Mission Data System

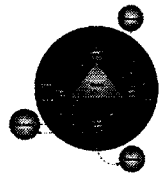
An end-to-end platform for high dependability autonomous monitoring and control of physical systems

HDCP Workshop
May 29–30, 2002

DRAFT

Dan Dvorak, Kenny Meyer, Kirk Reinholtz
Jet Propulsion Laboratory
California Institute of Technology

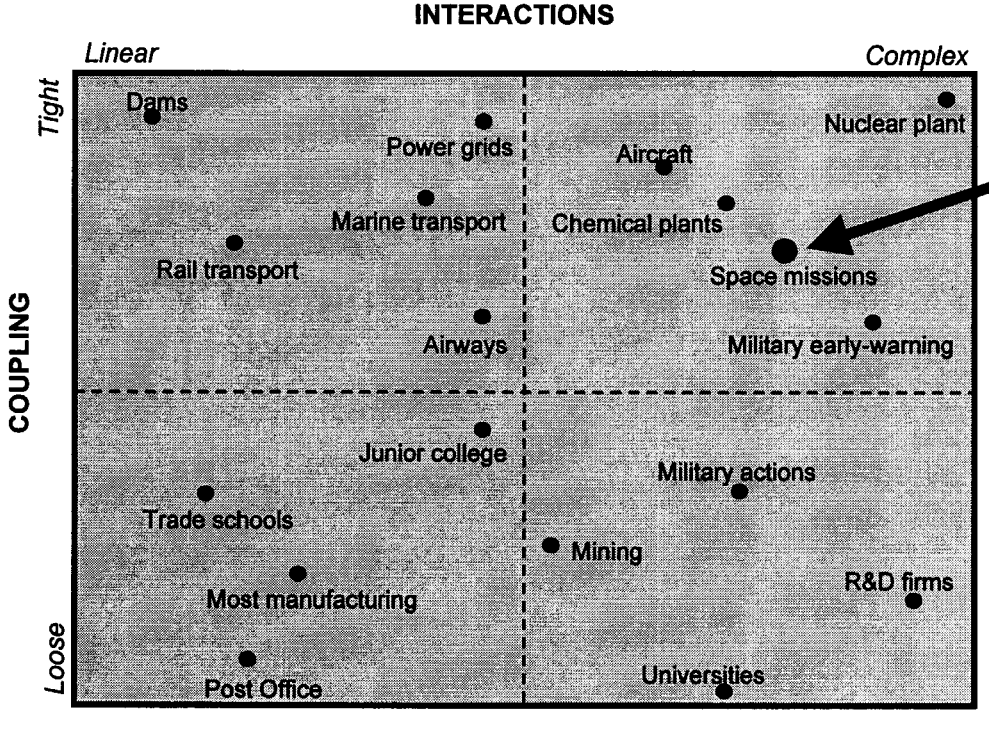
Preface



- MDS addresses autonomous monitoring and control of physical systems
- MDS has a broad scope:
  - A systems engineering process
  - A matching software architecture of two dimensions:
    - A state- and goal-based architecture for monitoring and control
    - A component & connection architecture for software engineering
  - Core framework software
  - Iterative development process
- Opportunities for improving dependability exist in all 4 areas

Origins of MDS

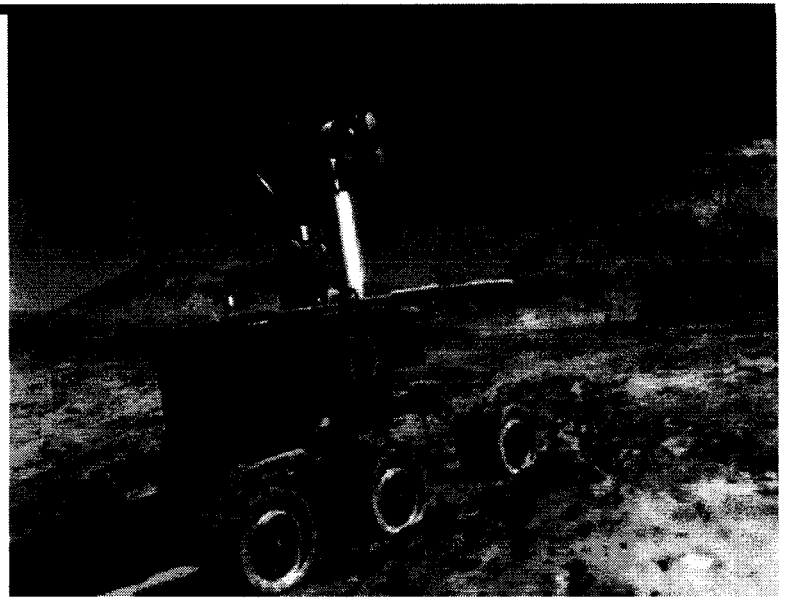
- MDS conceived as a unified flight / ground / test architecture for unmanned space science missions
- Systems are high-risk due to complex interactions and tight coupling; space is an unforgiving environment



You are here.

From *Normal Accidents: Living with High-Risk Technologies*, by Charles Perrow, 1984.

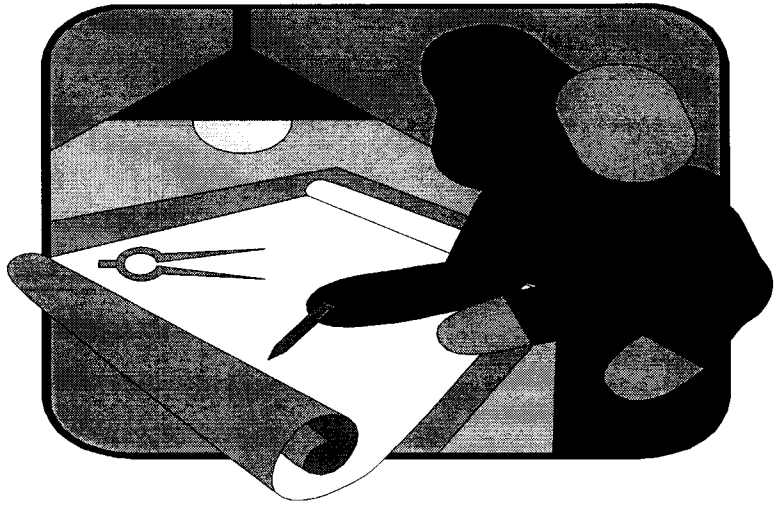
MDS Problem Domain



- Autonomous monitoring and control of physical systems
  - Mobile robots (spacecraft, planetary rovers, lawn mowers, ...)
  - Immobile robots (manufacturing robots, industrial process control, toasters, ...)
  - Vast sensor/actuator web: "The Edge" of the world-wide web
- Scope includes:
  - Real-time control and estimation
  - Deliberative planning and scheduling of goals
  - Management of engineering and science data
  - Infrequent data transport across links having huge round-trip delays
  - Modeling of complex interactions
  - Reactive control, including fault protection, at multiple time scales
  - Management of many limited resources (power, memory, pointing, ...)
  - Human operation/monitoring of autonomous agents

MDS Approach & Scope

- Apply *product line practice* to:
  - Exploit commonalities across products
  - Reduce cycle time and costs
  - Improve reliability



- Scope includes:
  - Systems engineering analysis & design process
    - Collaborative, iterative analysis & design & refinement captured in a structured form
  - State architecture
    - States, goals, models, estimators, controllers, measurements, commands, ...
  - Component architecture
    - Components, interfaces, ports, connections, roles, configuration, rules, ...
  - Core framework software
    - State variables, goal networks, time services, data catalog, logging, naming, units, ...
  - Iterative build & test process
    - Work packages, package promotion, daily builds, baselines, test cases, ...

MDS Products

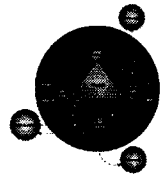
MDS customers will get...

1. **Unified flight, ground and test architecture**
   - Based on several broad organizing themes
2. **Orderly systems engineering methodology**
   - Bridging the gaps from analysis to requirements to software design to operations
3. **Object-oriented design**
   - Comprised of classes, templates, patterns, test scenarios, and so on, described in the Unified Modeling Language
4. **Frameworks**
   - Core infrastructure *and* discipline-specific sets of classes, algorithms, and models adaptable to a variety of missions
5. **Processes, tools, and documentation**
   - Integrated development, test and simulation environments and guidelines, monitoring, diagnosis and verification tools, adapters guides, and so on
6. **Examples**
   - Executable instantiations of frameworks that will serve as examples for real applications
7. **Reusable software**
   - An expanding legacy of code easily applied to successive missions

Architectural Themes

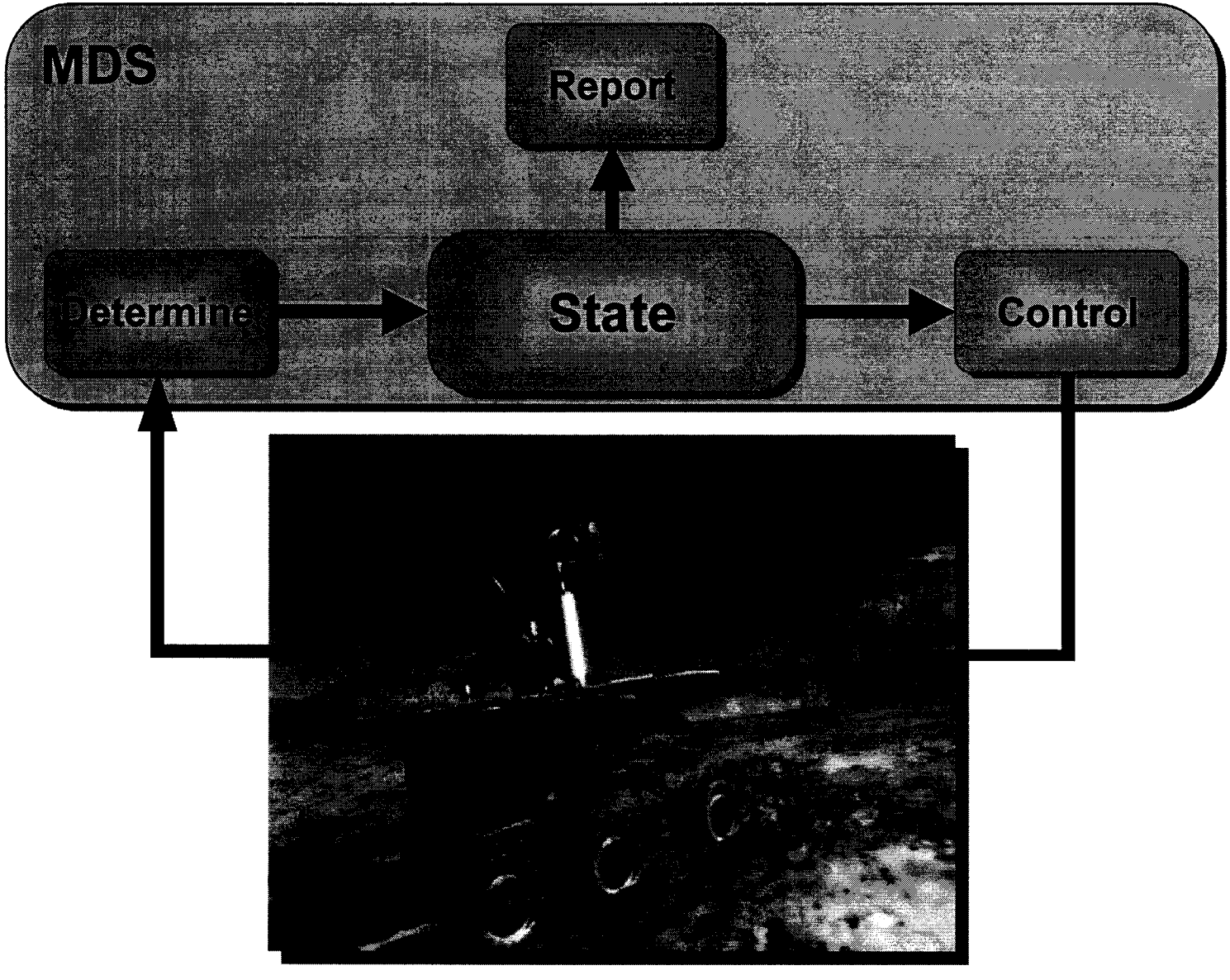
- Formal representation and manipulation of state are central
- State uncertainty is acknowledged and used in decision making
- Goal-based operation specifies intent; simplifies operations
- Express domain knowledge in models, not program logic
- Closed-loop control for in situ reaction to events
- Fault protection is integral part of design, not an add-on
- Real-time resource management (for power, fuel, etc.)
- Clean separation of state determination from control
- Clean separation of data management from data transport
- Navigation and attitude control built from a common base
- Plan to migrate capability from ground to flight
- Make interactions visible/analyzable at an architectural level

"Architecture Hoisting"



- Things that are "buried in the analysis" or "buried in the code" are hard to review, analyze, modify, and manage
- In many areas MDS makes things *explicit* and brings them out into the light of day. We call this "architecture hoisting":
  - Components communicate only via connections
  - Architecture configuration is explicit and inspectable
  - Units of measurement are explicit and enforced
  - Initialization dependencies are explicit and enforced
  - State timelines make estimated and planned states visible to all
  - Operation based on explicit, unambiguous constraints on state & time
  - Synchronization at component boundary is explicit and inspectable
  - Physical interactions are explicitly represented in models
  - Resources (power, etc) are managed as state allocations
- All this explicitness offers many opportunities for improving dependability through analysis and verification

State-Based Architecture



Bridging the Gap Between Systems and Software Engineering

Systems engineers follow a disciplined "state analysis", asking & answering questions such as these:

The state architecture bridges the gap through a shared set of architectural elements

Software engineers build the system by adapting a software framework having the same architectural elements

What do you want to achieve?
Move rover to rock. → ***Goal*** → Goal Network, Simulation, …

What's the state to be controlled?
Rover position relative to rock. → ***State Variable*** → State Knowledge, Hardware Proxy, Sequential Estimation

How do you know what that is?
Measure relative position with stereo camera. → ***Measurement*** → Measurement Package, Data Catalog, Graph Library

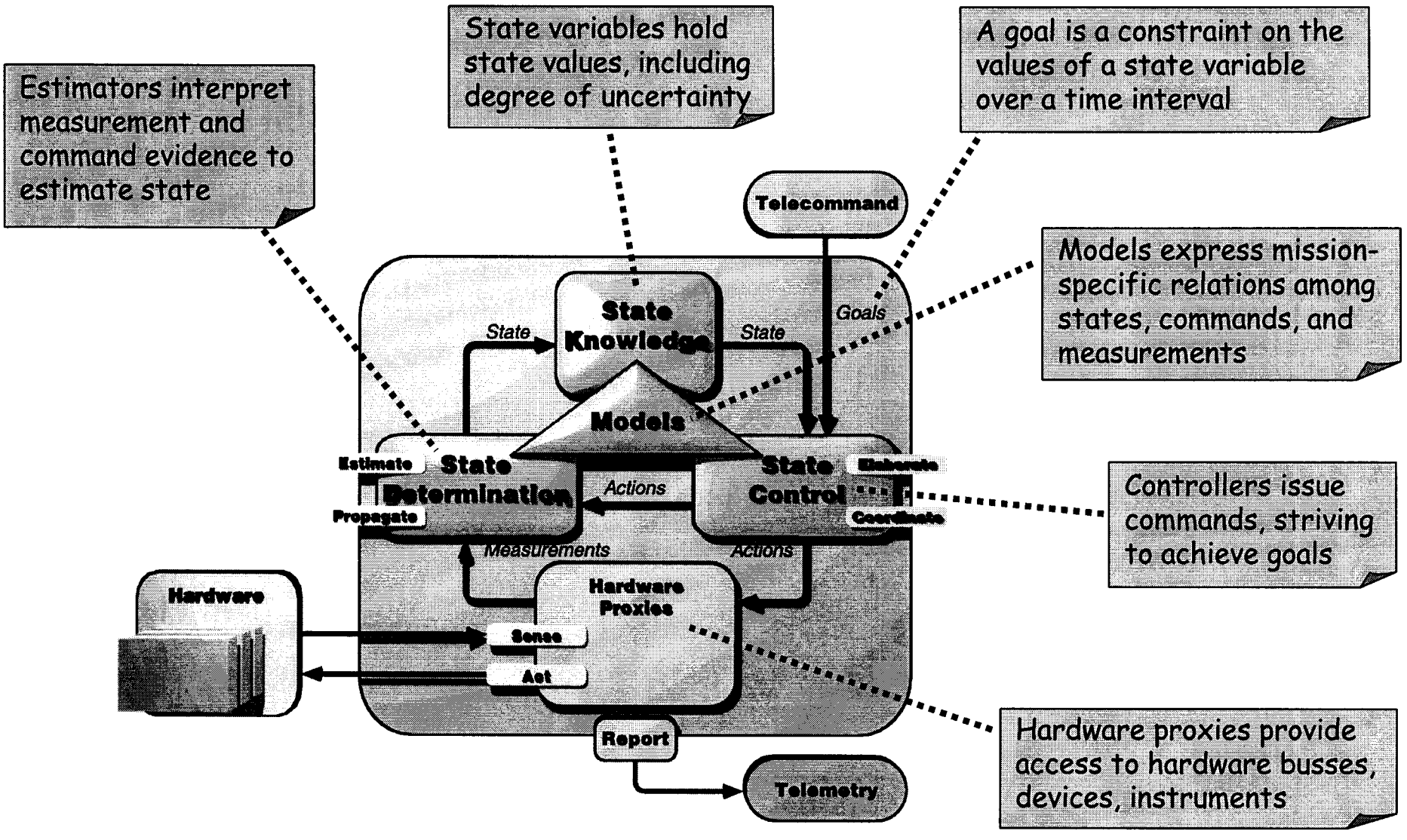
What does the stereo camera measure?
Distance to terrain features, light level, camera power (on/off), camera health. → ***Measurement Model*** → State Models, Coordinate Systems, SI Units, Naming

How do you control light level?
Wait until the sun is up. → ***State Model*** → Time Services, Linear Algebra, Event Logging

Where is sun relative to horizon?
…

Etc. → …, …, …

State-Based Architecture

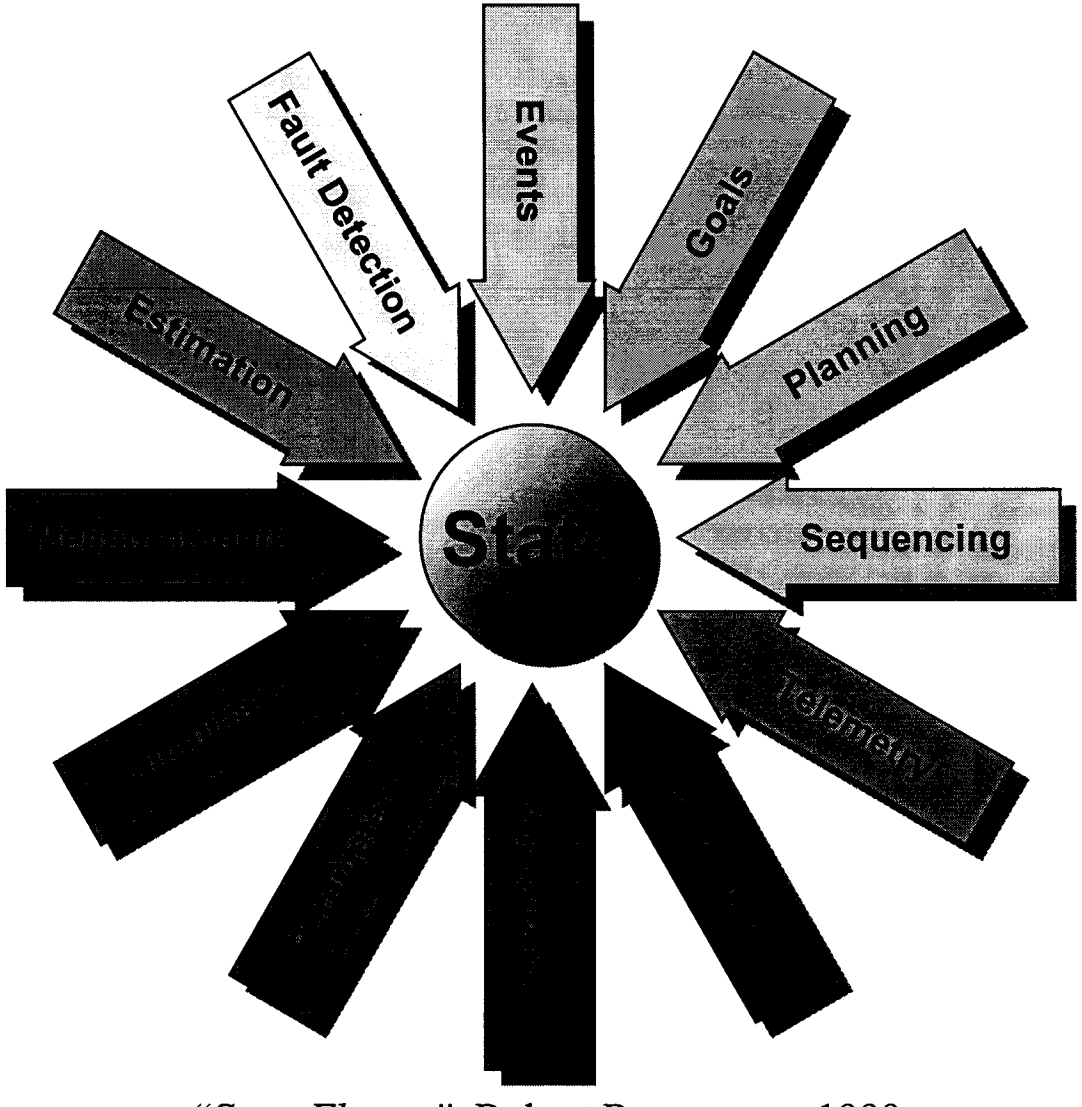


State is Central

All application-level functions of the MDS architecture involve the concept of state

- The **state** of a system at some time is a snapshot of all the "interesting" changeable features of the system
- A system **model** describes all the "interesting" features of how a system changes
- The **system** includes the spacecraft, its environment, and other actors in the environment (e.g. ground stations) with which it interacts in an "interesting" way

The essence of the MDS state-based architecture is to place the explicit management of state at the center of all system activities

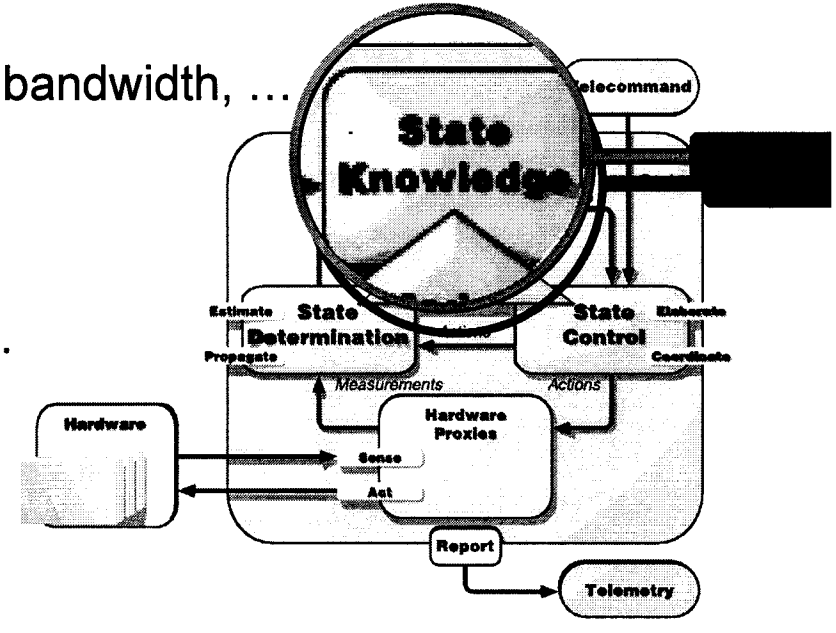


"State Flower", Robert Rasmussen, 1999.

State Knowledge
Everything You Need to Know

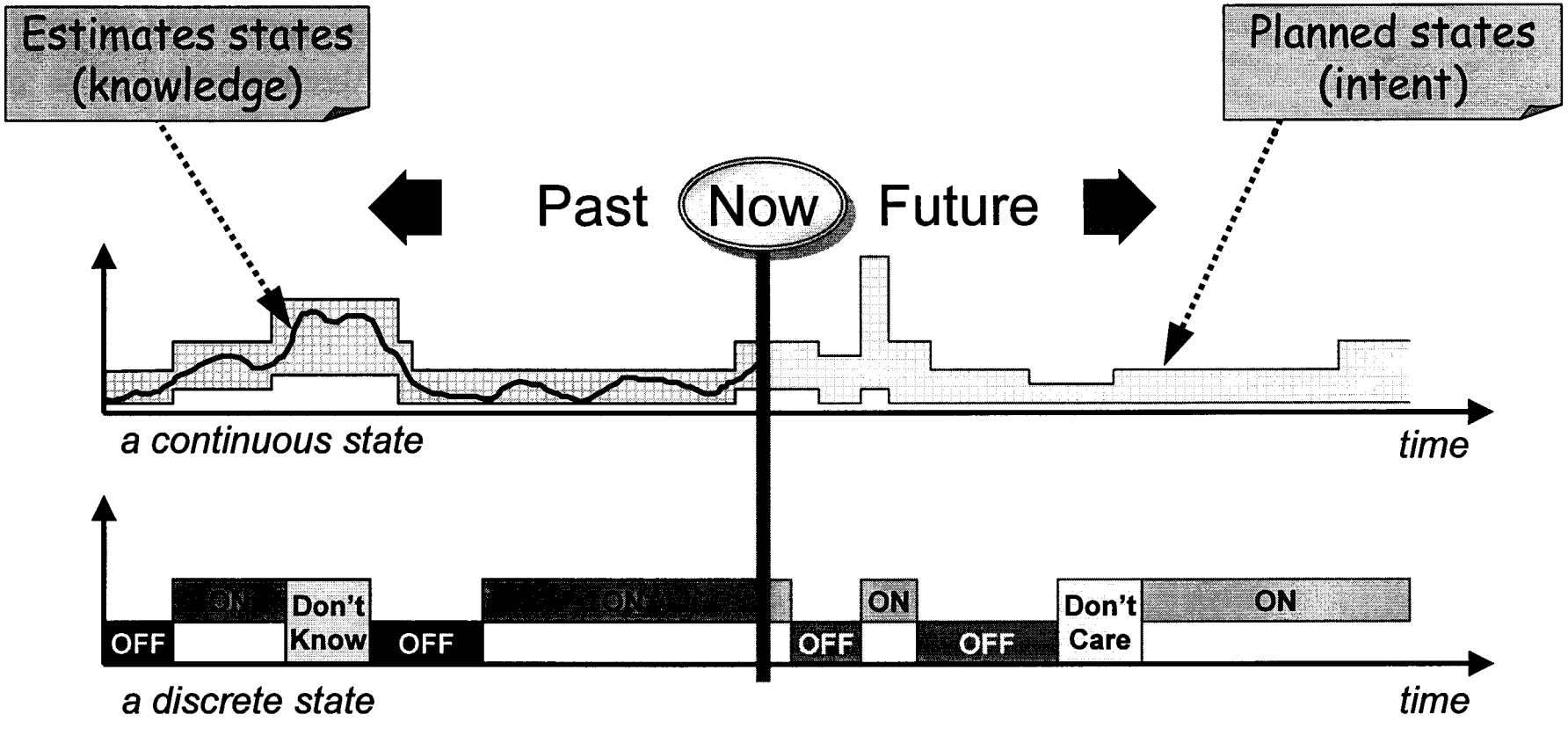
- Dynamics
  - Vehicle position & attitude, gimbal angles, wheel rotation, …
- Environment
  - Ephemeris, light level, atmospheric profiles, terrain, …
- Device status
  - Configuration, temperature, operating modes, failure modes, …
- Parameters
  - Mass properties, scale factors, biases, alignments,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

… and so on



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

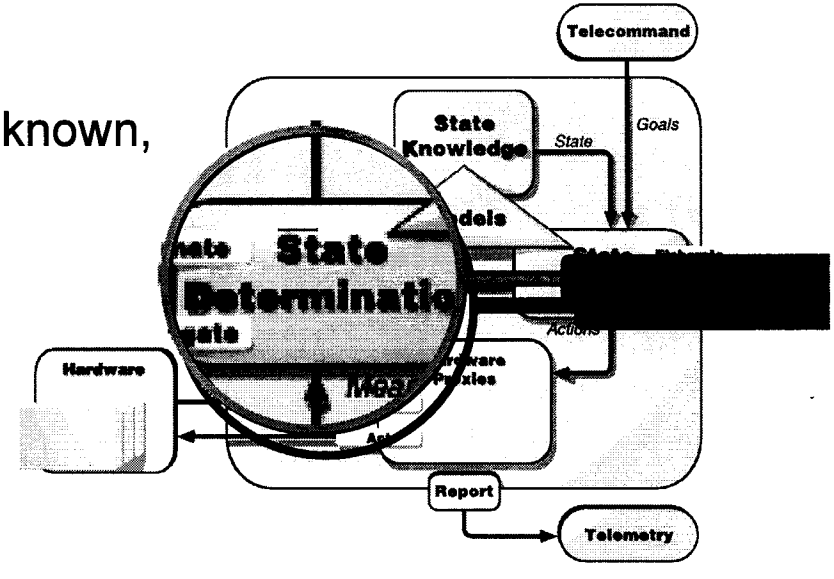


State Determination

Making Sense of the World

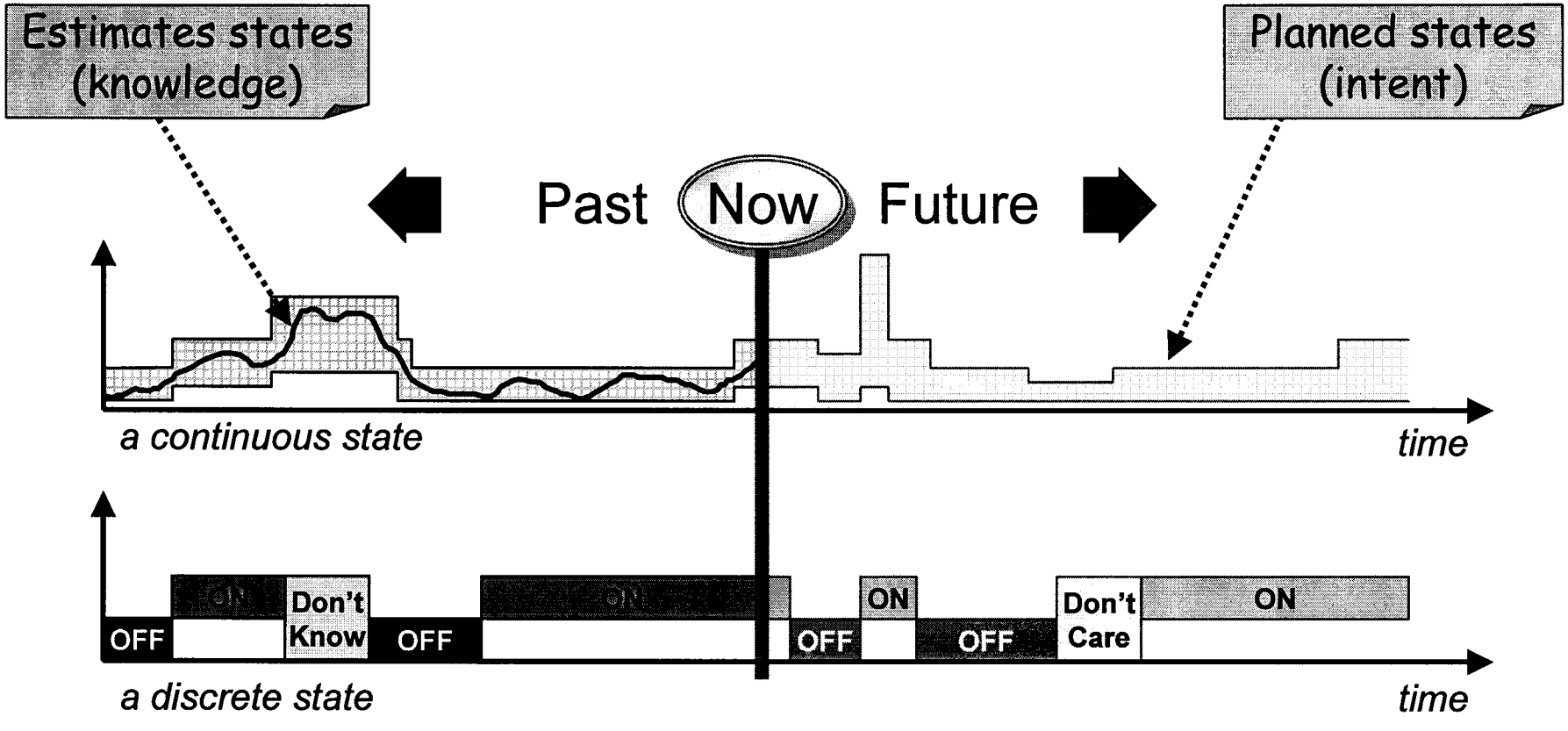
- One can act only on one's **knowledge** of the system
  - Knowledge is **what** you know, **not how** you know it
  - Observations (e.g., measurements) are *not* knowledge
- **Estimators** find "good" explanations for observations and other evidence, given a **model** of how things work
  - Knowledge may be **propagated** into the future, given models and plans

- All knowledge is **uncertain**
  - Judgment must be based both on what is known, and on how well it is known
- However, one *can* **achieve** local consistency of knowledge



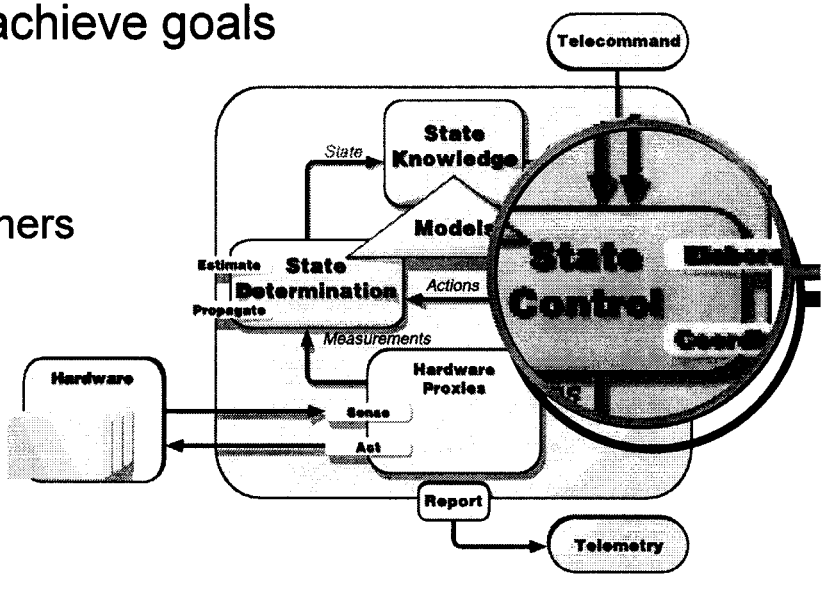
State Timelines

- **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



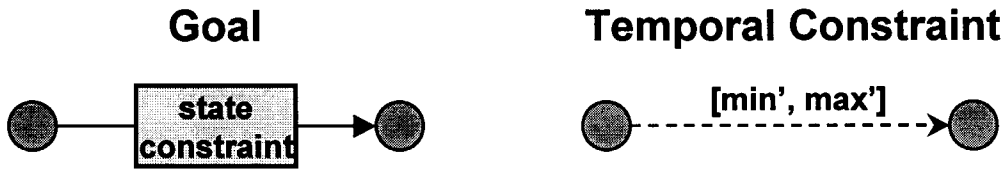
State Control
Closing the Loop

- Operators express their intent in the form of **goals**
  - Goals declare **what** should happen, **not how**
  - Goals specify constraints on state and constraints on time
- High level goals are elaborated recursively into lower level goals
  - **Elaboration** may be conditional, in order to react to present circumstances
  - **Coordination** of activities is accomplished by **scheduling**
  - Conflicts are resolved, with priority as final arbiter
- Knowledge of all states is maintained, as required to achieve goals
  - Knowledge is compared to goal constraints to test for compliance
- Corrective action is applied, as required to achieve goals
  - Alternate methods of **achievement** may be applied at any level
  - Unachievable goals (and their elaborations) are dropped individually without sacrificing others
- **Supports fault tolerance, critical activities, *in situ* autonomy, opportunistic science, and more**

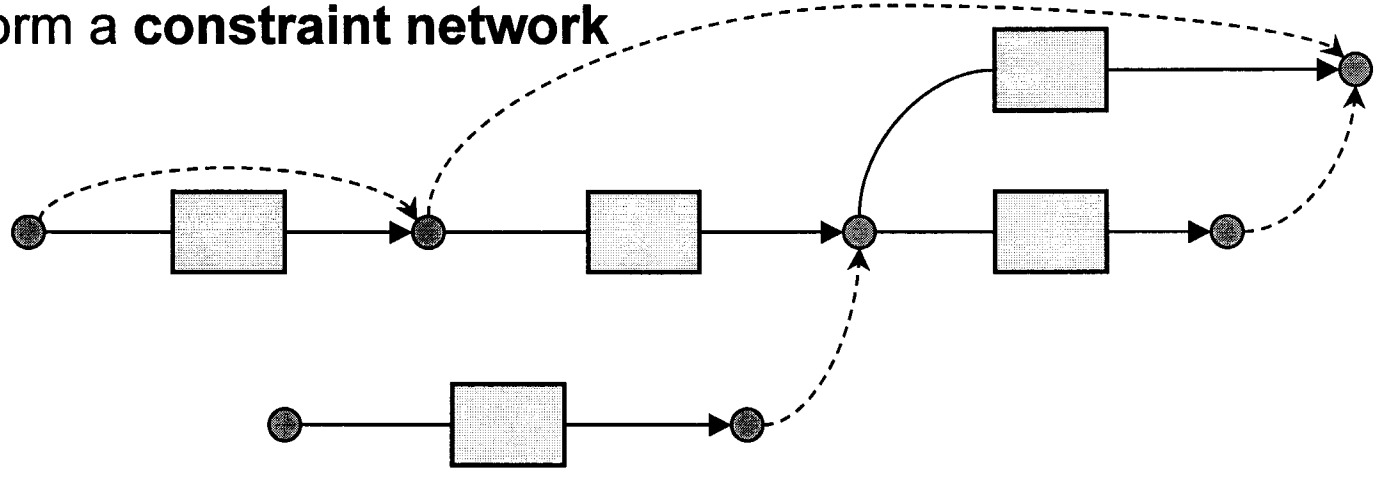


Constraint Networks

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

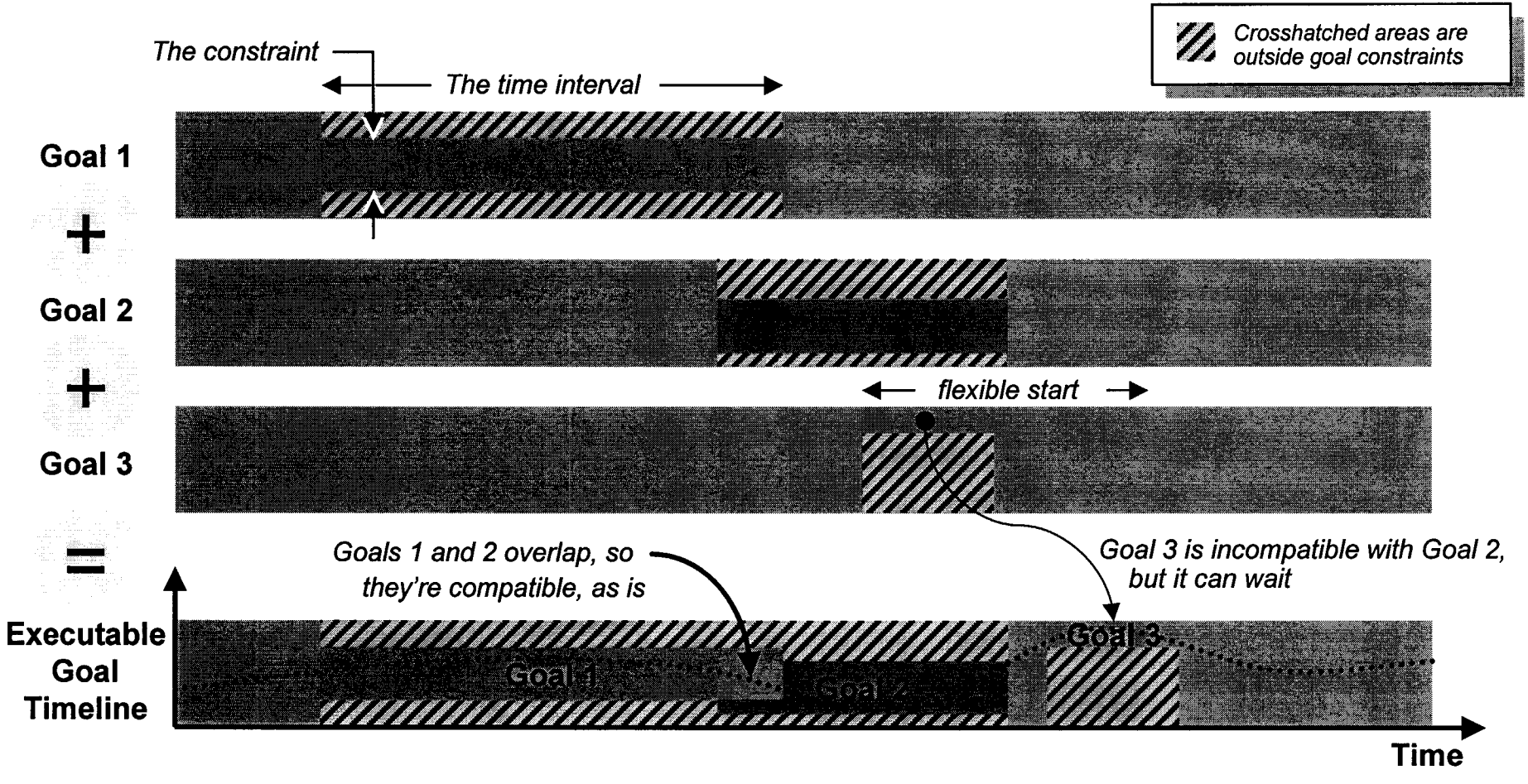


- Time points are often shared (e.g., one beginning as another ends)
- A collection of connected goals and temporal constraints form a **constraint network**



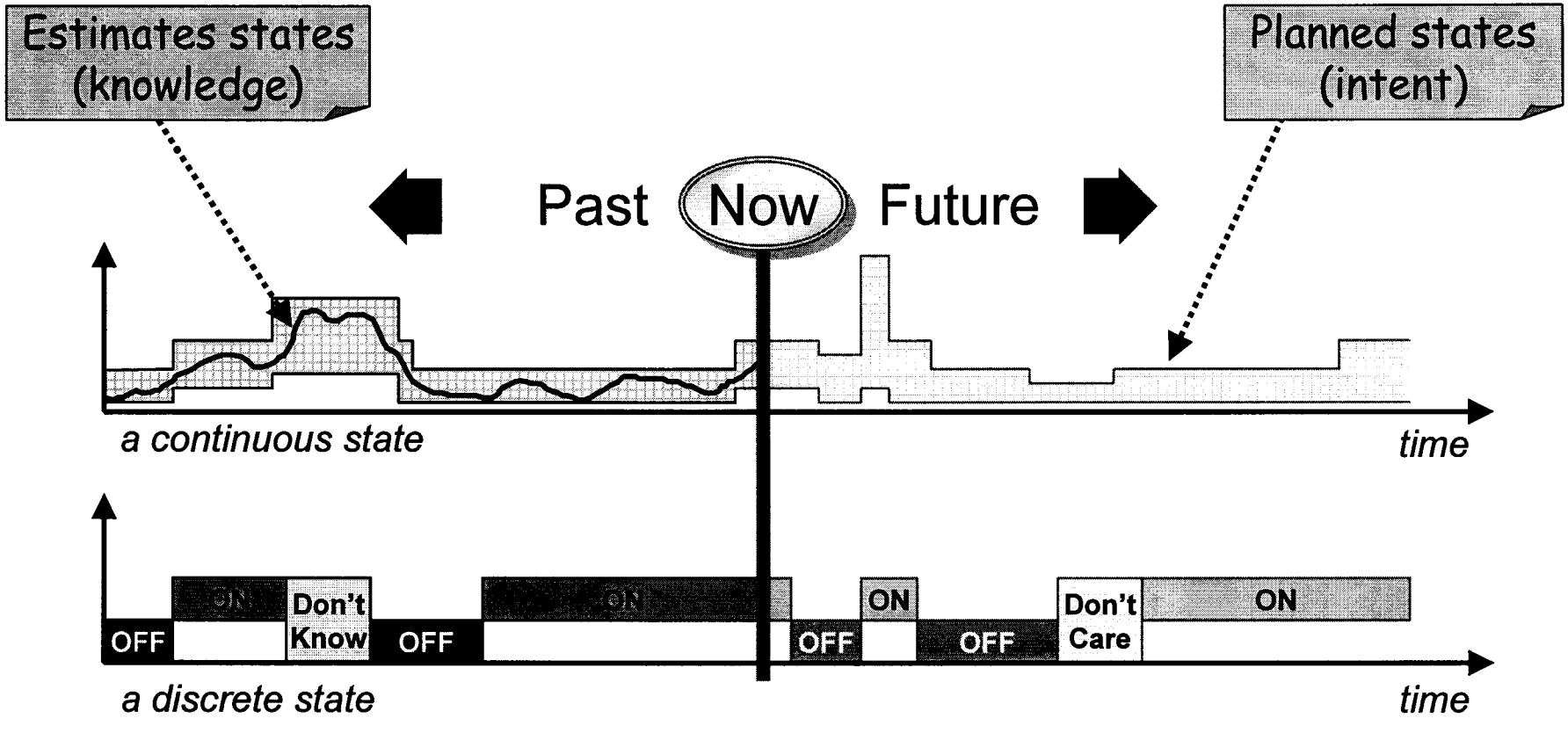
Resolving Conflicts

- Example: three goals on the same state



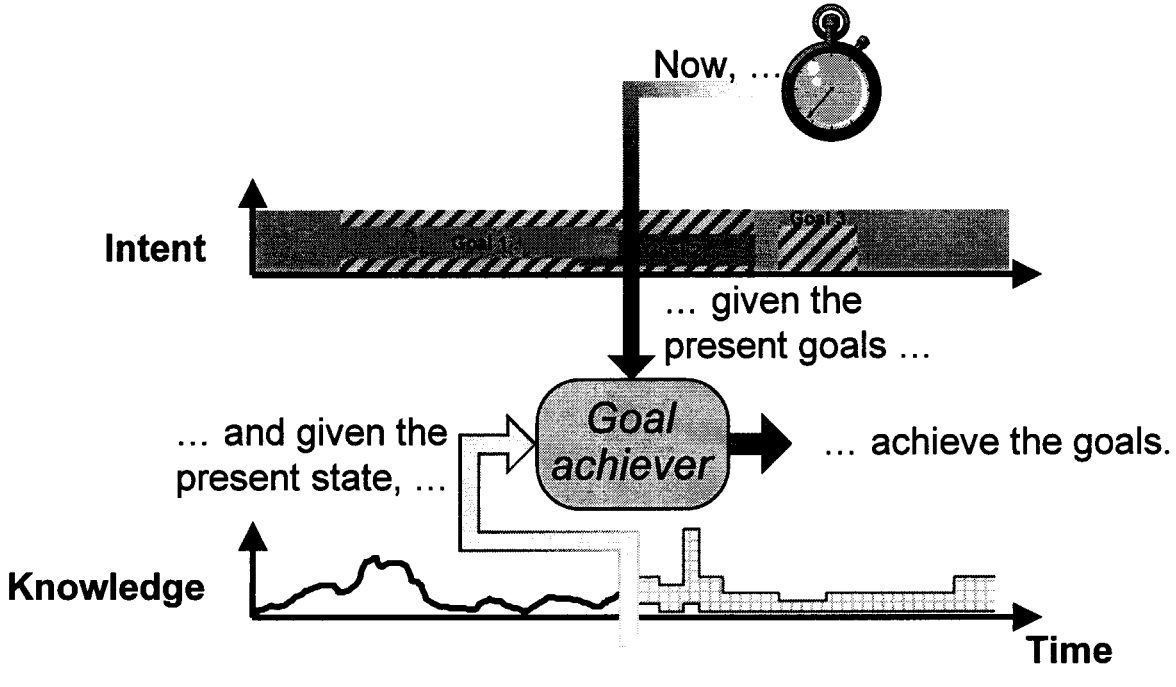
State Timelines

- **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



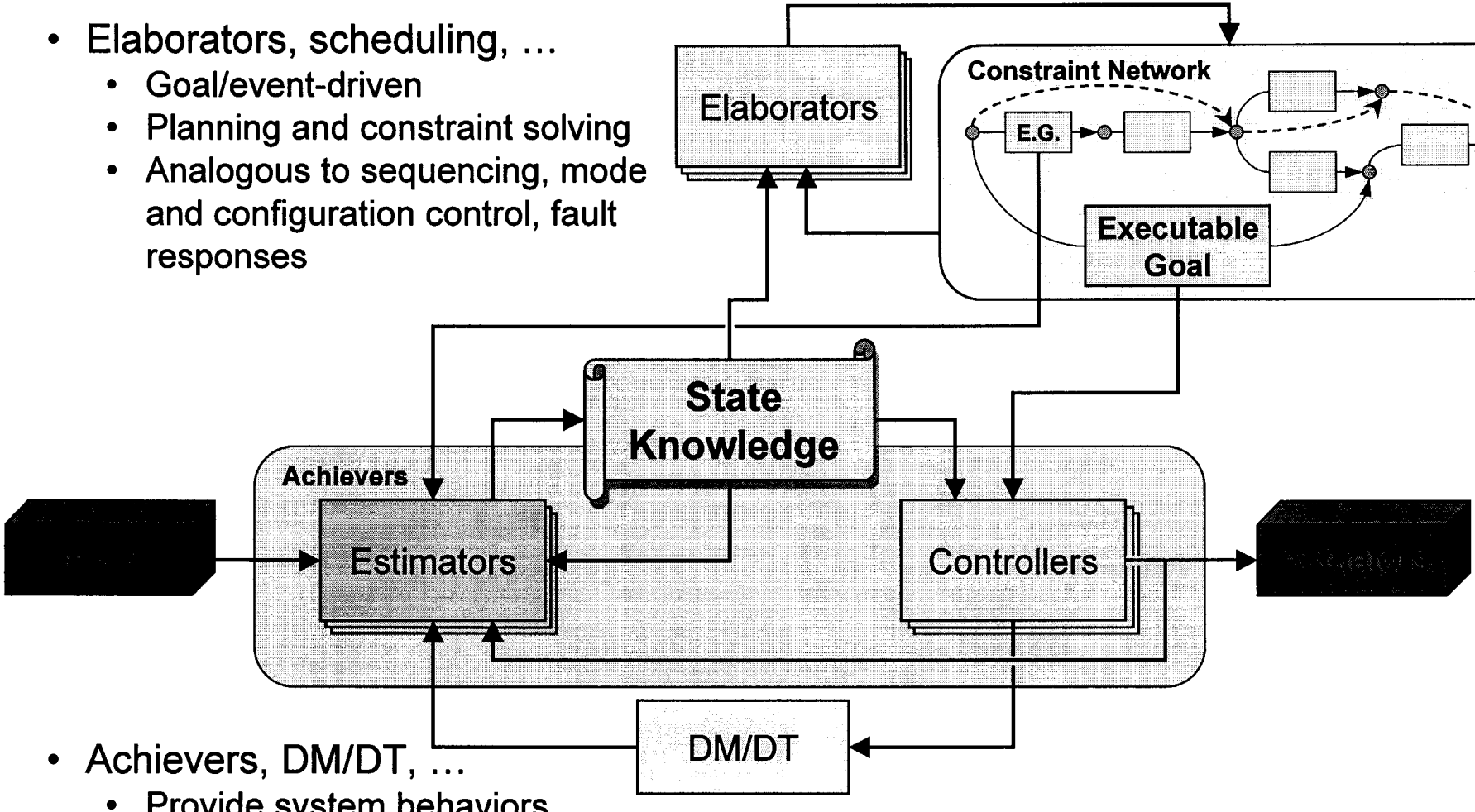
Timeline Execution

- 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



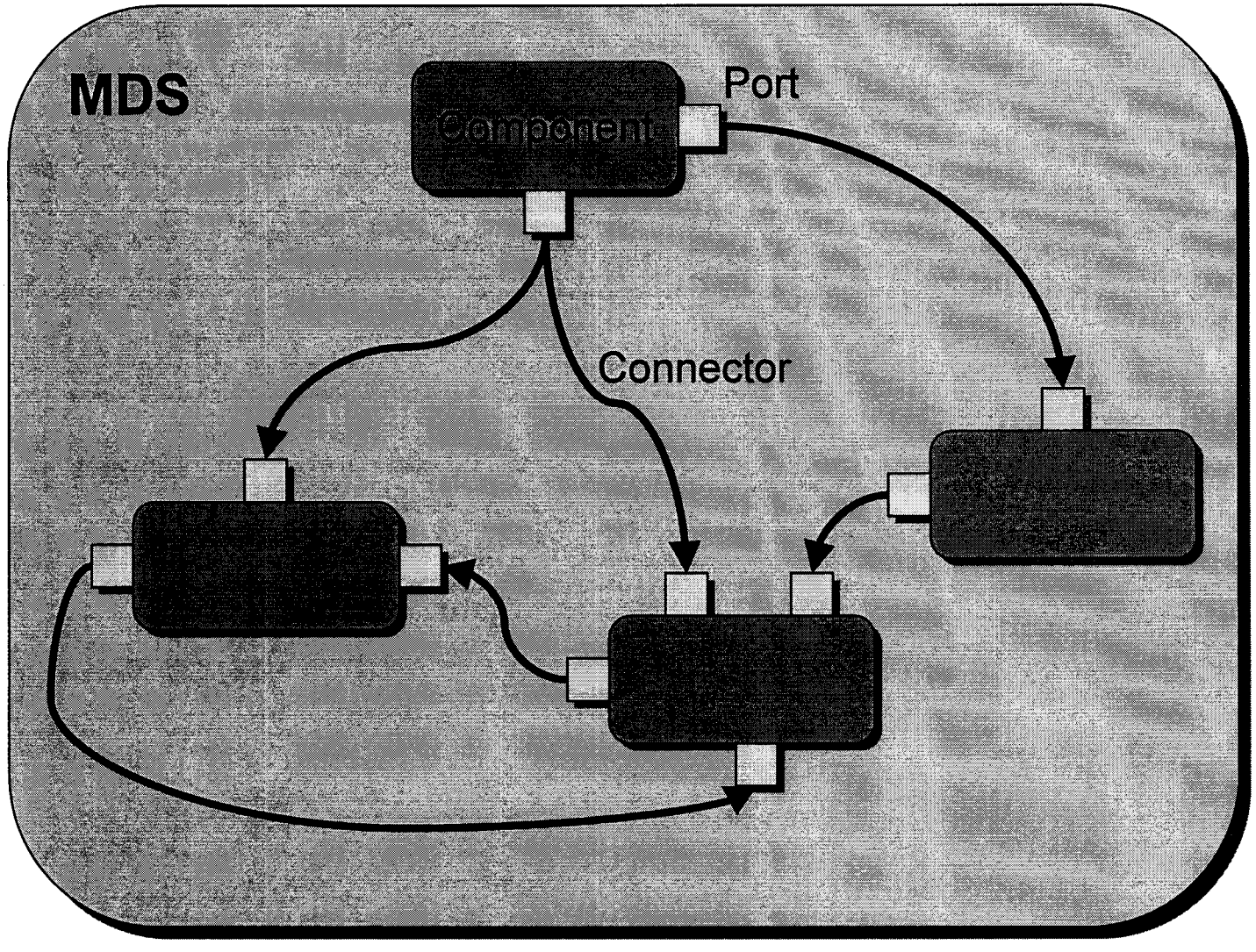
Putting It Together

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



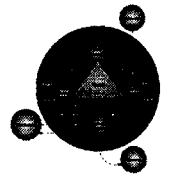
- Achievers, DM/DT, …
  - Provide system behaviors
  - Managed via goals and temporal constraints
  - Fairly conventional real-time monitoring and control processes

Component-Based Architecture



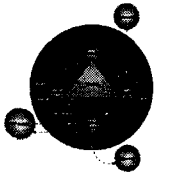
- First class types and instances
  - Components, connectors, interfaces
- Explicit interface ports
  - Describe semantics of interface use
- Type safety
- Role safety

Benefits: Development & Testing



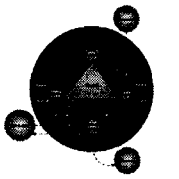
- Complex interactions understood at mission, system and component levels
- State decomposition facilitates tracking of system scalability
- Domain knowledge expressed explicitly as models
- Component architecture facilitates measures of responsiveness
- Architecture requires state be determined honestly from the evidence
- Architecture provides a mechanism for identifying the range and impact of faults
- Architecture authorizes and monitors all resource usage
- Uniform architecture facilitates cost tracking associated with system complexity and system reconfiguration.
- Cost models based on objective development data facilitate estimation of fault-related development.

Benefits: Safer Software



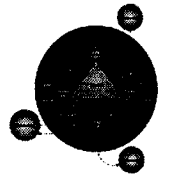
- Initialization/finalization dependencies explicitly represented
  - Dependency order enforced
  - Automatic detection of circular dependencies
- Software interfaces and interactions elevated to first-class design elements
  - Eliminates 'hidden' interactions that cause hard-to-find problems
- Component approach facilitates use of advanced verification techniques
  - Detect synchronization errors that cause deadlocks and data races
- Run-time component manager enforces architectural rules regarding instantiation of components and valid connections

Benefits: Verification by Design



- Goals specify explicit constraints on state and time that are continually monitored, so deviations from expected behavior are immediately reported.
- Errors in units of measurement detected through SI units package
- Component manager enforces architectural rules about legal connections
- The initialization/finalization package reports circular dependencies and improperly held resources
- Unified state architecture supports direct comparison between simulated state and estimated state
- Separation of models from reusable algorithms makes validation of mission-specific items simpler
- 'Smart pointers' eliminate problems of memory leaks and dangling pointers

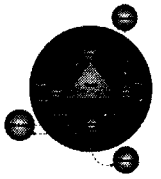
Benefits: Mission Operations

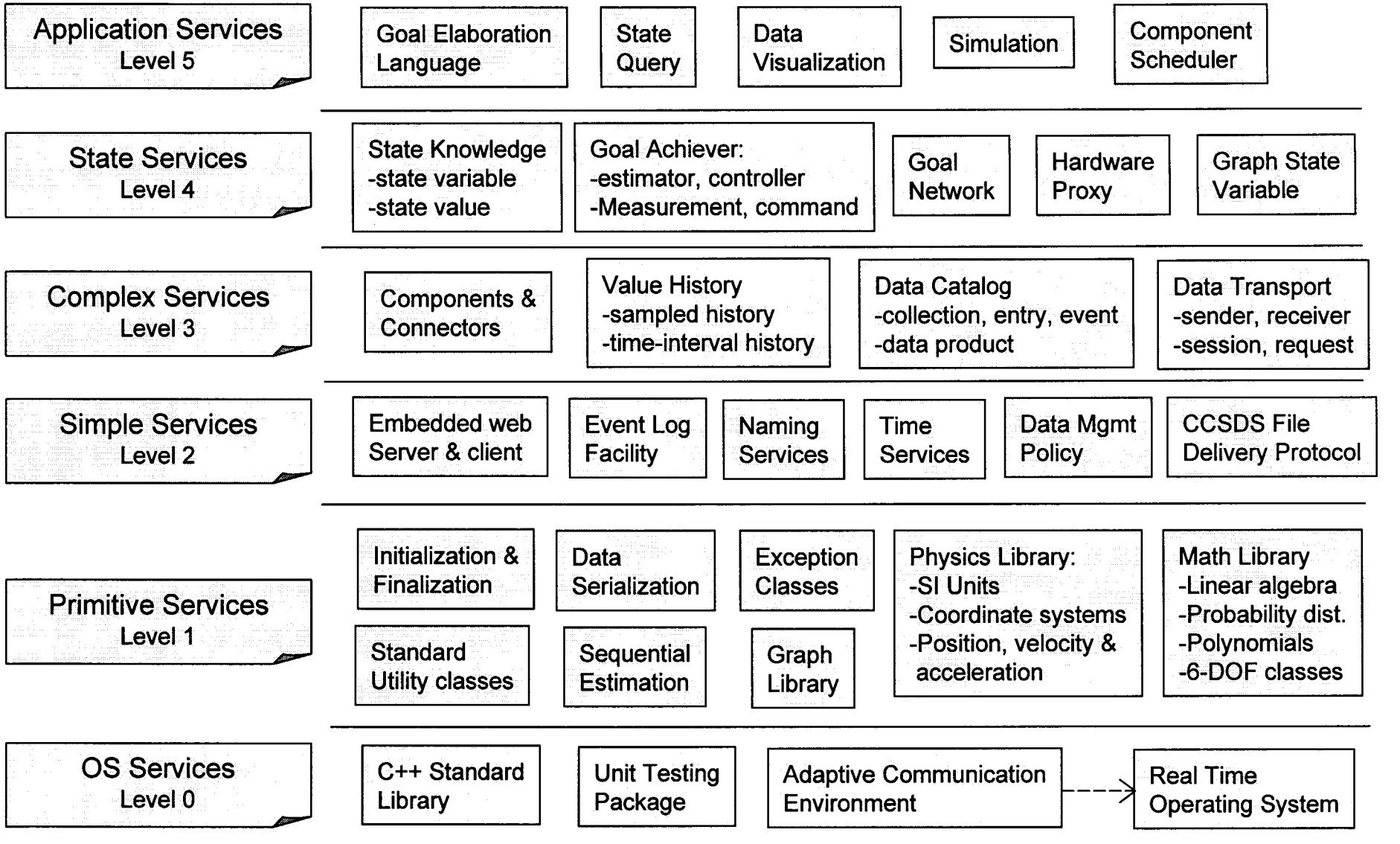


- Operation based on *what* (constraints on state) rather than *how* (command sequences)
- Operators can migrate capability and responsibility from ground to flight system to simplify operations and reduce communication needs
- Models express key functional relations among system states, command effects, and observed measurements
- Disciplined architecture provides uniform fault-metric mechanisms
- Fault sources are explicitly captured in goal-failure trees
- Fault response is explicitly modeled as goal elaborations
- Uniform data collection mechanism facilitates run-time monitoring and metrics associated with adaptivity and diagnosability

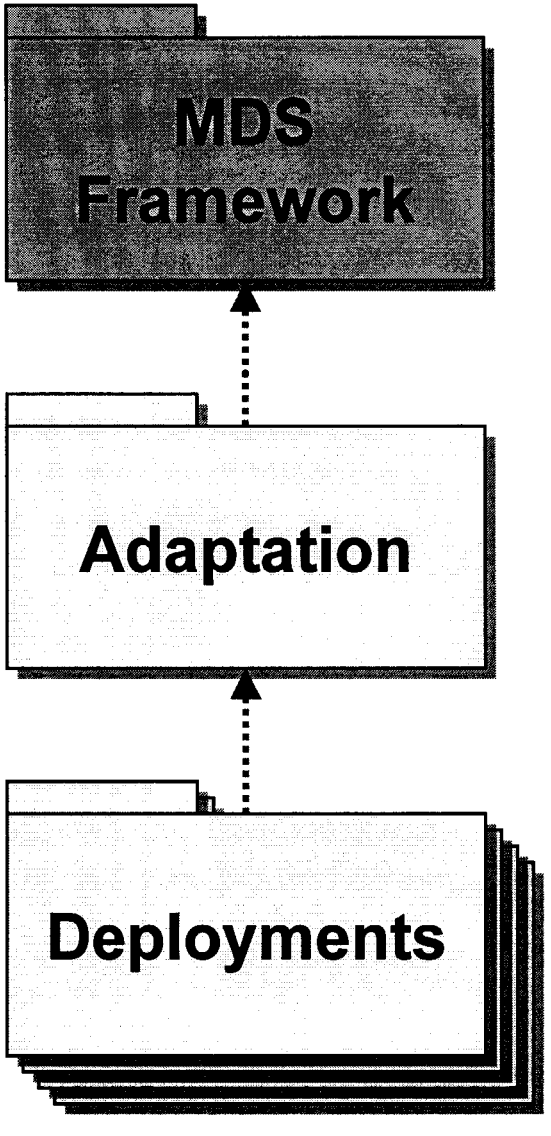
MDS Framework Packages

Each requirements is generally levied on a particular package



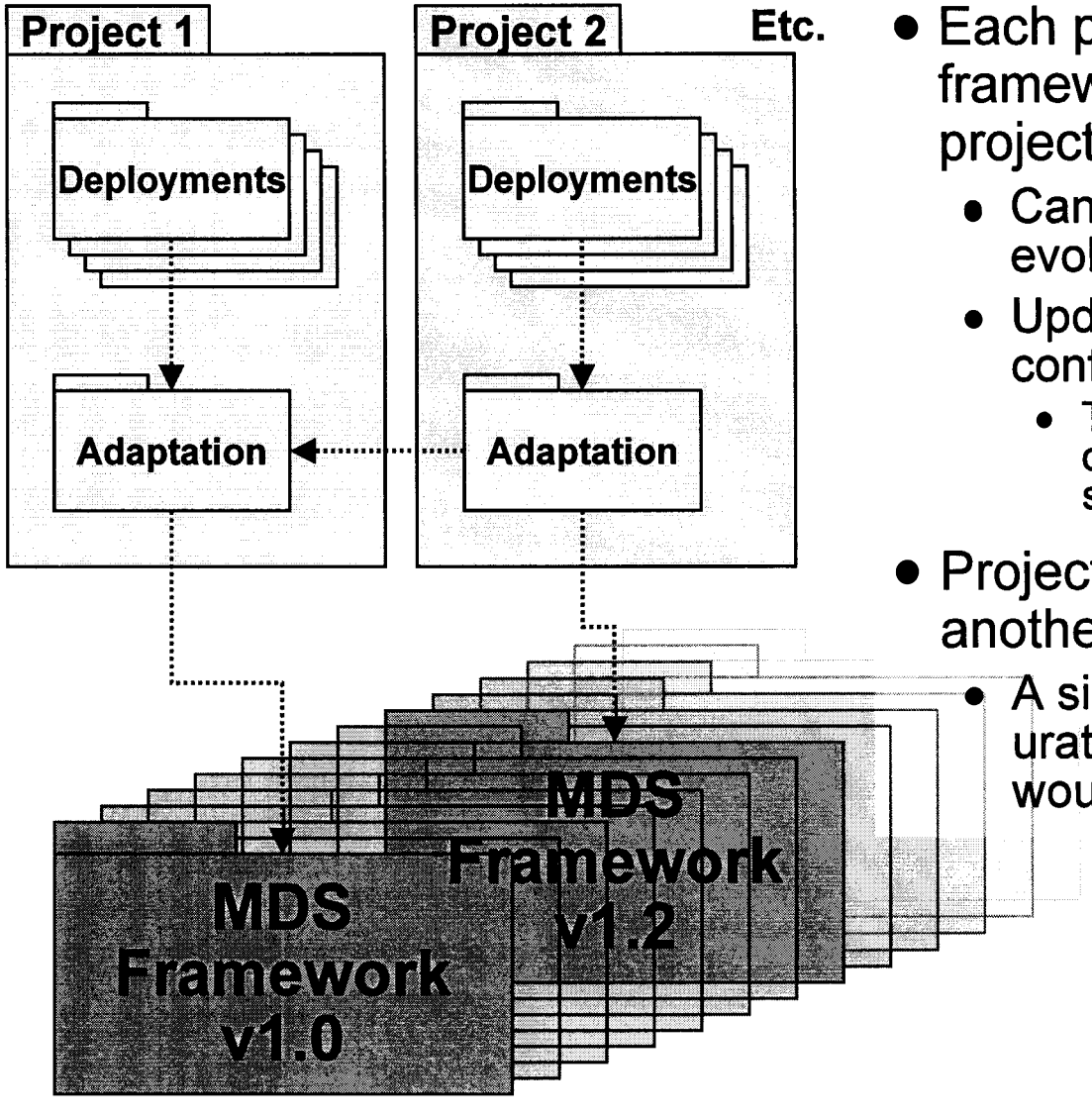


The MDS Common Model



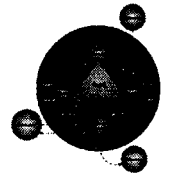
- The **MDS Framework** is the collection of most core classes within the MDS architecture
  - Developed and maintained exclusively by MDS
  - Uniform (except for versioning) across MDS adaptations
- Each project does an **Adaptation** of the framework
  - Captures project requirements and scenarios
  - 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.)

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

Summary



- MDS addresses…
  - Architectures for both functional and software design interactions
  - Unification and reuse across deployments and projects
  - A wide range of technical issues from autonomous control to data management
  - The collaboration of systems and software engineering
  - Processes, tools, and design rigor up to 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

End