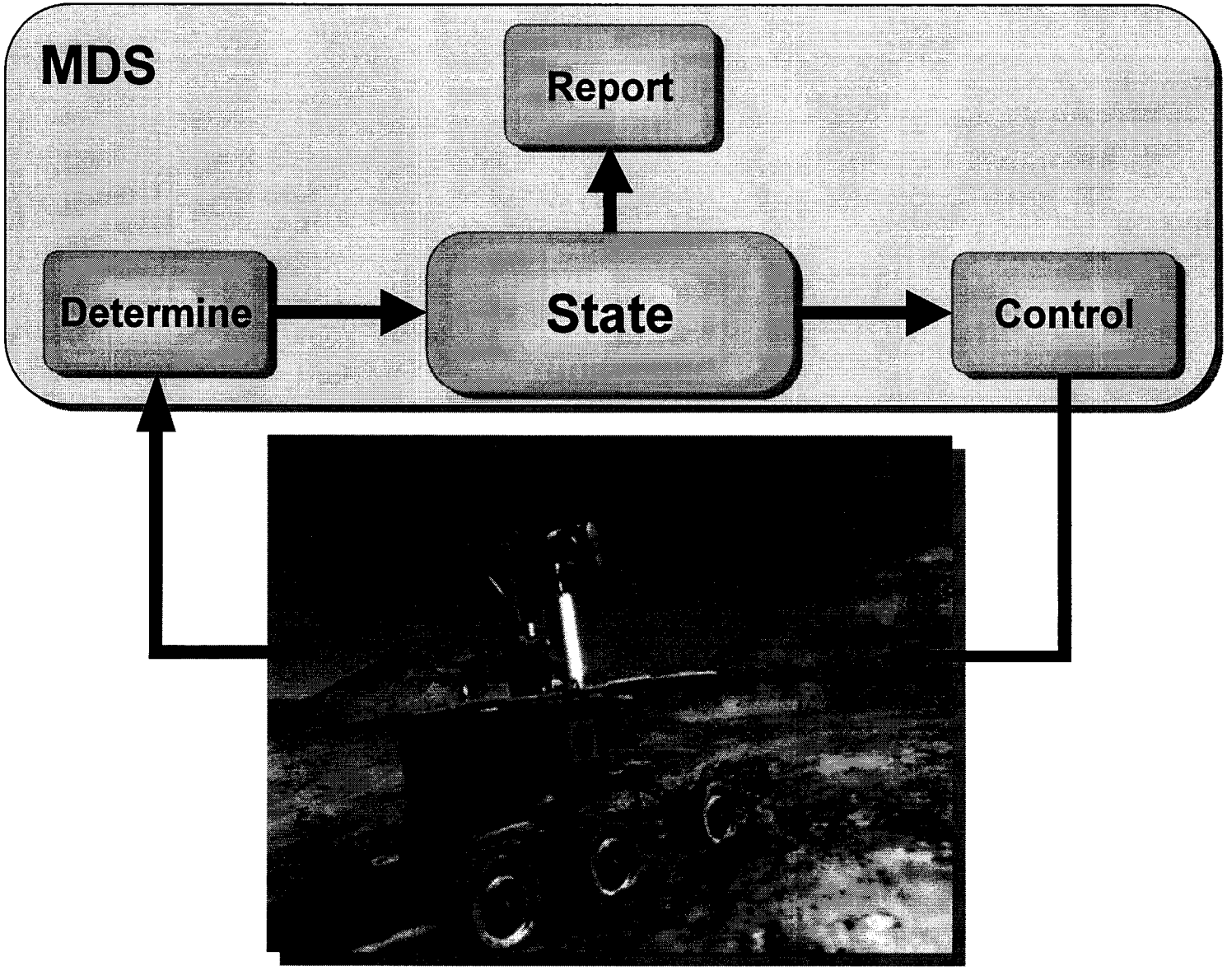


- These guidelines are derived from the direction defined and maintained by the MDS Architect (Robert Rasmussen)



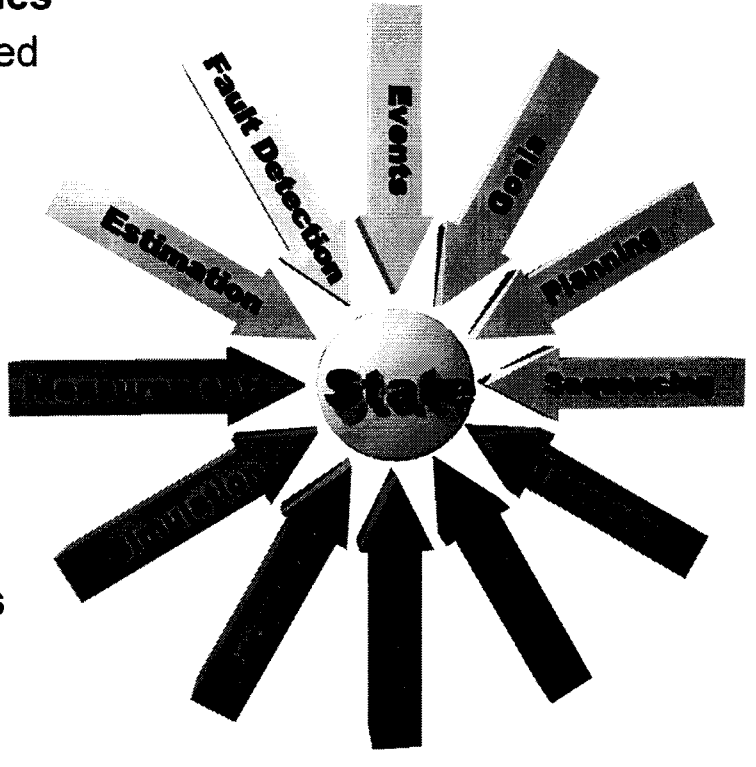
Use a State-Based Architecture as the Central Organizing Principle

State-Based Architecture

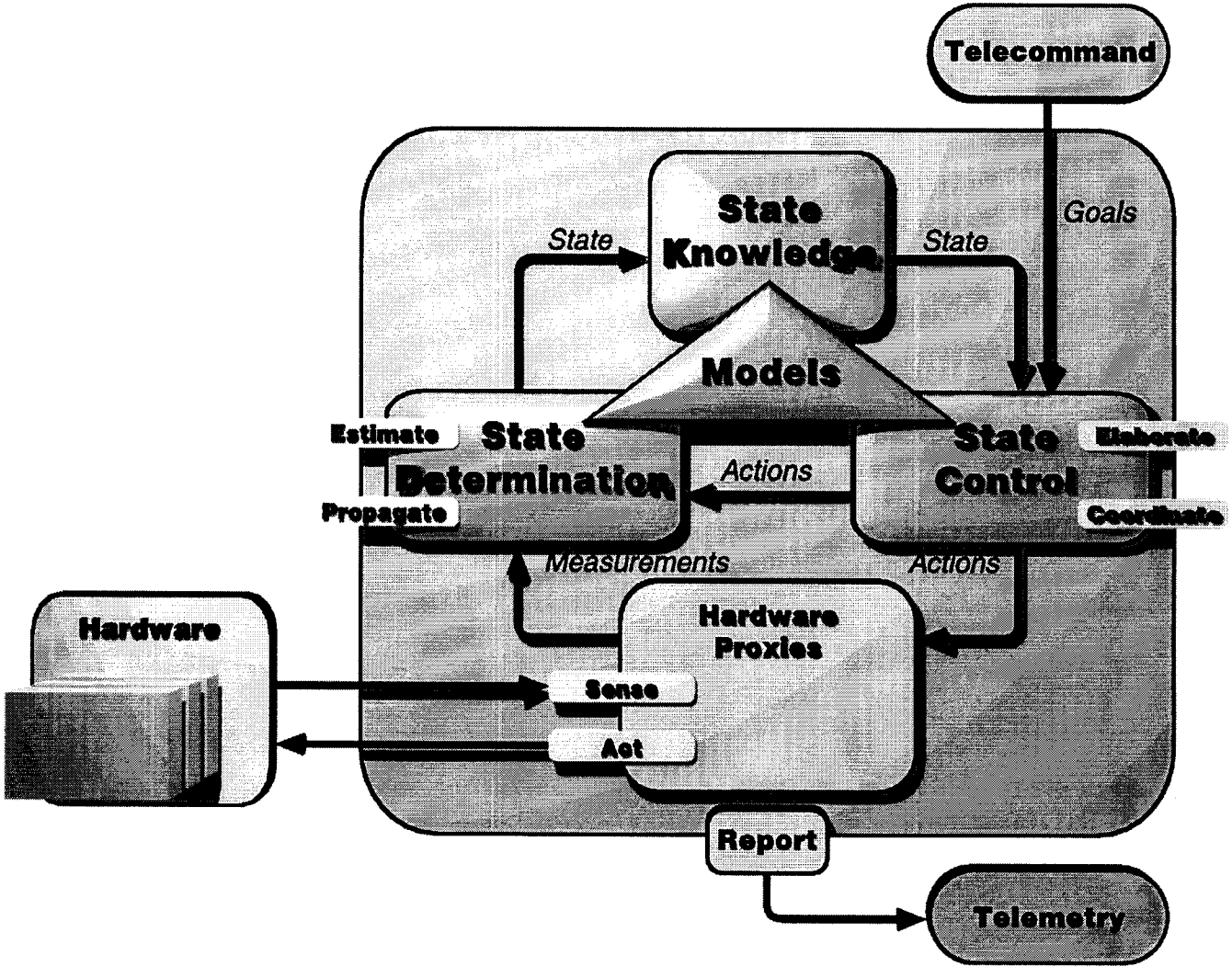


State is Central

- A **system** comprises project assets in the context of some external environment that influences them
- The function of mission software is to monitor and control a system to meet operators' intents
- MDS manages all essential aspects of this function via **state**
  - Knowledge of the system, including its environment, is represented over time in **state variables**
  - The behavior of the system is represented by **models** of this state
  - Interaction with the system is achieved via modeled relationships between state and interface data (**measurements** and **commands**), as mediated by **hardware proxies**
  - Information is reported, stored, and transported as **histories** of state, measurements, and commands
  - Operators' intent, including flight rules and constraints, are expressed as **goals** on system states

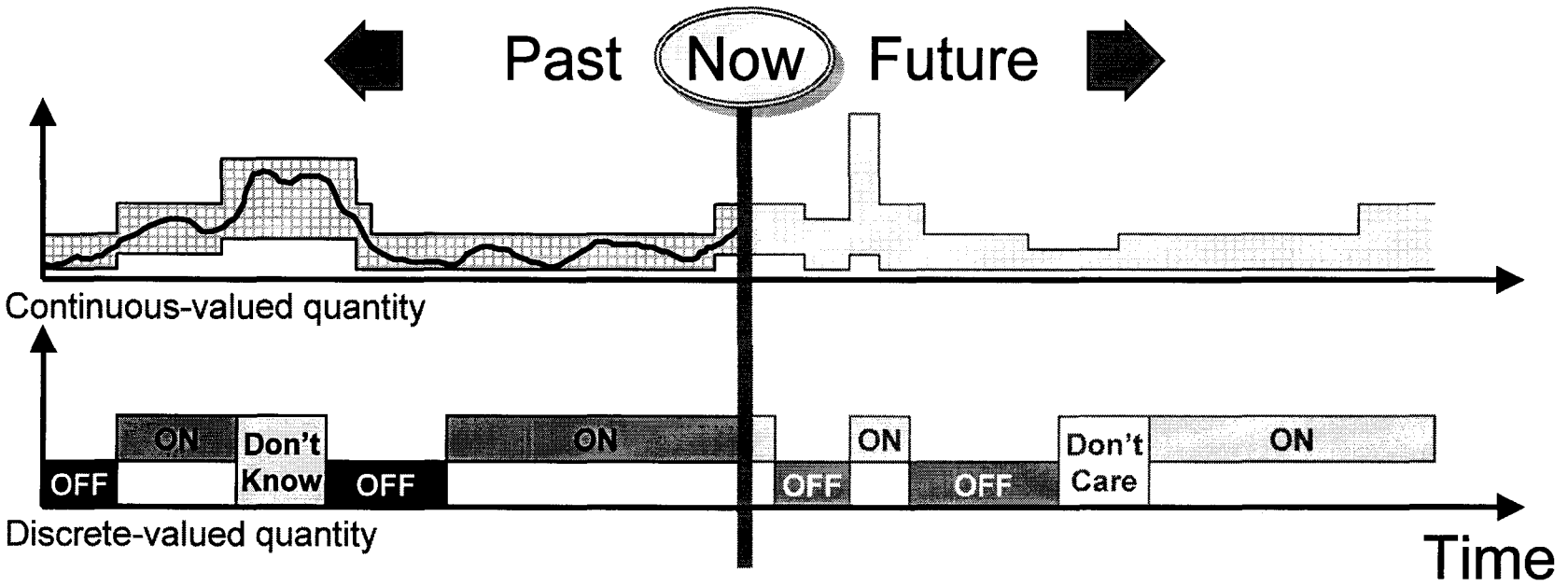


A High Level View



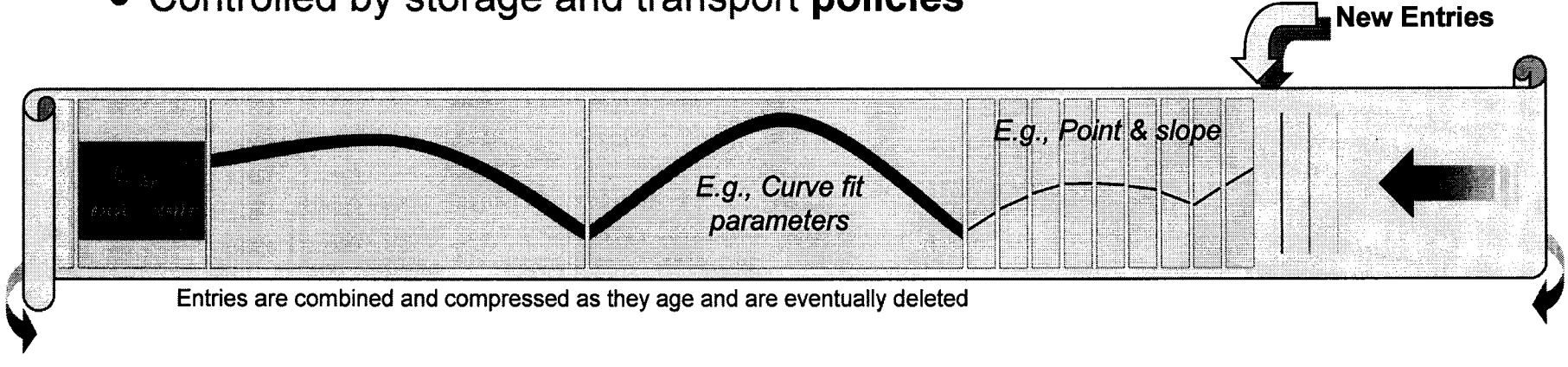
State Timelines

- **State timelines** maintain the value or set of possible values (e.g., a range) of a state variable as a function of time
- They capture both knowledge and intent about state



Value Histories

- A container mechanism supporting functions that produce values over time (state variable timelines, measurements, commands, …)
- Encapsulate the interface to **data management** persistent storage and **data transport**
  - Stored and transported as **data products**
  - Selected data products are preserved across resets
- Leverage the use of models to preserve continuous information using less storage space
- Can also simply store a set of discrete value instances
- Controlled by storage and transport **policies**



Use Goal-Directed Behavior to Express State Intent

State Intent

- Control is exercised over the system by imposing …
  - **Constraints on states**, which limit the range of a state variable
    - State is allowed flexibility within these bounds
  - **Constraints on time**, which limit the duration between two **time points**
    - Time points are variable points in time
    - These times are allowed flexibility, but again, with constraints

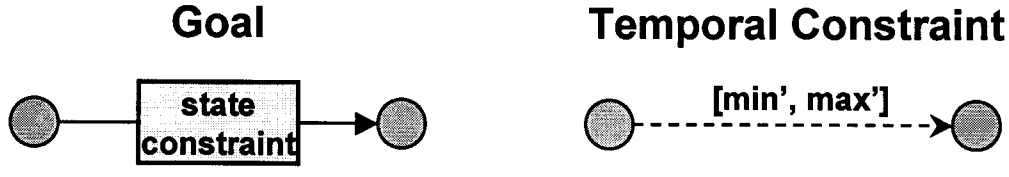
- A state constraint between two time points is called a **goal**
- A time constraint between two time points is called a **temporal constraint**

- Goals and temporal constraints are expressions of **intent**

- Success in constraint achievement is an objective matter
  - Criteria are explicitly expressed in constraint evaluation code
  - Directly verifiable during test, since constraints are explicitly evaluated

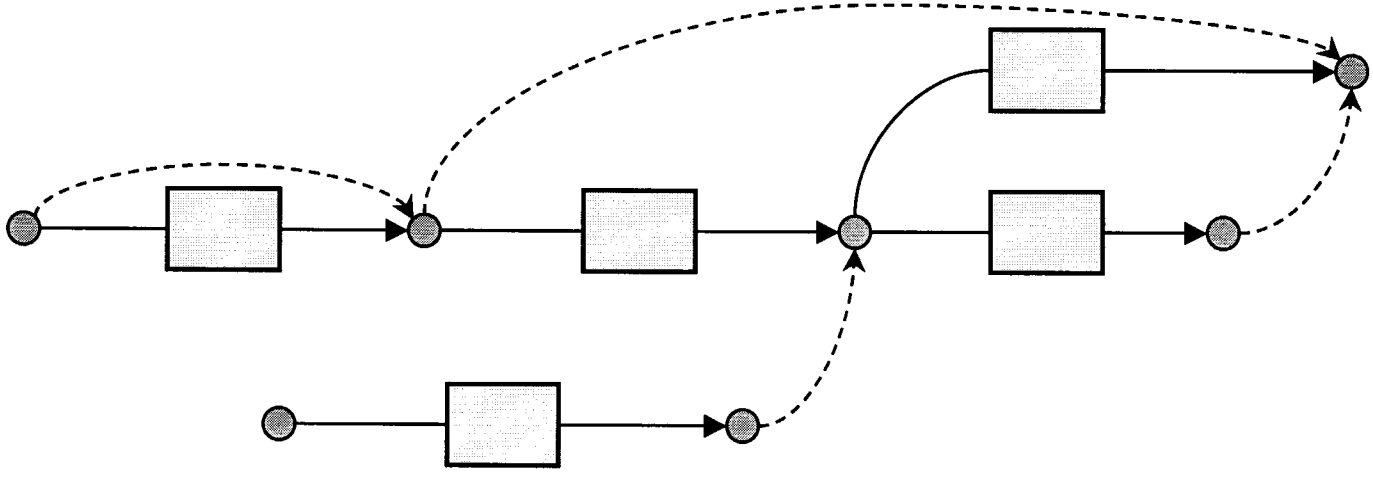
Constraint Networks

- Goals and temporal constraints each connect a pair of time points



- Time points are often shared (e.g., one beginning as another ends)

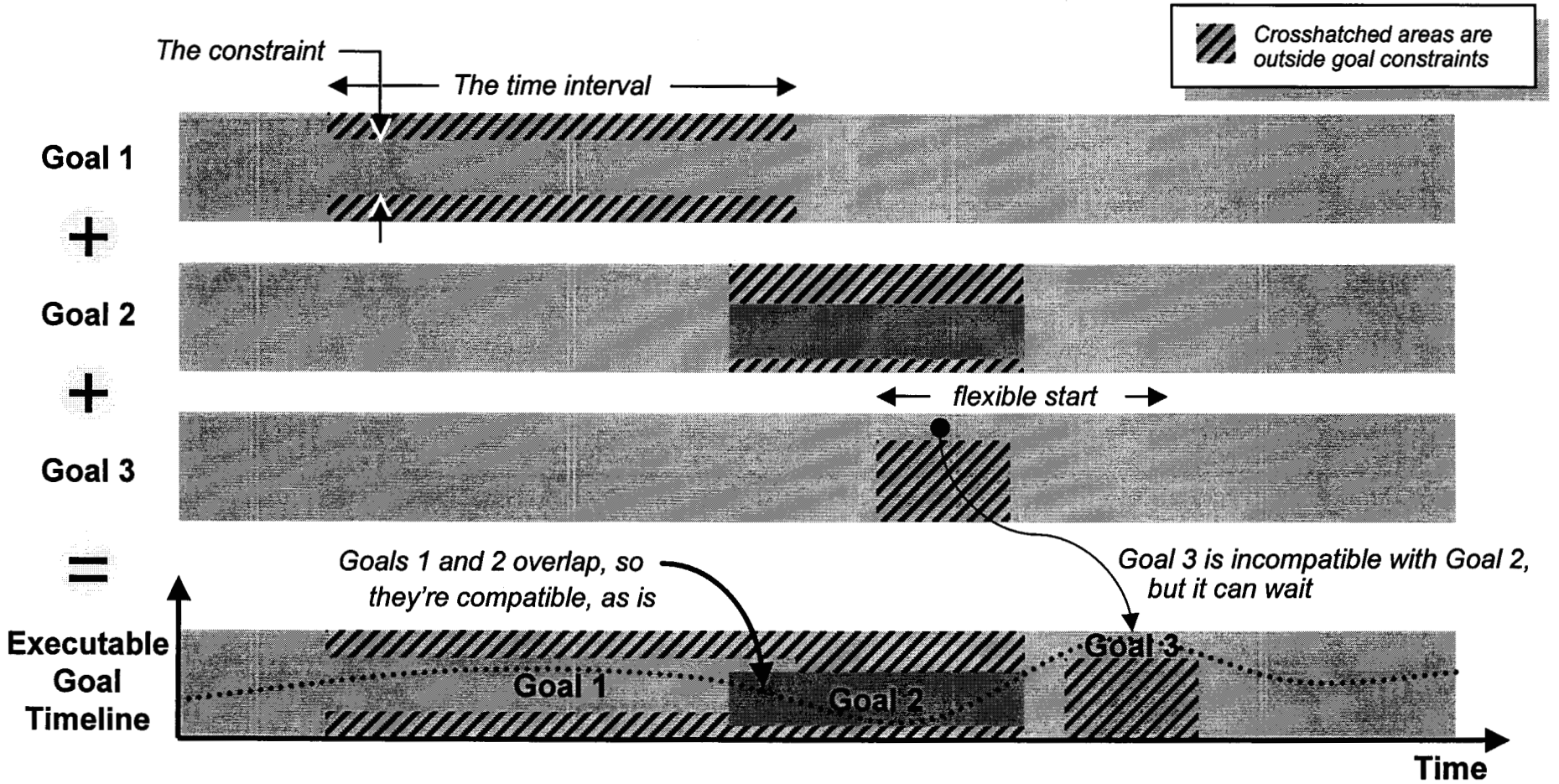
- A collection of connected goals and temporal constraints form a **constraint network**



Use Integrated Planning and Scheduling to Achieve Automomy

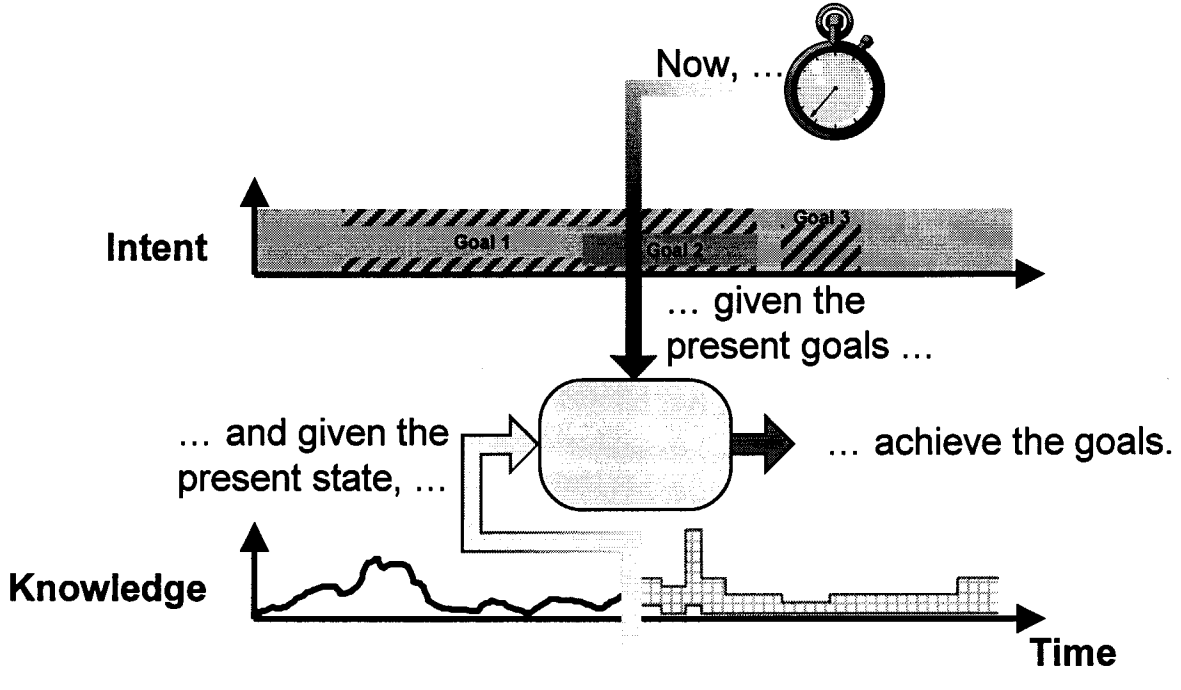
Resolving Conflicts

- Example: three goals on the same state



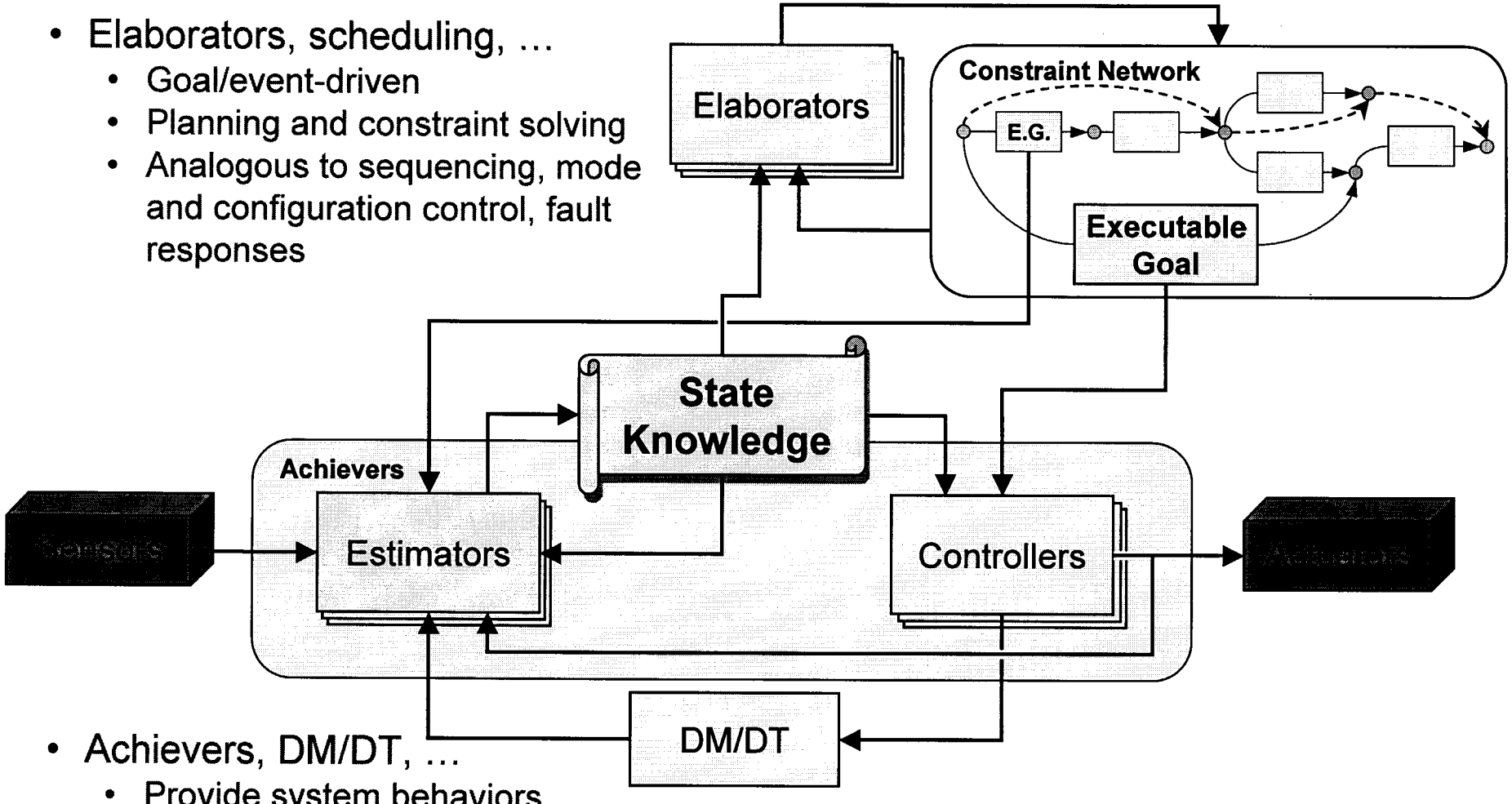
Timeline Execution

- Goals are accepted if successfully placed on the timeline for the goal state variable
- Goals are frozen and acted upon when they appear on the timeline in the immediate future
- Goals are acted upon by **achievers** assigned to each state variable
- Elaborators monitor execution and adapt plans, as necessary

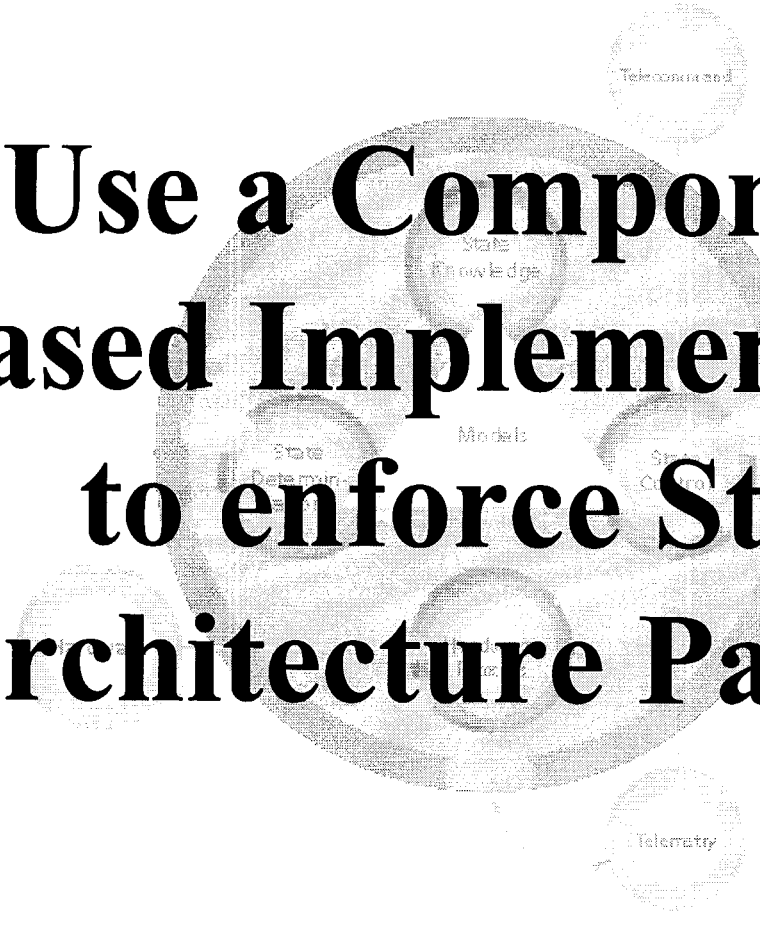


Putting It Together

- Elaborators, scheduling, …
  - Goal/event-driven
  - Planning and constraint solving
  - Analogous to sequencing, mode and configuration control, fault responses

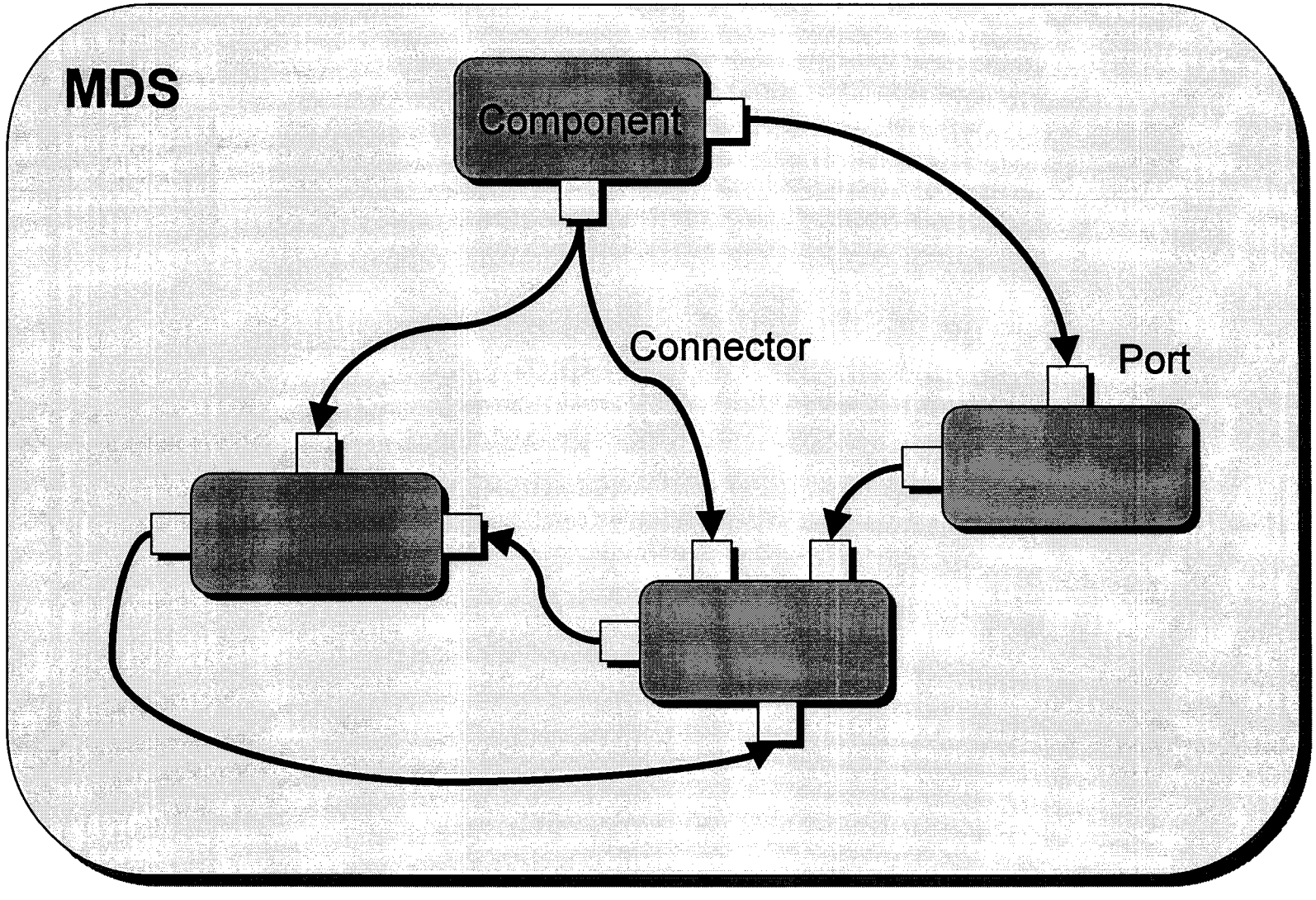


- Achievers, DM/DT, …
  - Provide system behaviors
  - Managed via goals and temporal constraints
  - Fairly conventional real-time monitoring and control processes



Use a Component-Based Implementation to enforce State Architecture Patterns

Component-Based Architecture



Components are Fundamental

- The Component Architecture establishes the elements of software design and their coherent integration
- Components and their connections embody…
  - The elements of functionality
  - Their types and registered instances within a deployment
  - Their interfaces and distribution across platforms
  - Their coordinated execution and synchronization
- These issues are raised to the level of symbolic realization
  - Software organization is established independently and systematically
  - It can be manipulated directly — including at run time, if necessary
  - Complexity becomes a manageable entity

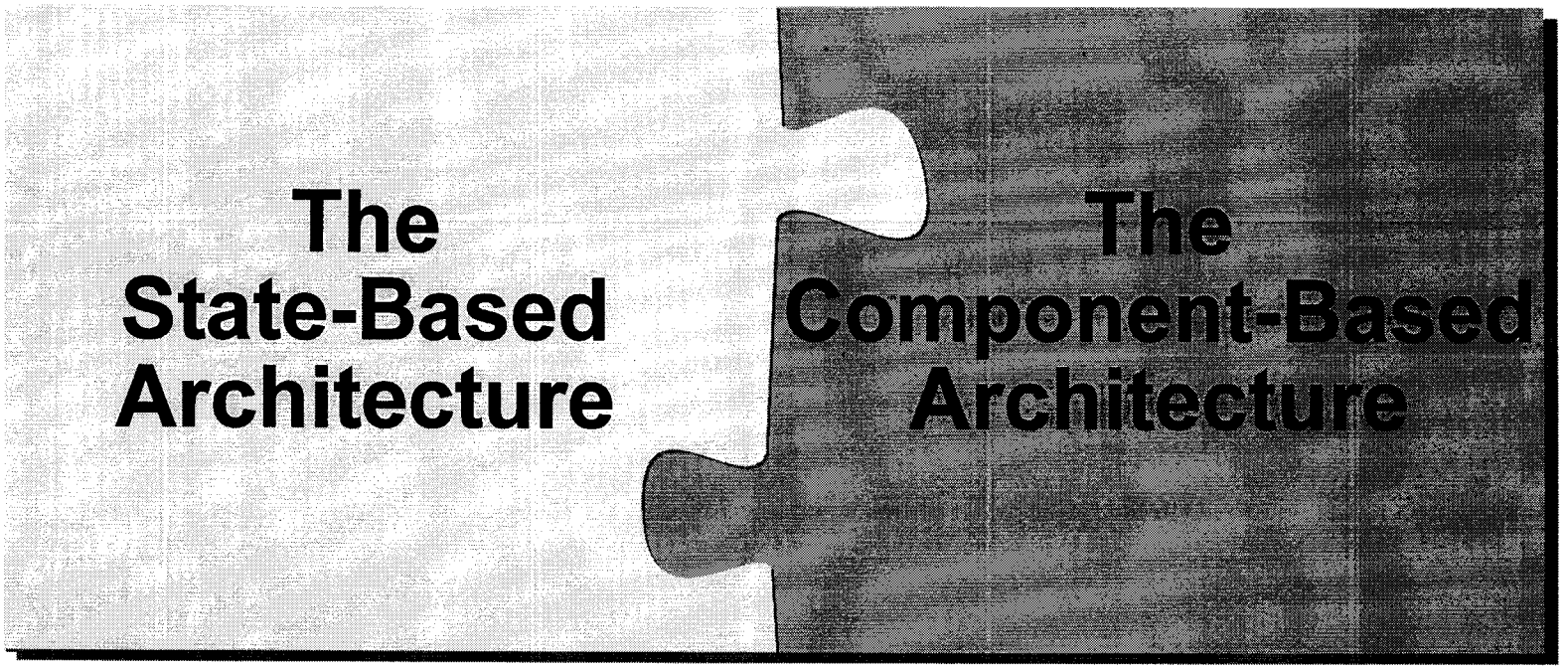
- The State Architecture establishes the elements of functionality and their functional relationships
  - E.g., state variables, achievers, hardware proxies, and so on
- It *does not* establish the software design

Connection Rules

- Functional elements of the State Architecture are structural elements in the Component Architecture
  - State variables, achievers, hardware proxies, and so on, are Components
- State Architecture elements all interrelate in a few formally established patterns
  - E.g., measurements are used only by estimators,
    goals are directed to state variables,
    only controllers issue commands,
    only estimators update state knowledge,
    and so on
  - These are rules on connections within the Component Architecture of the design
- The Component Architecture implements and enforces these patterns
  - Compliance is inspectable
  - Exceptions must be overtly managed — nothing is hidden

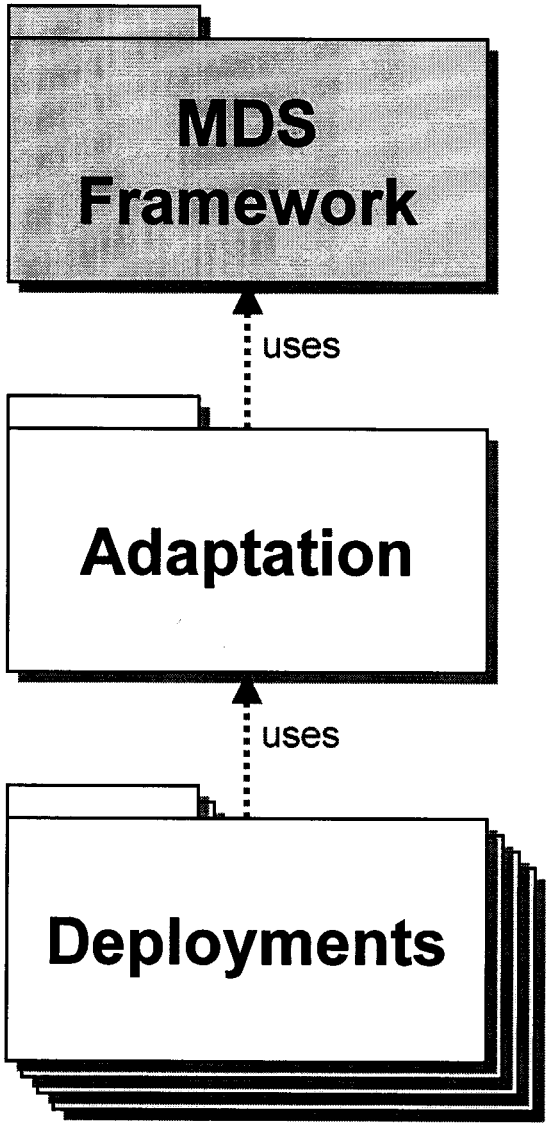
Use Incremental Development of a Software Framework to Establish the Foundation for Reusability

The Whole Picture
MDS Framework



- The State and Component Architectures are defined within a set of classes called the **MDS Framework**
  - Frameworks are the elements of a partially complete application
  - The MDS framework is organized in a hierarchy of dozens of packages
  - Each project **adapts** the framework by extending it in mission-specific ways

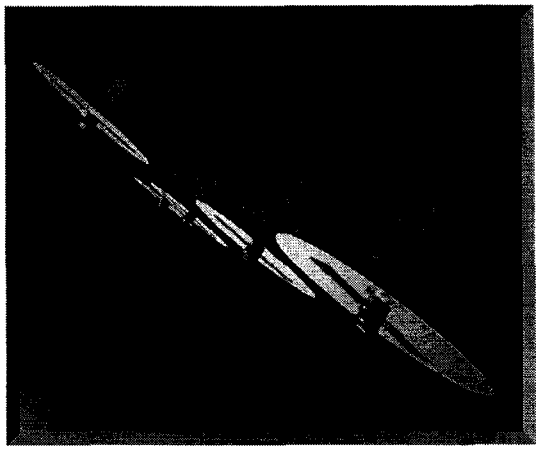
The MDS Common Model



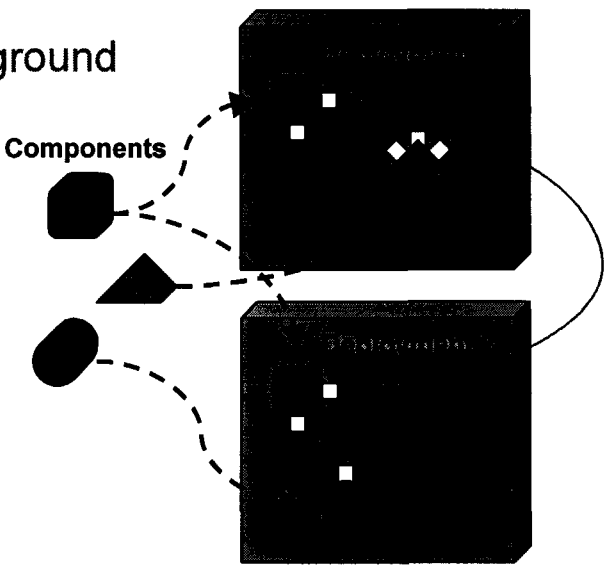
- The **MDS Framework** is the collection of most core classes within the MDS architecture
  - Developed and maintained exclusively by MDS
  - Uniform (except for versioning) across MDS adaptations
- Each project does an **Adaptation** of the framework
  - Captures project requirements and scenarios
  - Extends framework classes to address functions and configurations specific to the project
  - Reusable extensions are generalized (if necessary) and moved to the framework
- Several **Deployments** of the adaptation are defined
  - These are the executable configurations to be used in various settings (test beds, flight, ground, etc.)
- Framework is developed incrementally with example adaptations and deployments

Use Framework Adaptation to Create Deployments for Project-Level Reuse

Deployments



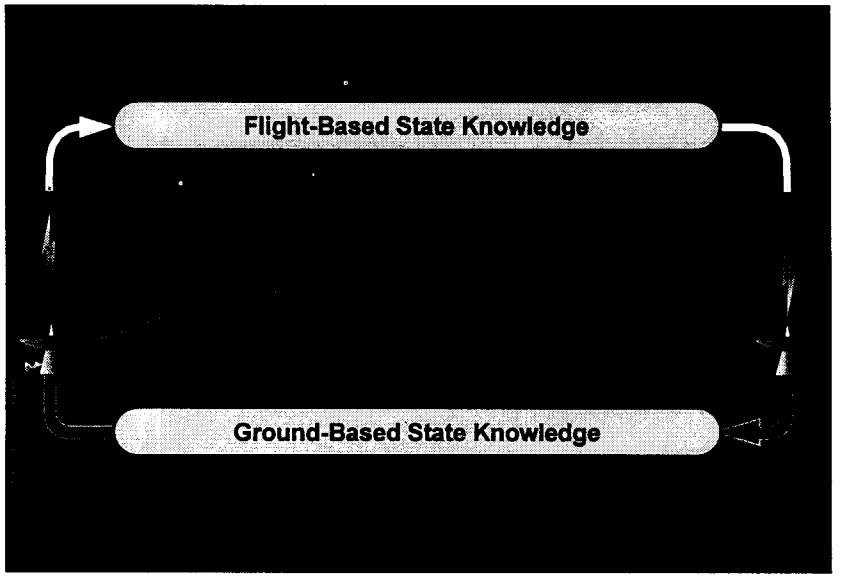
- A **deployment** is an executable product
  - Each project will have several deployments
  - E.g., the flight software, the simulation software during test, parts of the ground software, and so on
- Each deployment is constructed from components, connected as appropriate for that application
  - Not every component belongs in every deployment
  - E.g., attitude is usually estimated only on board, while trajectory is usually estimated only on the ground
- Deployments may be interconnected
- For remote links, deployments communicate via component proxies
  - Exchanges between a component and its proxy are managed by data transport services



For example...

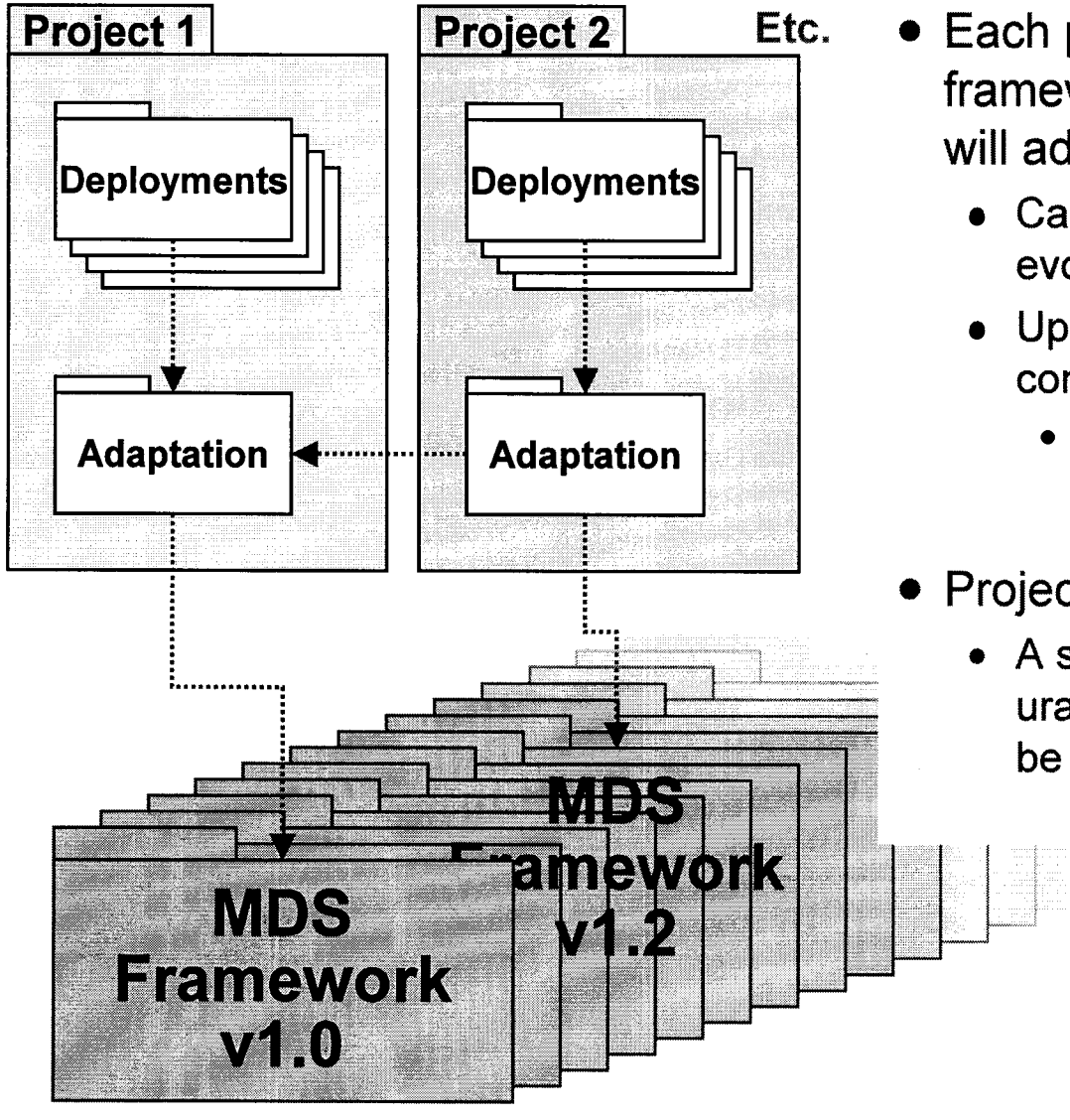
Ground⇄Flight Knowledge Exchange

- State knowledge is needed in both places
  - Common representation
  - Coordinated, consolidated & maintained, as appropriate
- Information is exchanged via state variable proxies
  - Original source in one deployment
  - Copied (at some level) to a proxy in the other

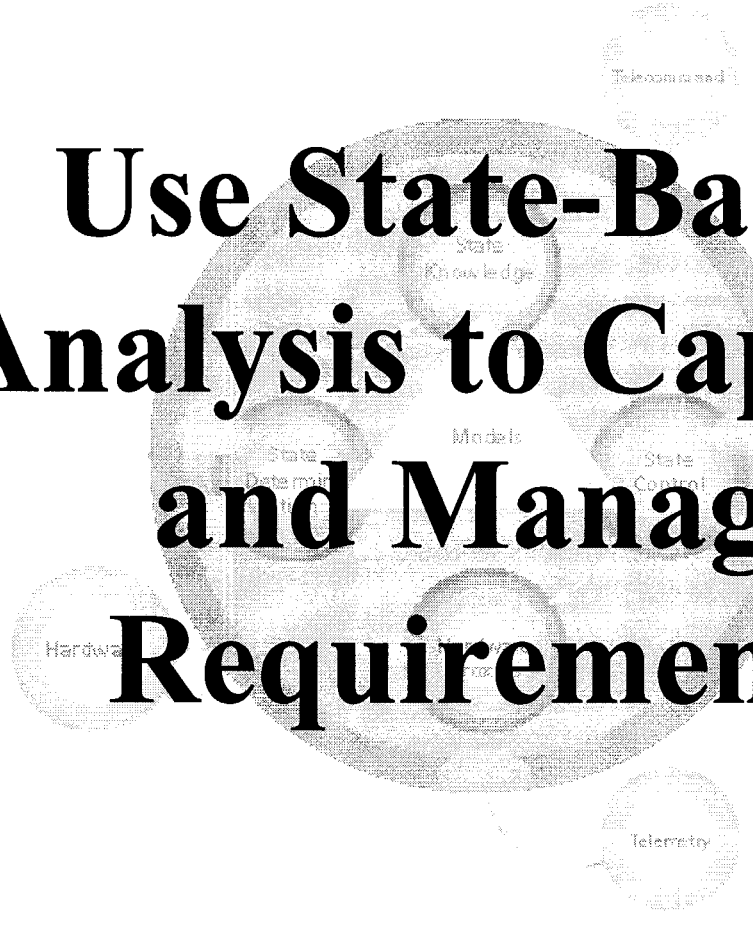


- Ground-based state determination is...
  - Typically for things like orbit determination, calibration, ...
  - Up-linked as necessary (trajectories, parameters, ...)
- Flight-based state determination is...
  - Typically for things like attitude determination, device states, faults, ...
  - Down-linked as available (part of telemetry)

Reuse Among Projects



- Each project uses the same framework, except that later projects will adapt later versions
  - Can continue to track framework evolution up to some freeze point
  - Updates to frozen version are confined to that project
    - Though mainline framework development may decide to make some of the same updates
- Projects can adapt from one another
  - A similar track-then-freeze configuration management process would be necessary
    - Example adaptations currently under way are:
      - Mars Rover Mobility Adaptation
      - Mars Lander Powered Terminal Descent Adaptation



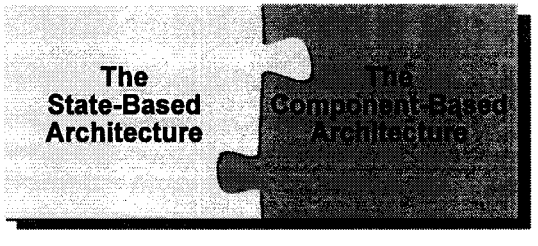
Use State-Based Analysis to Capture and Manage Requirements

Systems Engineering

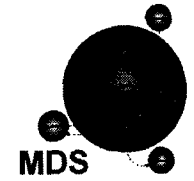
- Systems and software engineering need to complement one another
  - Systems engineering must define the system and behavior
  - Software must understand the system and guide its behavior

- **State Analysis** is a model–based process defined by MDS to aid systems and software engineering
  - State analysis prompts comparatively methodical and rigorous analyses of systems
  - MDS permits the uniform expression of systems engineering concepts in software architectural terms
  - Due to the alignment of State and Component architectures, both functionality and software design are considered simultaneously
  - Resulting products map directly onto the MDS architectural elements
  - Most MDS adaptation requirements can be defined by state analysis

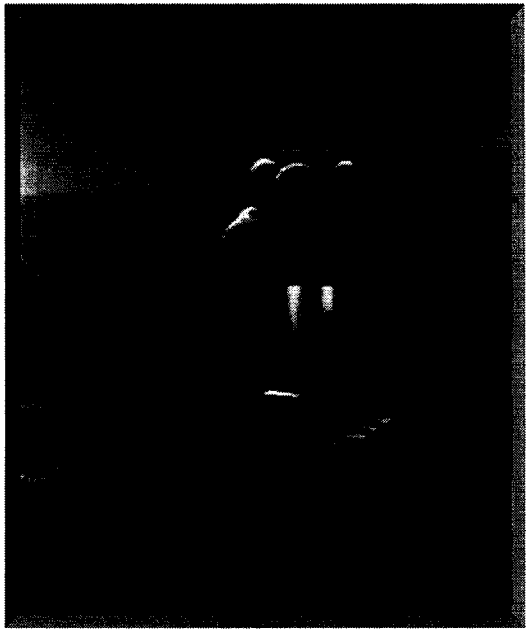
- State and Component architecture specifications are supported by tools, which will ultimately evolve into a unified code generation system for MDS



MDS Guideline Summary



- 1. Use a State-Based Architecture as the Central Organizing Principle
- 2. Use Goal-Directed Behavior to Express Intent
- 3. Use Integrated Planning and Scheduling to Achieve Autonomy
- 4. Use a Component-Based Implementation to enforce State Architecture Patterns
- 5. Use Incremental Development of a Software Framework to Establish the Foundation for Reusability
- 6. Use Framework Adaptation to Create Deployments for Project-Level Reuse
- 7. Use State-Based Analysis to Capture and Manage Requirements

Conclusion

- MDS provides a revolutionary integration of systems and software engineering which addresses…
  - Architectures for both functional and software design interactions
  - Unification of flight, ground, and test elements
  - Reuse across deployments and projects
  - A wide range of technical issues including autonomy
  - Processes, tools, and design ready for the challenge of a flight program
- State and Component Architectures are the bedrock of our approach
  - Each exploits a relatively small but powerful set of ideas
  - The two architectures complement one another in a natural but far-reaching manner