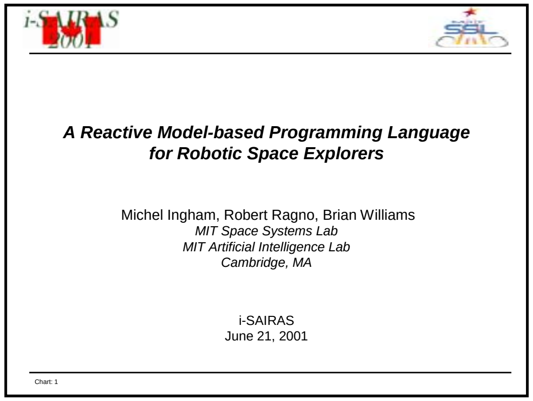
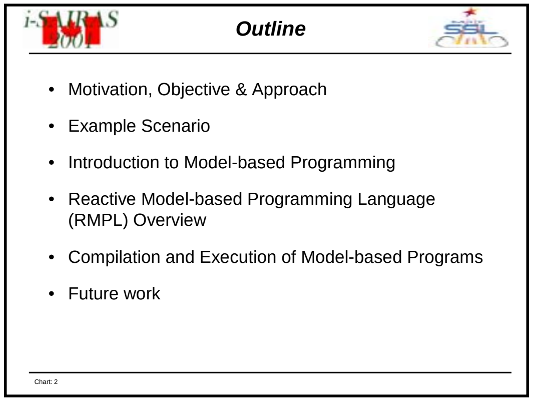


A Reactive Model-based Programming Language for Robotic Space Explorers

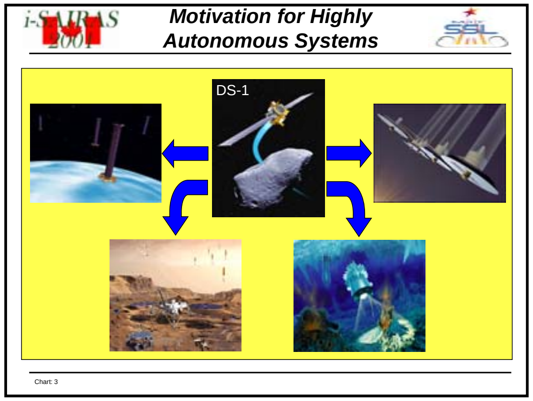
Michel Ingham, Robert Ragno, Brian Williams
MIT Space Systems Lab
MIT Artificial Intelligence Lab
Cambridge, MA

i-SAIRAS
June 21, 2001

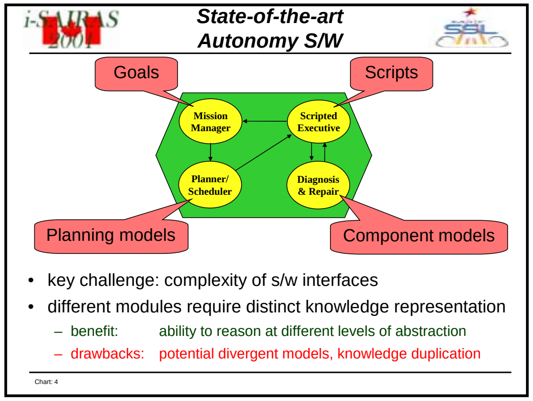


Outline

- Motivation, Objective & Approach
- Example Scenario
- Introduction to Model-based Programming
- Reactive Model-based Programming Language (RMPL) Overview
- Compilation and Execution of Model-based Programs
- Future work

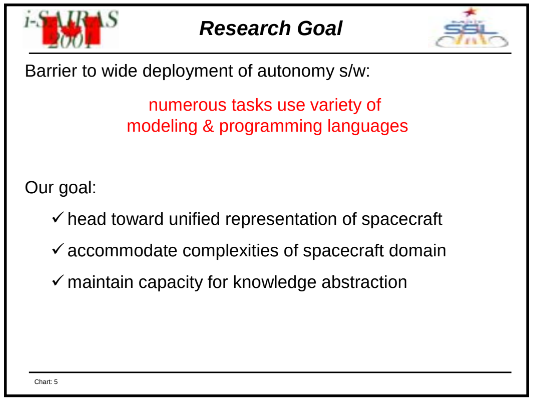


Motivation for Highly Autonomous Systems



State-of-the-art Autonomy S/W

- key challenge: complexity of s/w interfaces
- different modules require distinct knowledge representation
  - benefit: ability to reason at different levels of abstraction
  - drawbacks: potential divergent models, knowledge duplication



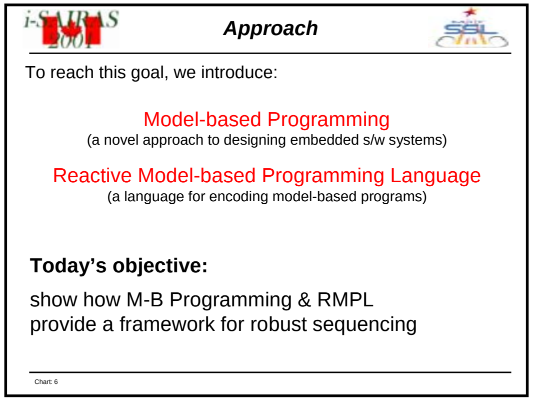
Research Goal

Barrier to wide deployment of autonomy s/w:

numerous tasks use variety of modeling & programming languages

Our goal:

✓ head toward unified representation of spacecraft
✓ accommodate complexities of spacecraft domain
✓ maintain capacity for knowledge abstraction



Approach

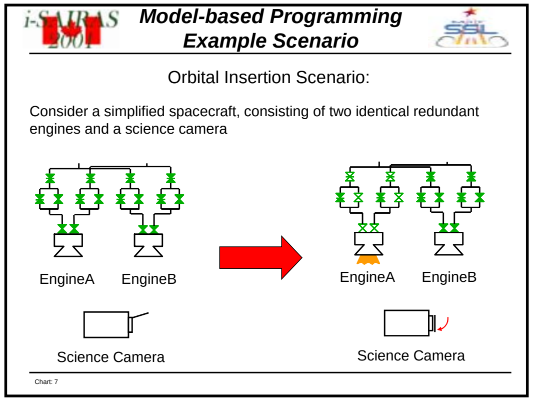
To reach this goal, we introduce:

Model-based Programming
(a novel approach to designing embedded s/w systems)

Reactive Model-based Programming Language
(a language for encoding model-based programs)

Today's objective:

show how M-B Programming & RMPL provide a framework for robust sequencing

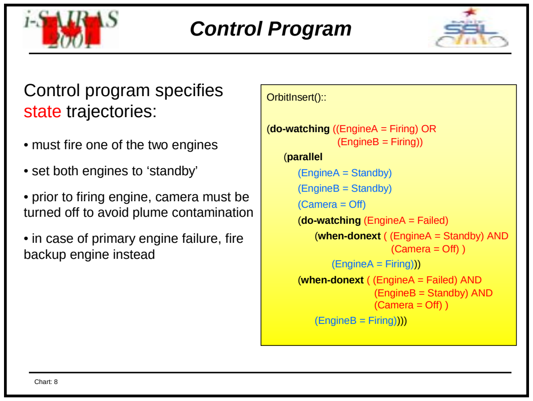


Model-based Programming Example Scenario

Orbital Insertion Scenario:

Consider a simplified spacecraft, consisting of two identical redundant engines and a science camera

Chart: 7



Control Program

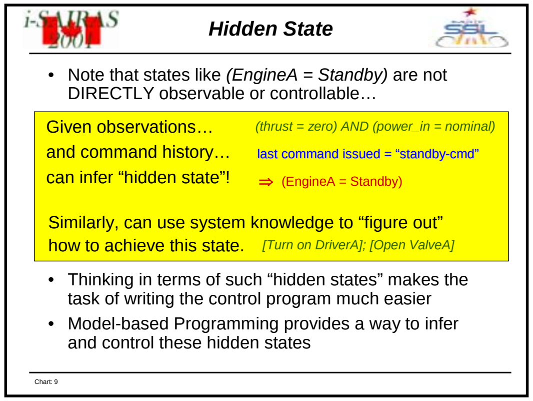
Control program specifies state trajectories:

- must fire one of the two engines
- set both engines to ‘standby’
- prior to firing engine, camera must be turned off to avoid plume contamination
- in case of primary engine failure, fire backup engine instead

```
OrbitInsert()::

(do-watching ((EngineA = Firing) OR
              (EngineB = Firing))
    (parallel
       (EngineA = Standby)
       (EngineB = Standby)
       (Camera = Off)
       (do-watching (EngineA = Failed)
          (when-donext ( (EngineA = Standby) AND
                         (Camera = Off) )
             (EngineA = Firing)))
       (when-donext ( (EngineA = Failed) AND
                      (EngineB = Standby) AND
                      (Camera = Off) )
          (EngineB = Firing))))
```

Chart: 8



Hidden State

- Note that states like *(EngineA = Standby)* are not DIRECTLY observable or controllable…

Given observations… *(thrust = zero) AND (power_in = nominal)*

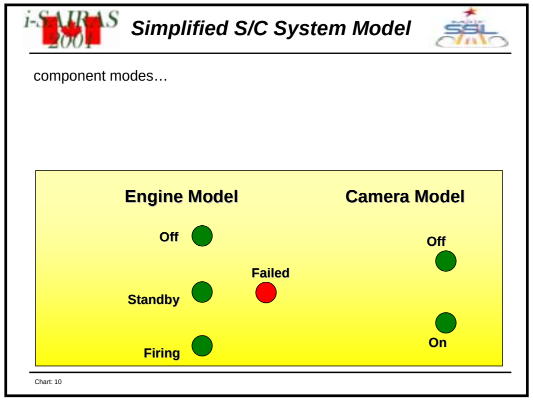
and command history… last command issued = “standby-cmd”

can infer “hidden state”! ⇒ (EngineA = Standby)

Similarly, can use system knowledge to “figure out” how to achieve this state. *[Turn on DriverA]; [Open ValveA]*

- Thinking in terms of such “hidden states” makes the task of writing the control program much easier
- Model-based Programming provides a way to infer and control these hidden states

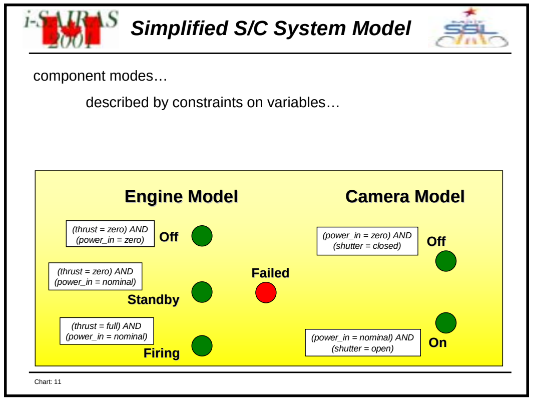
Chart: 9



Simplified S/C System Model

component modes…

Chart: 10



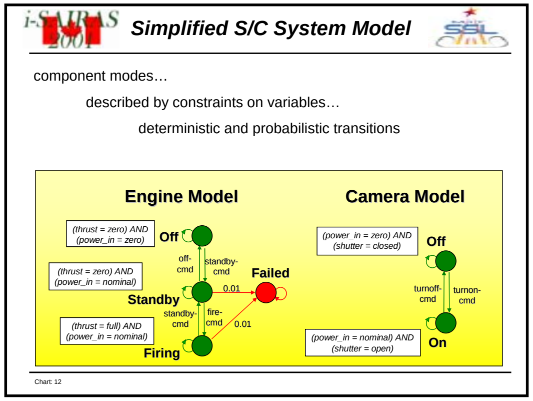
Simplified S/C System Model

component modes…

described by constraints on variables…

Engine Model
Camera Model
(thrust = zero) AND (power_in = zero)
Off
(thrust = zero) AND (power_in = nominal)
Standby
(thrust = full) AND (power_in = nominal)
Firing
Failed
(power_in = zero) AND (shutter = closed)
Off
(power_in = nominal) AND (shutter = open)
On

Chart: 11



Simplified S/C System Model

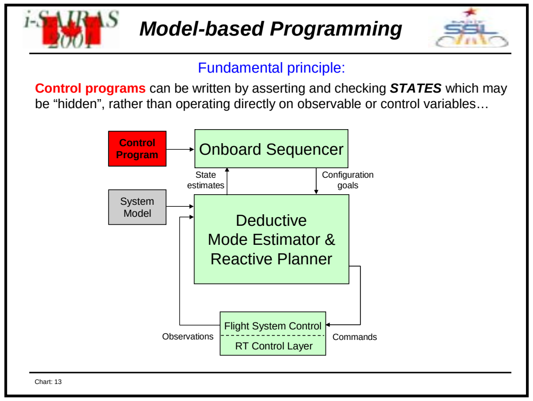
component modes…

described by constraints on variables…

deterministic and probabilistic transitions

Engine Model
Camera Model
(thrust = zero) AND (power_in = zero)
Off
off-cmd
standby-cmd
(thrust = zero) AND (power_in = nominal)
Standby
0.01
Failed
standby-cmd
fire-cmd
0.01
(thrust = full) AND (power_in = nominal)
Firing
(power_in = zero) AND (shutter = closed)
Off
turnoff-cmd
turnon-cmd
(power_in = nominal) AND (shutter = open)
On

Chart: 12

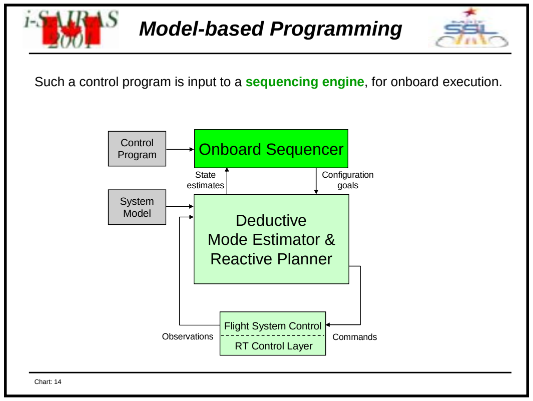


Model-based Programming

Fundamental principle:

Control programs can be written by asserting and checking ***STATES*** which may be “hidden”, rather than operating directly on observable or control variables…

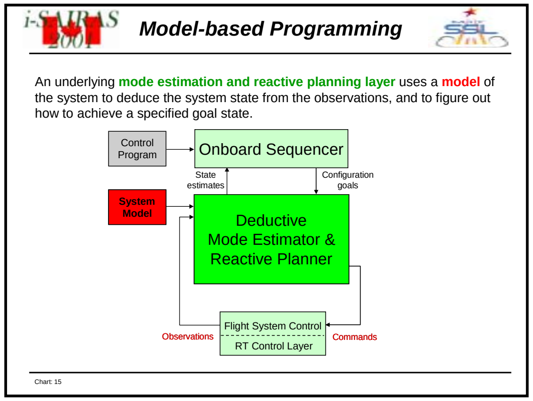
Chart: 13



Model-based Programming

Such a control program is input to a **sequencing engine**, for onboard execution.

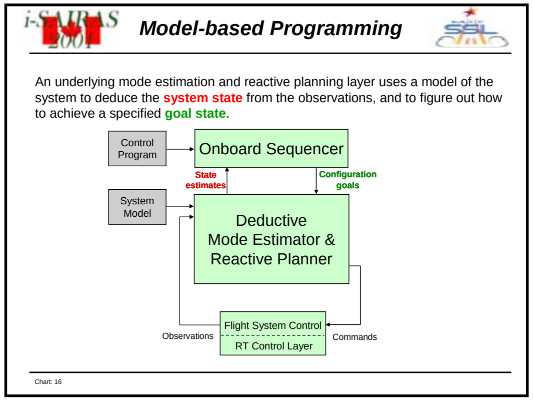
Chart: 14



Model-based Programming

An underlying **mode estimation and reactive planning layer** uses a **model** of the system to deduce the system state from the observations, and to figure out how to achieve a specified goal state.

Chart: 15



Model-based Programming

An underlying mode estimation and reactive planning layer uses a model of the system to deduce the **system state** from the observations, and to figure out how to achieve a specified **goal state**.

Chart: 16

RMPL Constructs

• constraint	*c*
• concurrency	(parallel *exp1 exp2* …)
• sequential ordering	(sequence *exp1 exp2* …)
• conditional branching	(if-thennext-elsenext *c then-exp else-exp*)
• guarded transition	(unless-thennext *c exp*)
• iteration	(always *exp*)
• extended guarded transition	(when-donext *c exp*)
• iterated guarded transition	(whenever-donext *c exp*)
• preemption	(do-watching *c exp*)

Chart: 17

RMPL Model of Computation

- To support efficient execution, RMPL code is compiled into Hierarchical Constraint Automata (HCA):

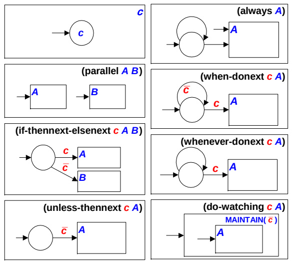


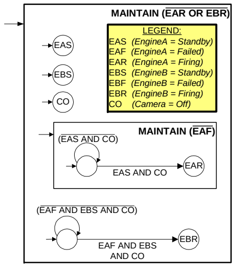
Chart: 18

RMPL Control Program

```
OrbitInsert()::
(do-watching ((EngineA = Firing) OR
              (EngineB = Firing))
   (parallel
      (EngineA = Standby)
      (EngineB = Standby)
      (Camera = Off)
      (do-watching (EngineA = Failed)
         (when-donext ( (EngineA = Standby) AND
                        (Camera = Off) )
            (EngineA = Firing)))
      (when-donext ( (EngineA = Failed) AND
                     (EngineB = Standby) AND
                     (Camera = Off) )
         (EngineB = Firing))))
```

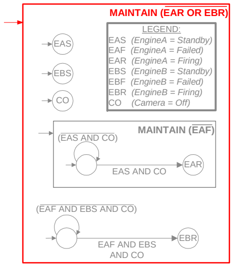
Compiling RMPL to HCA

```
OrbitInsert()::
(do-watching ((EngineA = Firing) OR
              (EngineB = Firing))
   (parallel
      (EngineA = Standby)
      (EngineB = Standby)
      (Camera = Off)
      (do-watching (EngineA = Failed)
         (when-donext ( (EngineA = Standby) AND
                        (Camera = Off) )
            (EngineA = Firing)))
      (when-donext ( (EngineA = Failed) AND
                     (EngineB = Standby) AND
                     (Camera = Off) )
         (EngineB = Firing))))
```



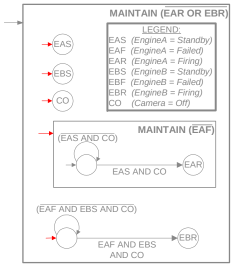
Compiling RMPL to HCA

```
OrbitInsert()::
(do-watching ((EngineA = Firing) OR
              (EngineB = Firing))
   (parallel
      (EngineA = Standby)
      (EngineB = Standby)
      (Camera = Off)
      (do-watching (EngineA = Failed)
         (when-donext ( (EngineA = Standby) AND
                        (Camera = Off) )
            (EngineA = Firing)))
      (when-donext ( (EngineA = Failed) AND
                     (EngineB = Standby) AND
                     (Camera = Off) )
         (EngineB = Firing))))
```



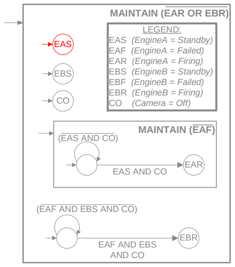
Compiling RMPL to HCA

```
OrbitInsert()::
(do-watching ((EngineA = Firing) OR
              (EngineB = Firing))
   (parallel
      (EngineA = Standby)
      (EngineB = Standby)
      (Camera = Off)
      (do-watching (EngineA = Failed)
         (when-donext ( (EngineA = Standby) AND
                        (Camera = Off) )
            (EngineA = Firing)))
      (when-donext ( (EngineA = Failed) AND
                     (EngineB = Standby) AND
                     (Camera = Off) )
         (EngineB = Firing))))
```



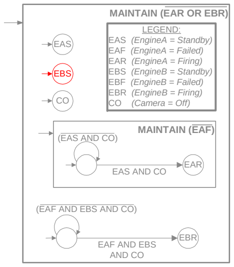
Compiling RMPL to HCA

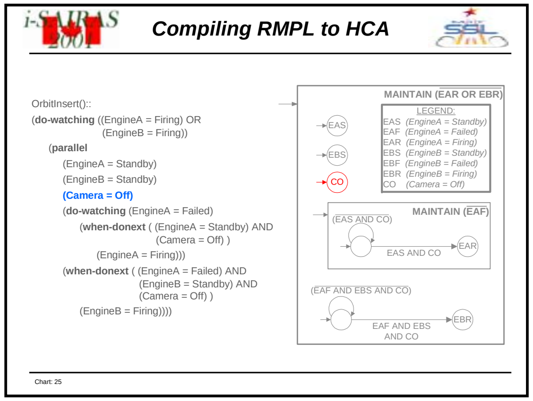
```
OrbitInsert()::
(do-watching ((EngineA = Firing) OR
              (EngineB = Firing))
   (parallel
      (EngineA = Standby)
      (EngineB = Standby)
      (Camera = Off)
      (do-watching (EngineA = Failed)
         (when-donext ( (EngineA = Standby) AND
                        (Camera = Off) )
            (EngineA = Firing)))
      (when-donext ( (EngineA = Failed) AND
                     (EngineB = Standby) AND
                     (Camera = Off) )
         (EngineB = Firing))))
```



Compiling RMPL to HCA

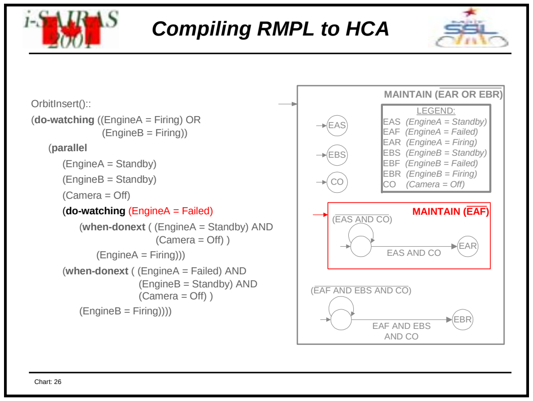
```
OrbitInsert()::
(do-watching ((EngineA = Firing) OR
              (EngineB = Firing))
   (parallel
      (EngineA = Standby)
      (EngineB = Standby)
      (Camera = Off)
      (do-watching (EngineA = Failed)
         (when-donext ( (EngineA = Standby) AND
                        (Camera = Off) )
            (EngineA = Firing)))
      (when-donext ( (EngineA = Failed) AND
                     (EngineB = Standby) AND
                     (Camera = Off) )
         (EngineB = Firing))))
```





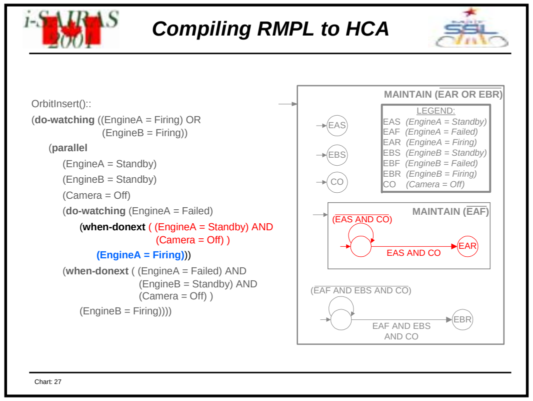
Compiling RMPL to HCA

```
OrbitInsert()::
(do-watching ((EngineA = Firing) OR
              (EngineB = Firing))
   (parallel
      (EngineA = Standby)
      (EngineB = Standby)
      (Camera = Off)
      (do-watching (EngineA = Failed)
         (when-donext ( (EngineA = Standby) AND
                        (Camera = Off) )
            (EngineA = Firing)))
      (when-donext ( (EngineA = Failed) AND
                     (EngineB = Standby) AND
                     (Camera = Off) )
         (EngineB = Firing))))
```



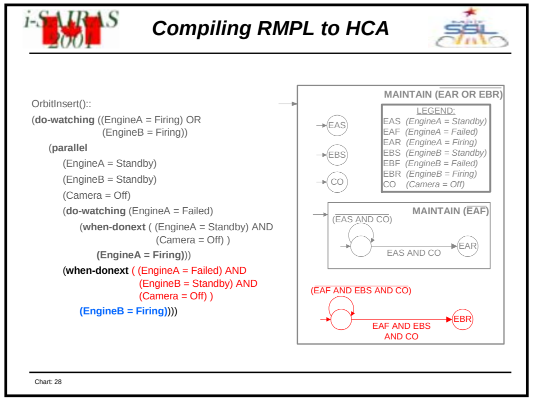
Compiling RMPL to HCA

```
OrbitInsert()::
(do-watching ((EngineA = Firing) OR
              (EngineB = Firing))
   (parallel
      (EngineA = Standby)
      (EngineB = Standby)
      (Camera = Off)
      (do-watching (EngineA = Failed)
         (when-donext ( (EngineA = Standby) AND
                        (Camera = Off) )
            (EngineA = Firing)))
      (when-donext ( (EngineA = Failed) AND
                     (EngineB = Standby) AND
                     (Camera = Off) )
         (EngineB = Firing))))
```



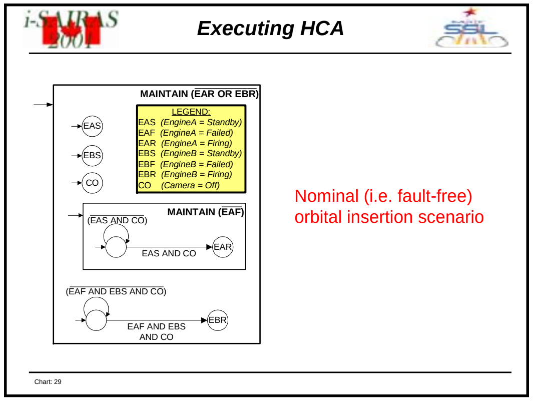
Compiling RMPL to HCA

```
OrbitInsert()::
(do-watching ((EngineA = Firing) OR
              (EngineB = Firing))
   (parallel
      (EngineA = Standby)
      (EngineB = Standby)
      (Camera = Off)
      (do-watching (EngineA = Failed)
         (when-donext ( (EngineA = Standby) AND
                        (Camera = Off) )
            (EngineA = Firing)))
      (when-donext ( (EngineA = Failed) AND
                     (EngineB = Standby) AND
                     (Camera = Off) )
         (EngineB = Firing))))
```



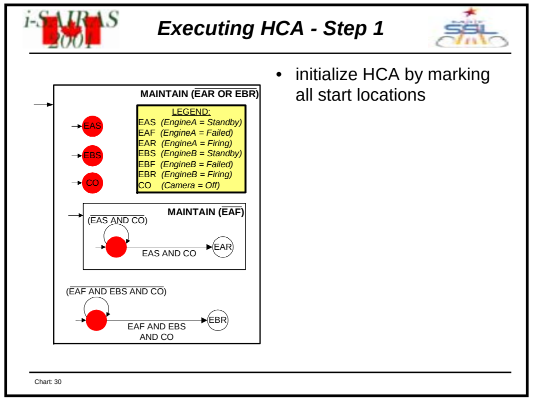
Compiling RMPL to HCA

```
OrbitInsert()::
(do-watching ((EngineA = Firing) OR
              (EngineB = Firing))
   (parallel
      (EngineA = Standby)
      (EngineB = Standby)
      (Camera = Off)
      (do-watching (EngineA = Failed)
         (when-donext ( (EngineA = Standby) AND
                        (Camera = Off) )
            (EngineA = Firing)))
      (when-donext ( (EngineA = Failed) AND
                     (EngineB = Standby) AND
                     (Camera = Off) )
         (EngineB = Firing))))
```



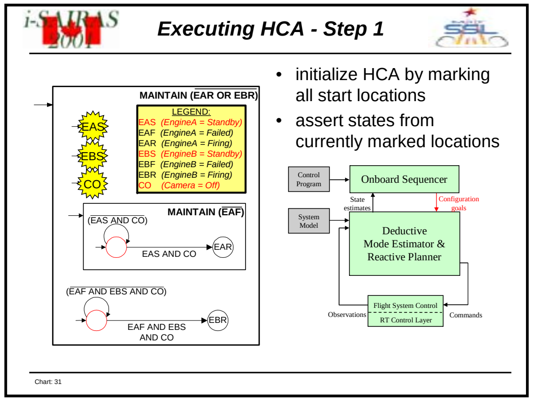
Executing HCA

Nominal (i.e. fault-free) orbital insertion scenario



Executing HCA - Step 1

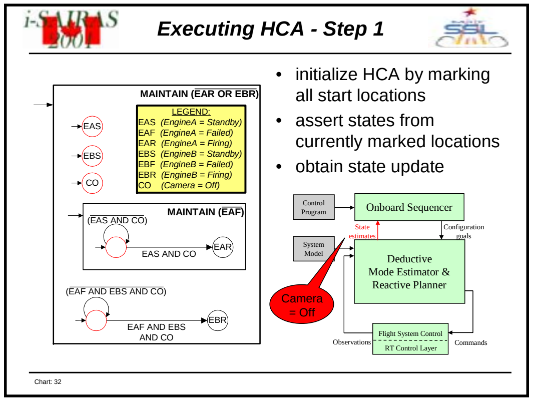
- initialize HCA by marking all start locations



Executing HCA - Step 1

- initialize HCA by marking all start locations
- assert states from currently marked locations

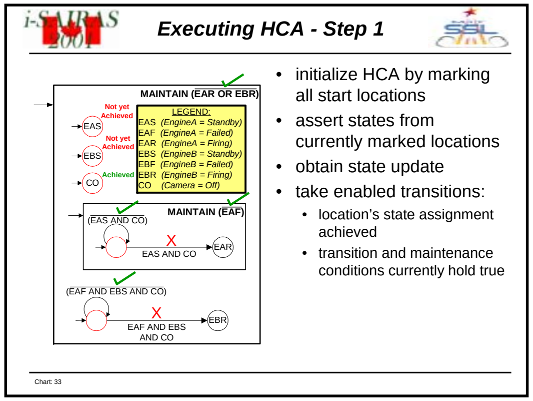
Chart: 31



Executing HCA - Step 1

- initialize HCA by marking all start locations
- assert states from currently marked locations
- obtain state update

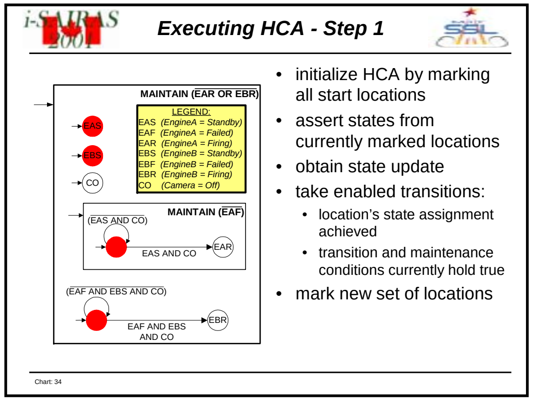
Chart: 32



Executing HCA - Step 1

- initialize HCA by marking all start locations
- assert states from currently marked locations
- obtain state update
- take enabled transitions:
  - location's state assignment achieved
  - transition and maintenance conditions currently hold true

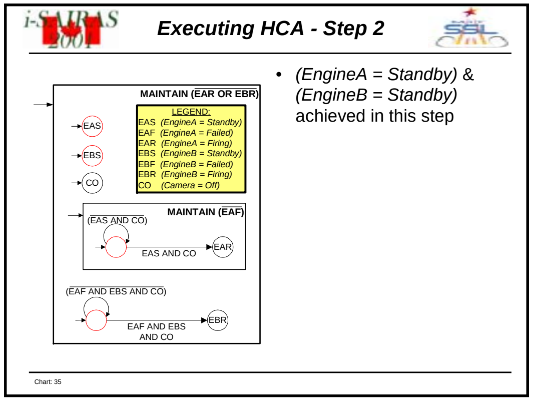
Chart: 33



Executing HCA - Step 1

- initialize HCA by marking all start locations
- assert states from currently marked locations
- obtain state update
- take enabled transitions:
  - location's state assignment achieved
  - transition and maintenance conditions currently hold true
- mark new set of locations

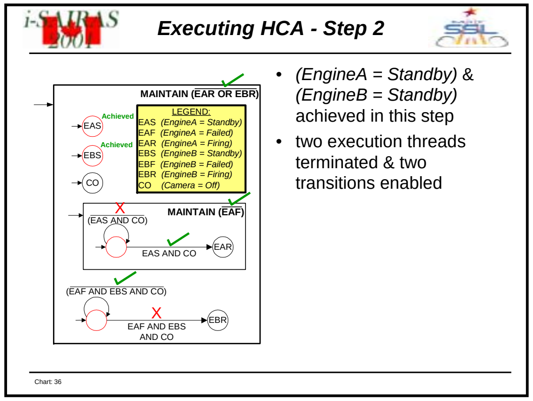
Chart: 34



Executing HCA - Step 2

- *(EngineA = Standby)* & *(EngineB = Standby)* achieved in this step

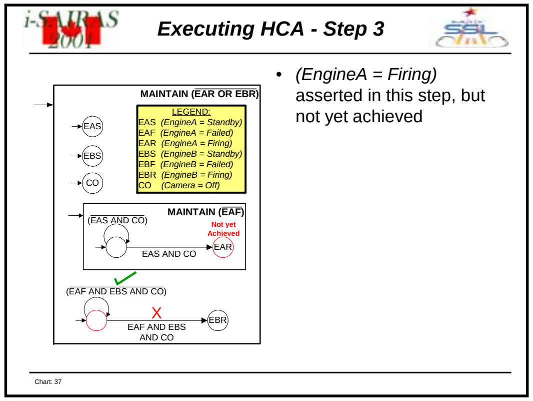
Chart: 35



Executing HCA - Step 2

- *(EngineA = Standby)* & *(EngineB = Standby)* achieved in this step
- two execution threads terminated & two transitions enabled

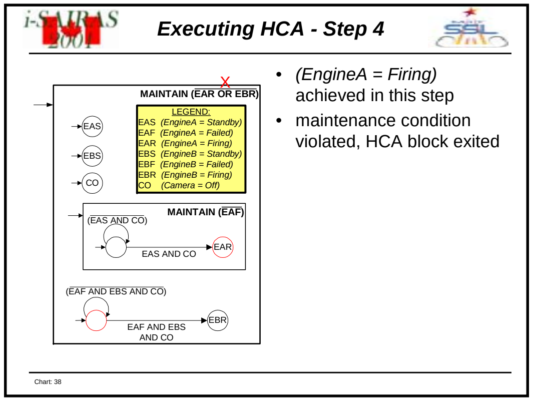
Chart: 36



Executing HCA - Step 3

- *(EngineA = Firing)* asserted in this step, but not yet achieved

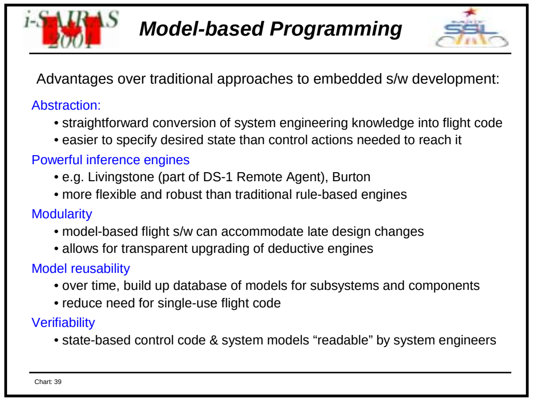
Chart: 37



Executing HCA - Step 4

- *(EngineA = Firing)* achieved in this step
- maintenance condition violated, HCA block exited

Chart: 38



Model-based Programming

Advantages over traditional approaches to embedded s/w development:

Abstraction:
- straightforward conversion of system engineering knowledge into flight code
- easier to specify desired state than control actions needed to reach it

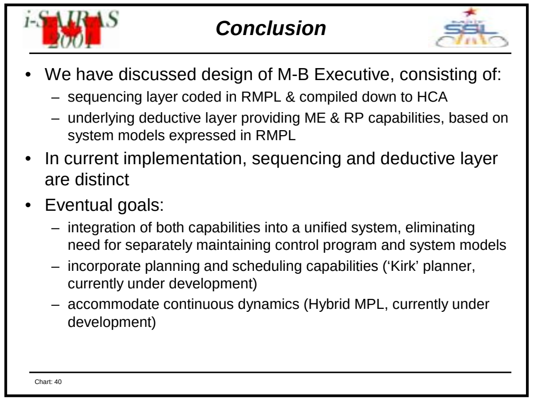
Powerful inference engines
- e.g. Livingstone (part of DS-1 Remote Agent), Burton
- more flexible and robust than traditional rule-based engines

Modularity
- model-based flight s/w can accommodate late design changes
- allows for transparent upgrading of deductive engines

Model reusability
- over time, build up database of models for subsystems and components
- reduce need for single-use flight code

Verifiability
- state-based control code & system models “readable” by system engineers

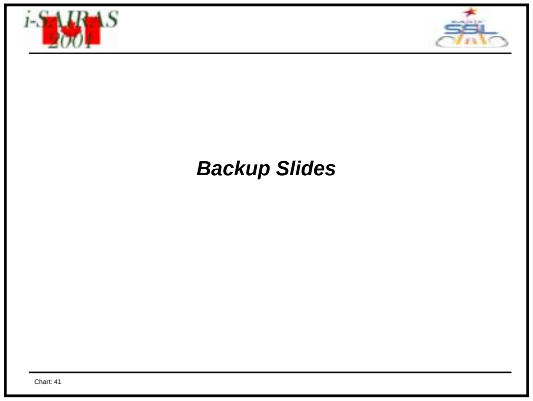
Chart: 39



Conclusion

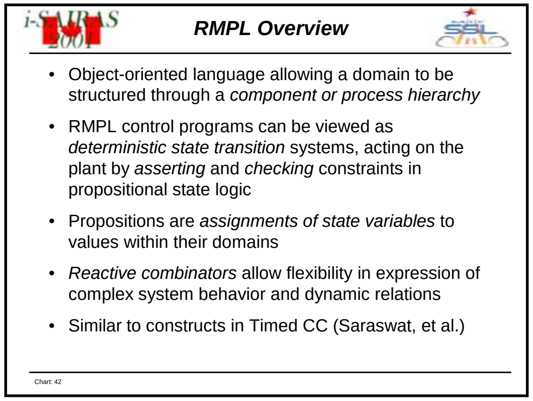
- We have discussed design of M-B Executive, consisting of:
  - sequencing layer coded in RMPL & compiled down to HCA
  - underlying deductive layer providing ME & RP capabilities, based on system models expressed in RMPL
- In current implementation, sequencing and deductive layer are distinct
- Eventual goals:
  - integration of both capabilities into a unified system, eliminating need for separately maintaining control program and system models
  - incorporate planning and scheduling capabilities (‘Kirk’ planner, currently under development)
  - accommodate continuous dynamics (Hybrid MPL, currently under development)

Chart: 40



Backup Slides

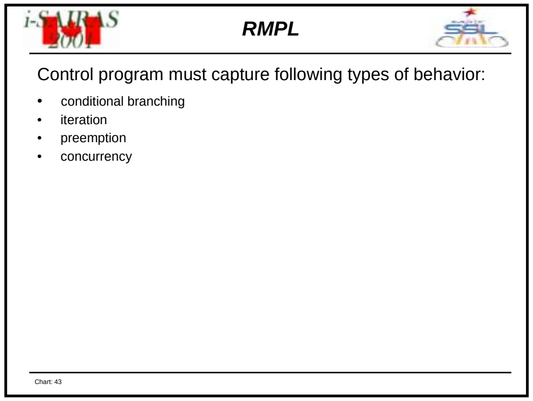
Chart: 41



RMPL Overview

- Object-oriented language allowing a domain to be structured through a *component or process hierarchy*
- RMPL control programs can be viewed as *deterministic state transition* systems, acting on the plant by *asserting* and *checking* constraints in propositional state logic
- Propositions are *assignments of state variables* to values within their domains
- *Reactive combinators* allow flexibility in expression of complex system behavior and dynamic relations
- Similar to constructs in Timed CC (Saraswat, et al.)

Chart: 42

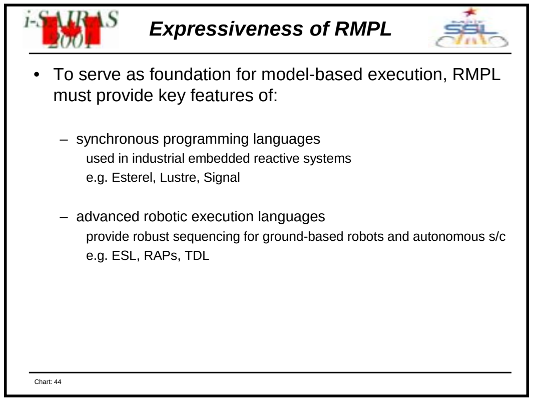


RMPL

Control program must capture following types of behavior:

- conditional branching
- iteration
- preemption
- concurrency

Chart: 43



Expressiveness of RMPL

- To serve as foundation for model-based execution, RMPL must provide key features of:
  - synchronous programming languages
    used in industrial embedded reactive systems
    e.g. Esterel, Lustre, Signal
  - advanced robotic execution languages
    provide robust sequencing for ground-based robots and autonomous s/c
    e.g. ESL, RAPs, TDL

Chart: 44