NASA Ames Research Center
Autonomous Systems and Robotics

PLEXIL Workshop

An Introduction to PLEXIL and the PLEXIL Executive

Part 1: Overview

This presentation was developed in Open Office Impress. Template courtesy of Chih-Hao Tsai (technology.cstsai.org/impress).

Outline

Background

Plexil plan structure

Plexil execution

What is a Plan?

A Plan is a course of action to achieve a set of *goals*.

The course of action may differ depending on the *initial state* the world is in.

Each action in a plan may only work properly when certain *conditions* are met.

When actions are executed, they may have certain *effects*.

What is PLEXIL?

PLEXIL is a language for expressing plans.

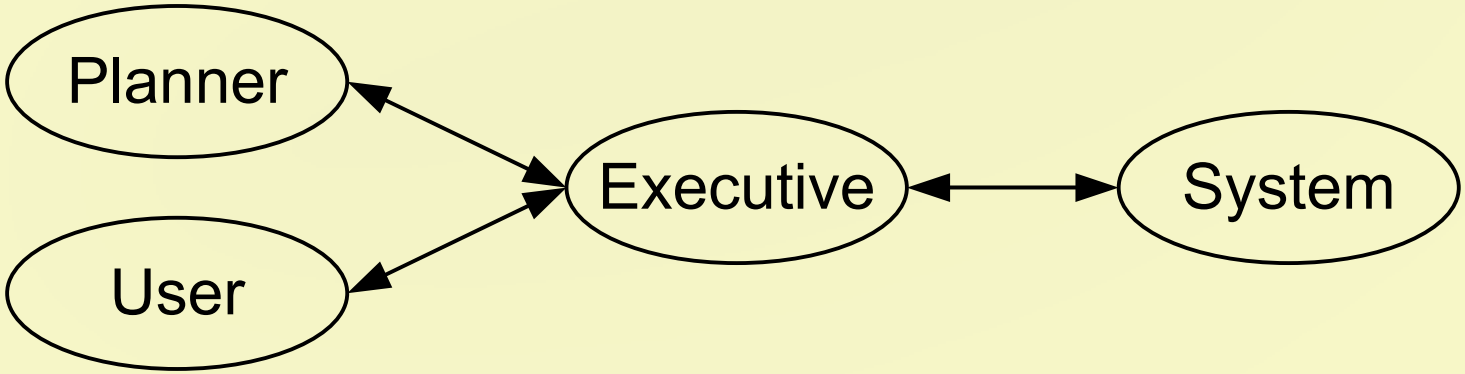
PLEXIL Design Goals

- Simple, small, uniform
- Efficient
- Well-defined
- Expressive
  - Sequences, branches, loops, concurrency
  - Time-driven, event-driven, condition-driven
- Enables verification and validation
  - Pre conditions, post conditions, invariants
- Deterministic

What is the PLEXIL Executive?

The PLEXIL Executive:

- Receives PLEXIL plans from the Planner or User.
- Sends plan execution status to the Planner or User.
- Sends commands to the System.
- Receives state information from the System.



Implementation Details

- The Plexil Executive is written in C++.
- The Plexil Executive interfaces to systems via an Interface Adaptor (C++) framework. Underlying communication can be provided by:
  - Sockets / TCP
  - CORBA
  - CMU's *IPC*

Plexil Syntax

Plexil has two user syntaxes:

- (standard) Plexil (`.ple` files)
- PlexiLisp (`.pli` files)

These user syntaxes produce PLEXIL XML (`.plx` files).

The Plexil Executive executes Plexil XML.

- Plexil and Plexilisp translate to XML.
- Standard Plexil is written in Java.
- Plexilisp is written in Emacs Lisp.
- Neither is needed for plan execution.

Recent Applications

PLEXIL has been used for:

- K10 Rover control
- Earth science drilling executive
- Rotorcraft system architecture (SIRCA)
- Execution of Procedure Representation Language
  - International Space Station procedures (simulation)
  - Habitat Demonstration Unit procedures (field tested)

Future Applications

Robotics

Unmanned vehicles and habitats

Systems and simulations

Intelligent software agents

How to Get Plexil

To download source and binary for Linux or Mac:

- Visit http://plexil.sourceforge.net
- Click on Download tab

To download source only:

```
svn co https://plexil.svn.sourceforge.net/svnroot/plexil/trunk plexil
```

To build, Plexil requires:

- GNU C/C++ 3.3.3 or newer
- Java SDK 1.4 or newer
- Jam build tool 2.5 or newer

Plexil Community

Visit the PLEXIL wiki at:

http://plexil.wiki.sourceforge.net

For help, problems, or bugs, send mail to:

plexil-support@lists.sourceforge.net

For general Plexil discussion:

plexil-discussion@lists.sourceforge.net

To join plexil-discussion, visit

Visit http://plexil.sourceforge.net

Click on the mailing lists tab.

Outline

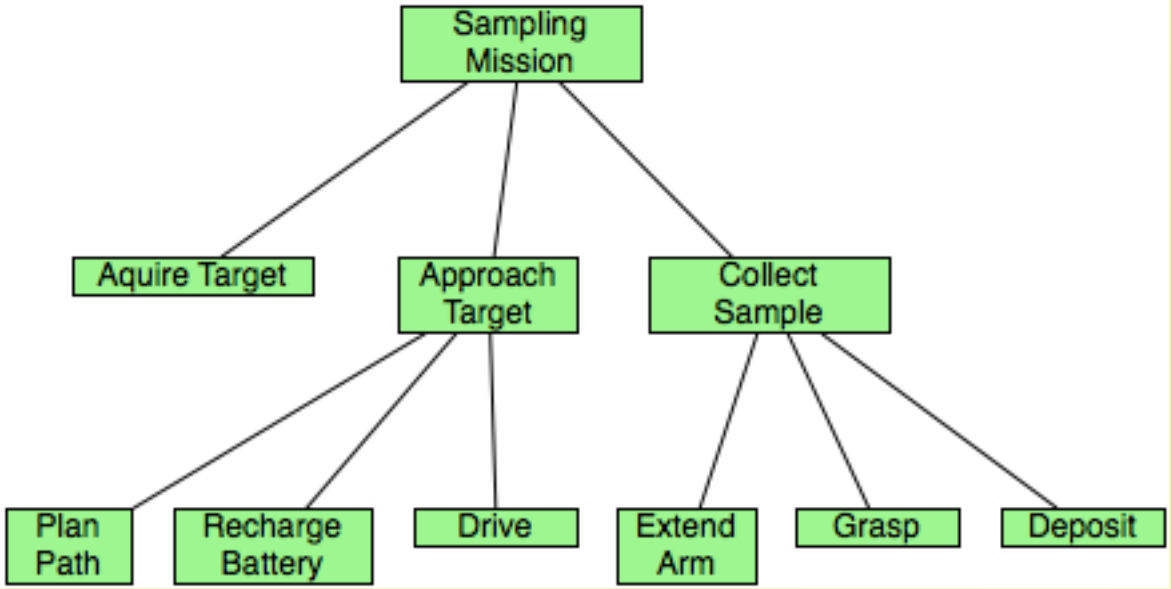
Background

Plexil plan structure

Plexil execution

Plan Topology

A Plexil plan is a tree of *actions*:



Action Types

- Sequences (several varieties)
- Concurrence
- If-Then-Else
- While and For loops
- Inter-executive message passing
- Core PLEXIL (*nodes*)
  - Interior nodes:
    - List
  - Leaf nodes:
    - Assignment
    - Command
    - Empty
    - Library call
    - Update

Core PLEXIL

Subset of PLEXIL into which all actions are translated

- Consists of *nodes*

For now, only three node types are important:

- List – contains other nodes
- Assignment – assigns a value to a variable
- Command – controls the System

Gate Conditions

Each node has four *gate conditions:*

- Start
- End
- Repeat
- Skip

These control *when* a node executes.

Check Conditions

Each node has three *check conditions*:

- Pre
- Post
- Invariant

If any of these fail, the node fails.

Start vs Pre

Start condition x < 10

Node starts if x < 10 and parent has started.

Pre condition x < 10

If node starts, and x >= 10, then node immediately fails, without executing its body.

Outline

Background

Plexil plan structure

Plexil execution

Plans are State Machines

A Plexil plan is a state machine.

A plan has *plan state* or *internal state*.

The world has *world state* or *external state*.

The plan responds to changes in world state.

The plan responds to *one* change at a time.

Plan State

A plan has two kinds of state:

- Node state
- Variable state

Node States

Inactive
Waiting
Executing
Finishing
Iteration_Ended
Failing
Finished

Node States

- Inactive
- Waiting
- Executing
- Finishing
- Iteration_Ended
- Failing
- Finished - outcome:
  - Skipped
  - Success
  - Failure

Node States

- Inactive
- Waiting
- Executing
- Finishing
- Iteration_Ended
- Failing
- Finished - outcome:
  - Skipped
  - Success
  - Failure – failure type:
    - Pre condition failed
    - Post condition failed
    - Invariant condition failed
    - Parent failed

World State

The world has two kinds of state:

- Variable state
- Command execution state

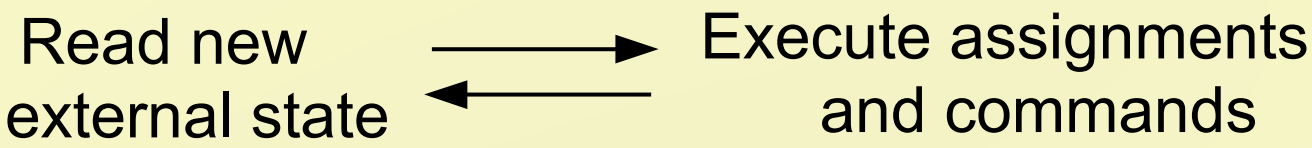
Events are State Changes

- Lookup(X)
  - Returns the value of X in the current state.
  - In certain contexts (gate conditions), subscribes the plan to changes in X.

Main Loop

Plexil's main loop:

- Read the new external state.
- Execute assignments and commands.



Nodes are State Machines

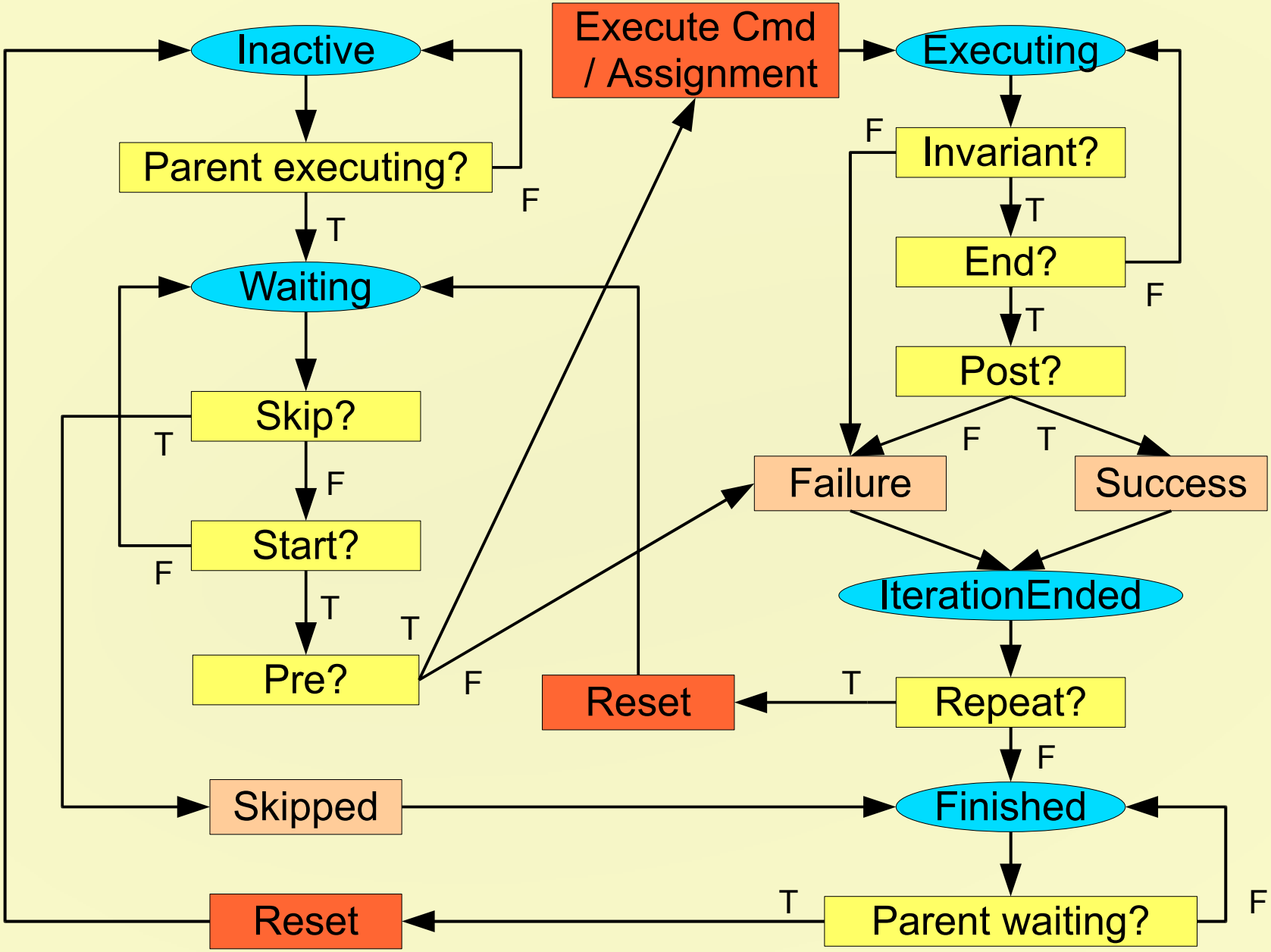
Each node is a state machine.

All nodes execute in parallel.

All nodes execute synchronously.

The plan state machine is the synchronous parallel combination of the node state machines.

Node State Machine



Starting and Stopping

Execution starts with all nodes INACTIVE.

Execution stops when no node can transition.

Execution Loops

Plan execution has three nested loops:

```
Loop for each external state change {
  Read external state;
  Loop until no node can transition {
    Loop for all nodes in parallel {
      Process one transition;
    }
  }
}
```

Example Plan

```
SafeDrive:  {
  Integer pictures = 0;
  NodeList:
  Loop:  {
     RepeatCondition: ! LookupOnChange(WheelStuck) &&
                              pictures < 10;
     NodeList:
     OneMeter:  {
        Command: Drive(1);
      }

      TakePic:  {
        StartCondition: OneMeter.state == FINISHED;
        Command: TakePicture();
      }

      Counter:  {
        Assignment: pictures = pictures + 1;
      }
    }
}
```

Example Plan Execution

SafeDrive	Loop	OneMeter	TakePic	Counter	pict	WS
Inactive	Inactive	Inactive	Inactive	Inactive	----	F
Waiting	Inactive	Inactive	Inactive	Inactive	----	F
Waiting	Waiting	Inactive	Inactive	Inactive	0	F
Executing	Executing	Inactive	Inactive	Inactive	0	F
Executing	Executing	Waiting	Waiting	Waiting	0	F
Executing	Executing	Executing	Waiting	Executing	1	F
Executing	Executing	IterE	Waiting	IterE	1	F
Executing	Executing	Finished	Waiting	Finished	1	F
Executing	Executing	Finished	Executing	Finished	1	F
Executing	Executing	Finished	IterE	Finished	1	F
Executing	Executing	Finished	Finished	Finished	1	F
Executing	IterE	Finished	Finished	Finished	1	F
Executing	Waiting	Finished	Finished	Finished	1	F
Executing	Executing	Inactive	Inactive	Inactive	1	F
...						
Executing	IterE	Finished	Finished	Finished	10	F
Executing	Finished	Finished	Finished	Finished	10	F
IterE	Finished	Finished	Finished	Finished	10	F
Finished	Finished	Finished	Finished	Finished	10	F

Example Plan Execution

SafeDrive	Loop	OneMeter	TakePic	Counter	pict	WS
Inactive	Inactive	Inactive	Inactive	Inactive	----	F
Waiting	Inactive	Inactive	Inactive	Inactive	----	F
Waiting	Waiting	Inactive	Inactive	Inactive	0	F
Executing	Executing	Inactive	Inactive	Inactive	0	F
Executing	Executing	Waiting	Waiting	Waiting	0	F
Executing	Executing	Executing	Waiting	Executing	1	F
Executing	Executing	IterE	Waiting	IterE	1	F
Executing	Executing	Finished	Waiting	Finished	1	F
Executing	Executing	Finished	Executing	Finished	1	F
Executing	Executing	Finished	IterE	Finished	1	T
Executing	Executing	Finished	Finished	Finished	1	T
Executing	IterE	Finished	Finished	Finished	1	T
Executing	Finished	Finished	Finished	Finished	1	T
IterE	Finished	Finished	Finished	Finished	1	T
Finished	Finished	Finished	Finished	Finished	1	T