A Survey of **7 Intelligent, Reactive (++) Agent Architectures**

Christopher H. Weber
Bryce AI Research Group
christopherweber@hotmail.com

Introduction

I am a Beliefs, Desires, Intentions (BDI) agent.

- Beliefs – The World:
  - There are 7 agent architectures
  - 35 minutes maximum available
- Desires – I have two goals:
  - Present the 7 architectures (Priority 1)
  - Find the best agent architecture for MABLE (Priority 2)
- Intentions – Review the 7 systems in step-wise, chronological fashion in this format:
  - An “easy” explanation of the system
  - A diagram of its structure/architecture
    - Describe the basics of the architecture
  - An example of the language/operators
    - Describe the basics of the language

MABLE in brief

MABLE is a learning agent, but it can become much like a reactive agent after being fully prepared.

Performance Measure criterion for selecting the best architecture for MABLE:

- Deliberative (perhaps as an add-on) and Reactive
- Expressive Description Language
- Verification and Validation
- Meta-Control and Reflectivity
- Interfaces nicely with any problem/application
- Mixed-initiative capabilities

The 7 Papers

1. 1987 **– RAPS – An Investigation into Reactive Planning in Complex Domains**, Firby
2. 1989 **– PRS – Decision-Making in an Embedded Reasoning System**, Georgeff & Ingrand
3. 1994 **– CIRCA – World Modeling for the Dynamic Construction of Real-Time Control Plans**, Musliner et. al.
4. 1997 **– 3T – Experiences with an Architecture for Intelligent, Reactive Agents**, Bonasso, Firby, et. al.
5. 2002 **– RMPL/TITAN – Model-Based Programming of Intelligent Embedded Systems and Robotic Space Explorers**, Williams et al.
6. 2004 **– SPARK – The SPARK (SRI Procedural Agent Realization Kit) Agent Framework**, Morley & Myers
7. 2006 **– UE/PLEXIL – Universal Executive and PLEXIL: Engine and Language for Robust Spacecraft Control and Operations**, Verma et. al.

RAPS -1987

•Systems that build or change their plans in response to the shifting situations at execution time are called reactive planners.

•A model of purely reactive planning is proposed based on the concept of reactive action packages (RAPs). A RAP is an independent entity pursuing some goal in competition with many others at execution time.

•A RAP is centered around procedural reasoning, provides language features for deductive state inference (memory-rules), and is interpreted by a deductive problem-solving reactive planner.

•RAPS provides constructs for resource locks for thread synchronization at the systems lowest level. The semantics are not specified in the paper.

•The paper does not specify the extent of RAP syntax and operators.

•Applications: none mentioned.

•Another system down the road will use RAPs in a more robust architecture.

A RAP

```
(define-rap (attach-at-site ?thesite)
  (succeed (docked ?thesite))
  (method
    (context (ferrous-hull ?thesite))
    (task-net
      (sequence
        (t1 (approach-site ?thesite))
        (t2 (magnetically-attach ?thesite)
            (wait-for (docked ?thesite))))))
  (method
    (context (not (ferrous-hull ?thesite)))
    (task-net
      (sequence
        (t1 (approach-site ?thesite))
        (t2 (grip-attach ?thesite)
            (wait-for (docked ?thesite)))))))
```

The RAP Interpreter

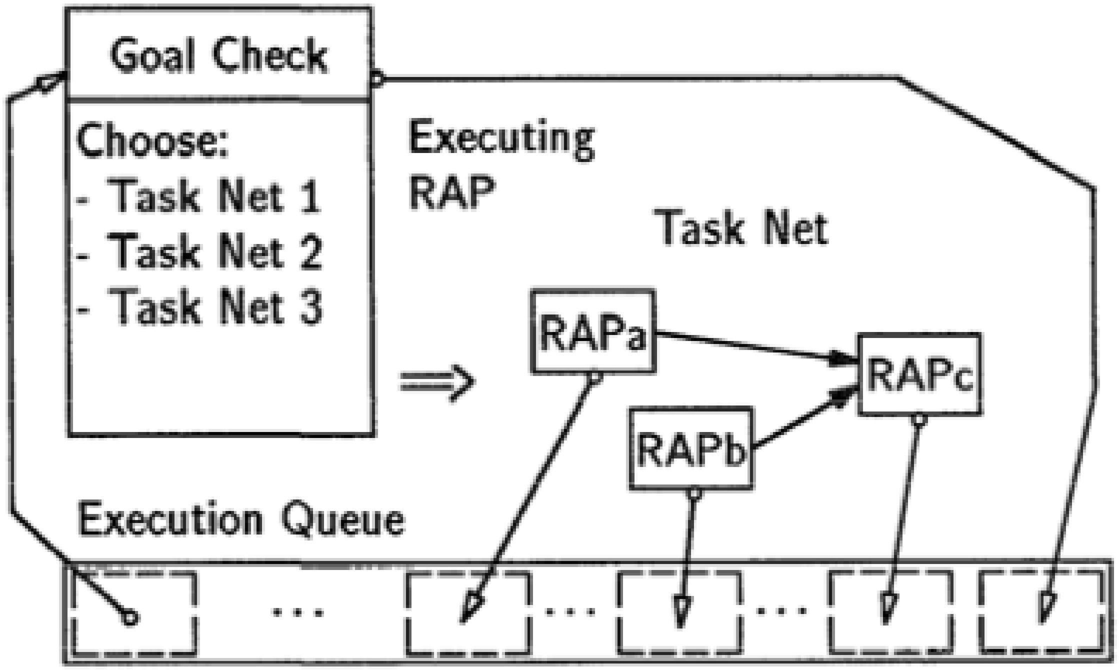


Figure 2: An Illustration of RAP Execution

RAPS System Structure

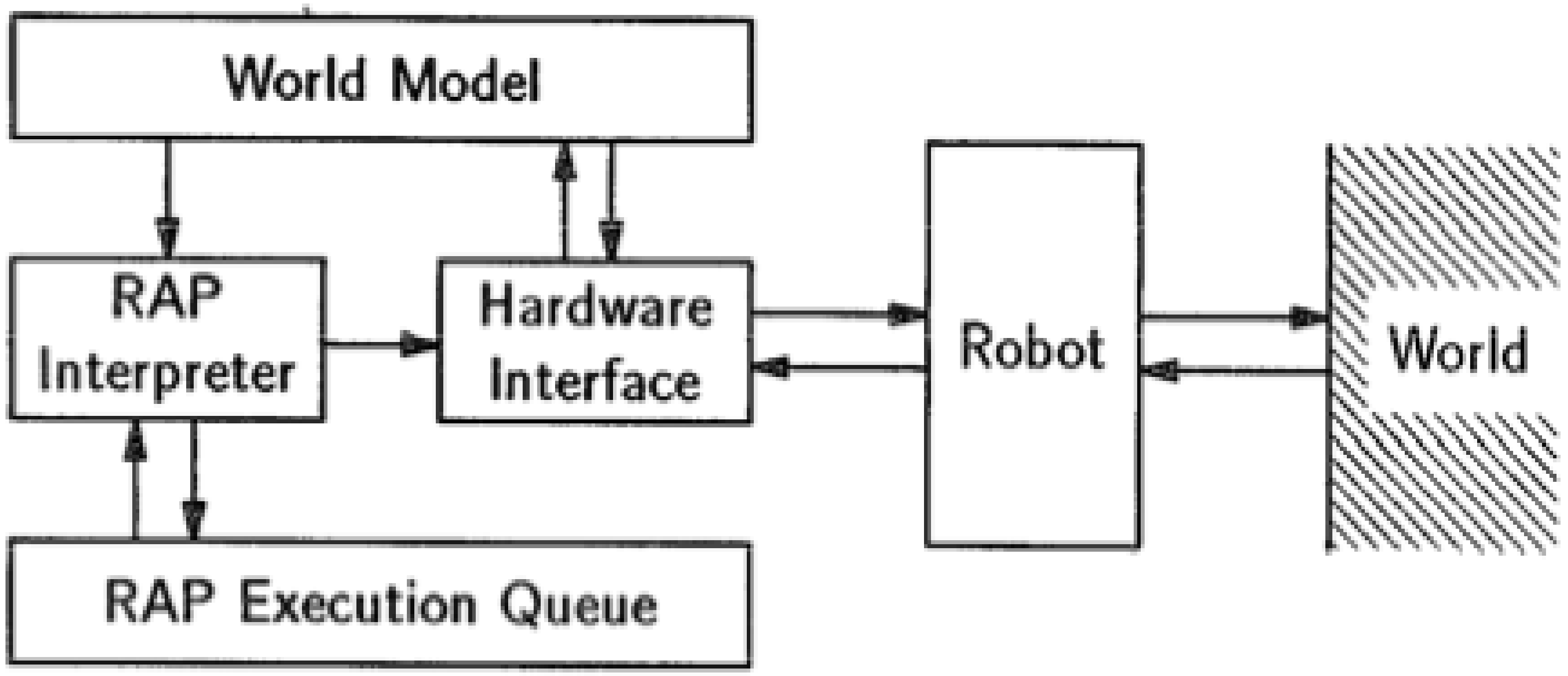


Figure 1: The RAP Execution Environment

RAPS SUMMARY

- Purely reactive planning cannot deal effectively with problems that require thinking ahead. This greatly complicates preventing failures and working on multiple goals. Loops can occur, and resources can be exhausted.

- Thus, a strategic lookahead planner is useful in extending this system (putting constraints on RAP behavior, execution ordering, etc.).

PRS -1989

- A classic DBI-based procedural reasoning system that can reason about alternative courses of action and plan in a continuously changing environment.
- Supports goal-directed reasoning and reacts rapidly to unanticipated environmental changes.
- Decision-making capabilities are integrated with meta-level capabilities: taking account of both bounds on resources and knowledge in real-time operation.
- Like RAPS, utilizes a procedural style, invokes tasks on the beliefs about world state, and expands the task hierarchically. However, RAPs does not balance decision-making requirements against the constraints on time and info typical of complex domains.
- Meets Laffey criterion for real-time reasoning systems: high performance, guaranteed response (external event response must occur within bounded time interval), temporal reasoning capabilities, asynchronous inputs, interrupt handling, continuous operation, noisy data, and focus shifting.
- Beliefs are encoded in fopl, the system itself is written in C++.
- Applications: a large-scale simulator.

PRS Agent Architecure

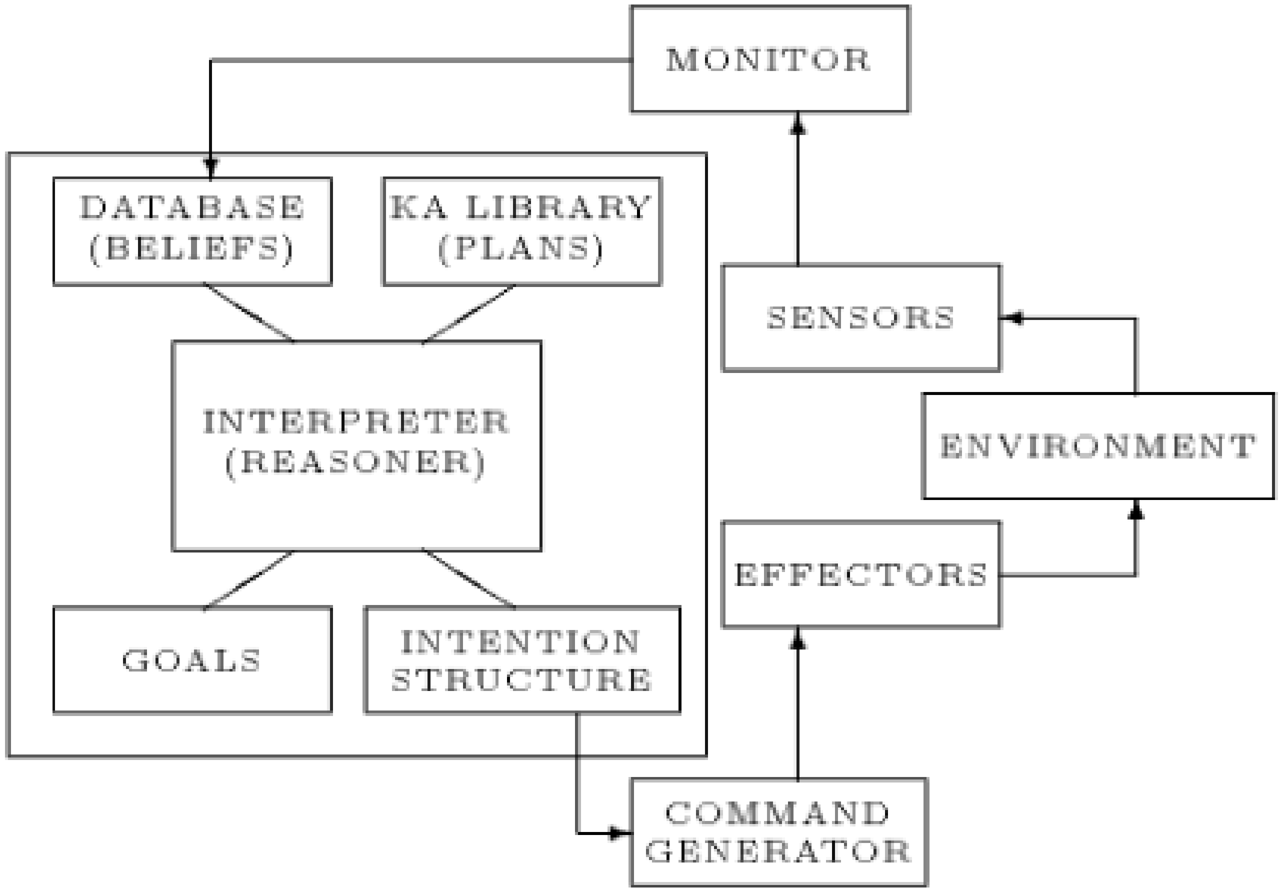


Figure 1: System Structure

Sample KA (Knowledge Area) Code

```
GOALS: (ACHIEVE cone_demo)
FACTS: (vehicle_status "True") (demo_done "False") (cone_found "False") (cone_reached "False")
(vehicle_reached "False") (vehicle_maxdist "False") (vehicle_stopped "True")

KA{
NAME: "traveled_off_road"
DOCUMENTATION: "When the vehicle is at the cone, it does some off roading"
PURPOSE: (ACHIEVE traveled_off_road)
CONTEXT: (FACT vehicle_status "True") (FACT vehicle_reached "False")
        (FACT STOPPED $STOPPED) (FACT REACHEDVEHICLE $REACHEDVEHICLE)
        (FACT OFFROAD $OFFROAD)
BODY:           (1 (EXECUTE start_behavior $OFFROAD) 2)
                (2 (EXECUTE check_behavior $STOPPED $vehicle_stopped) 3)
                (3 (FACT vehicle_stopped $value) 4) (OR
                                ((4 (TEST (== $value "False")) LOOP 2))
                        ((4 (TEST (== $value "True")) 5)
                        (5 (EXECUTE (6 (check_behavior $REACHEDVEHICLE $vehicle_reached) 6)
                        (6 (ASSERT vehicle_reached $vehicle_reached) 7)))
}

//start off road behavior
//while (not done)
//if (vehicle stopped) done = true
//if (reached vehicle) assert at vehicle
```

PRS - Planning or Not?

- The paper argues that simply choosing a course of action constitutes forming a plan.
- However, a planner is not a component of the PRS. Choosing from alternative plans is done by Meta-Level KAs, but the plans themselves are prewritten.
- By the ML KA used and its priority, weak planning is utilized.
- More complex ML KAs can examine time availability, costs and benefits and make for a more reliable system.
- If a rich set of KAs exists, planning can become quite weak even as the system robustly performs.

CIRCA -1994

- Cooperative Intelligent Real-time Control Architecture (CIRCA) executes complex AI methods to generate guaranteed real-time control plans.

- Its design is focused on:
  - flexibly achieve goals in changing environments via dynamic planning of control actions.
  - guaranteeing system safety.
  - handling limits in sensing and processing resources by allocating them intelligently.

- Worst-case timing data is employed; thus, CIRCA recognizes when its resources are not sufficient to guarantee achievement of any particular goal. In this case, it may alter its high-level plans/goals or give its best effort with no guarantees.

CIRCA -1994

- Be intelligent in real-time OR be intelligent about real-time: AI methods must meet deadlines OR AI methods reason about the real-time tasks that must meet deadlines, but AI process itself need not be so constrained.

- CIRCA works best in for domains that are:
  - Partially controllable: where the system can take actions to avoid undesirable states, but may not have control over all state changes.
  - Fully observable: the system can make adequate observations to uniquely decide which state it is in (ensures guarantees). Otherwise, guarantees are probabilistic.
  - Limited capacity: a system may not have enough capacity to handle all the demands placed on it.

- Written in C, and description language is STRIPS-like.
- Applications: Puma robot arm simulated via Deneb Robotic's IGrip system – box-packing off a conveyor belt.

CIRCA Agent Architecture

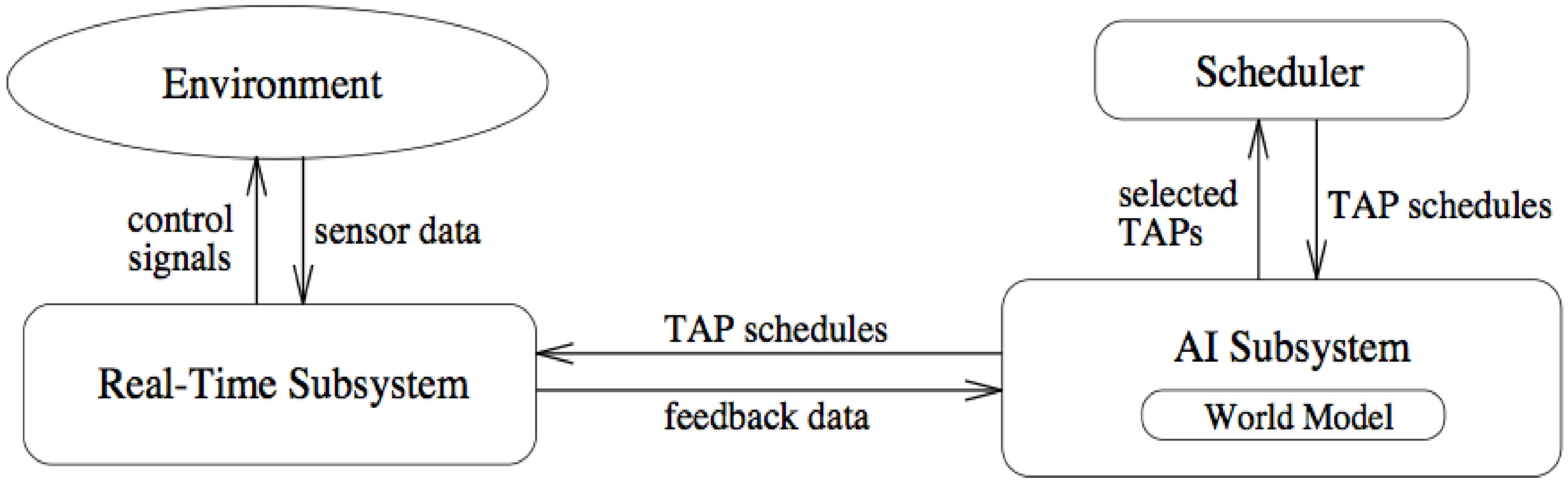
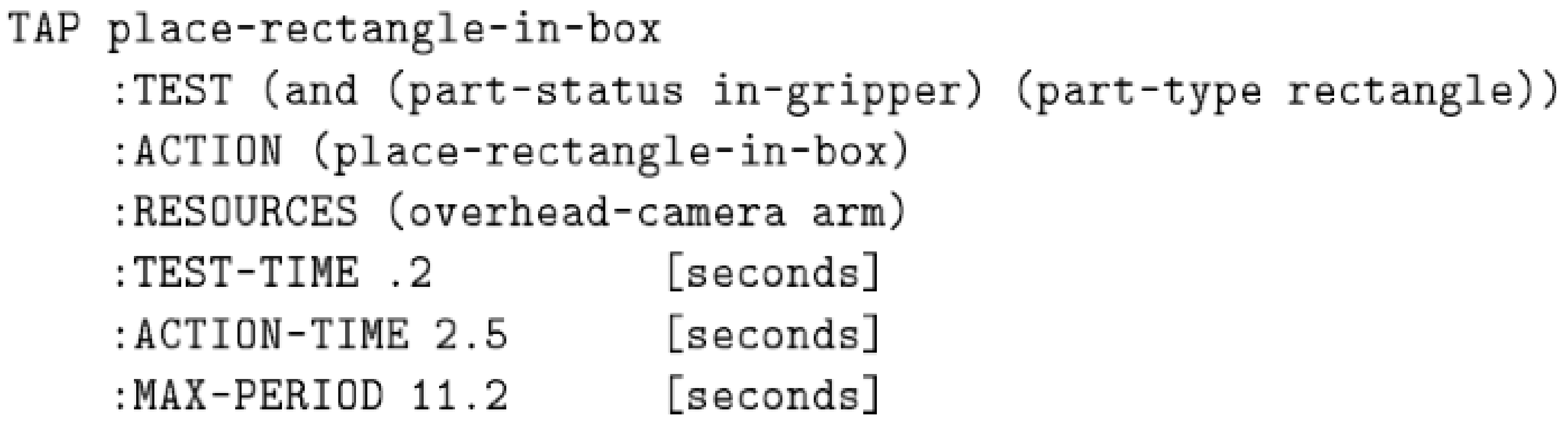


Figure 1: The Cooperative Intelligent Real-Time Control Architecture.

CIRCA - TAPs



```
TAP place-rectangle-in-box
    :TEST (and (part-status in-gripper) (part-type rectangle))
    :ACTION (place-rectangle-in-box)
    :RESOURCES (overhead-camera arm)
    :TEST-TIME .2          [seconds]
    :ACTION-TIME 2.5       [seconds]
    :MAX-PERIOD 11.2       [seconds]
```

Figure 3: An example TAP from the robot arm domain.

3T - 1997

- Agent architecture with three level of abstractions/tiers; hence, 3T. This modularity enables generalized knowledge across projects.
- Coordinates planning activities (deliberation/plan synthesis) with real-time behavior (reactivity/ situated reasoning) to deal with changing/dynamic environments.
- Avoids danger, maintains resources, and accepts human guidance.
- The decoupling of real-time execution from sequencing and planning allows the modification of sequences and plans with rebooting of that tier.
- The framework flows seamlessly across the tiers, from plan operators to continuous control loops.
- Lower level skills are written and debugged separately before integration to perform a task.

3T - 1997

- 3T includes useful software tools enabling rapid implementation and abstracts aways the need for the programmer to interface the incoming data to/from a skill.

- Uses RAPs.
- Written in LISP, C

- Applications: Robot tracking (over a 3-D terrain), WWW bot, closed ecological life support systems.

3T Agent Architecture

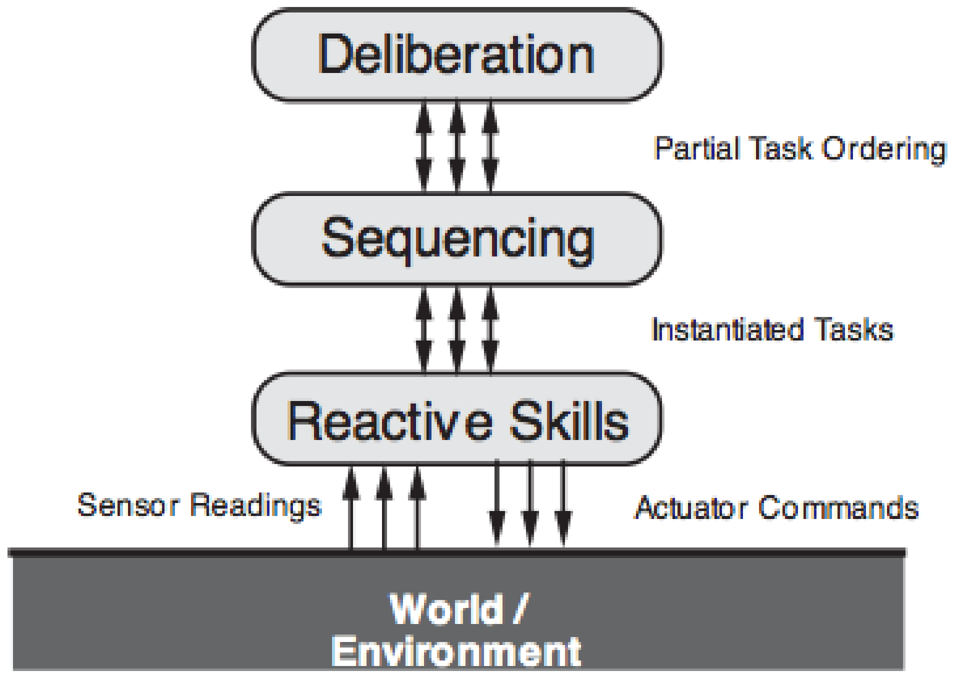


Figure 1: The 3T Intelligent Control Architecture

RMPL/TITAN - 2002

- Programming complex embedded systems involves reasoning through intricate system interactions along lengthy paths between sensors, actuators, and control processors. Resulting code generally lack modularity and robust failure-handling.
- Enter Model-Based Programming (MBP), which specifies high-level control strategies and modularizes system hardware and software. While executing a control strategy, executives track system state, find and handle faults, and perform system reconfiguration.
- Reactive Model-Based Programming Language (RMPL) provides the features of synchronous, reactive languages, with the added ability of reading and writing to state variables that are hidden within the physical plant being controlled.
- Titan executes RMPL programs using component-based declarative models of the plant to track states, analyze novel situations, and dynamically generate control sequences. Within a reactive control loop, Titan employs propositional inference to deduce the system's current and desired states, and model-based reactive planning moving the plan from current to desired state.
- Modified FSAs are extensively utilized, and the ways this system handles possible states to find a most likely state achieving real-time performance are very impressive.

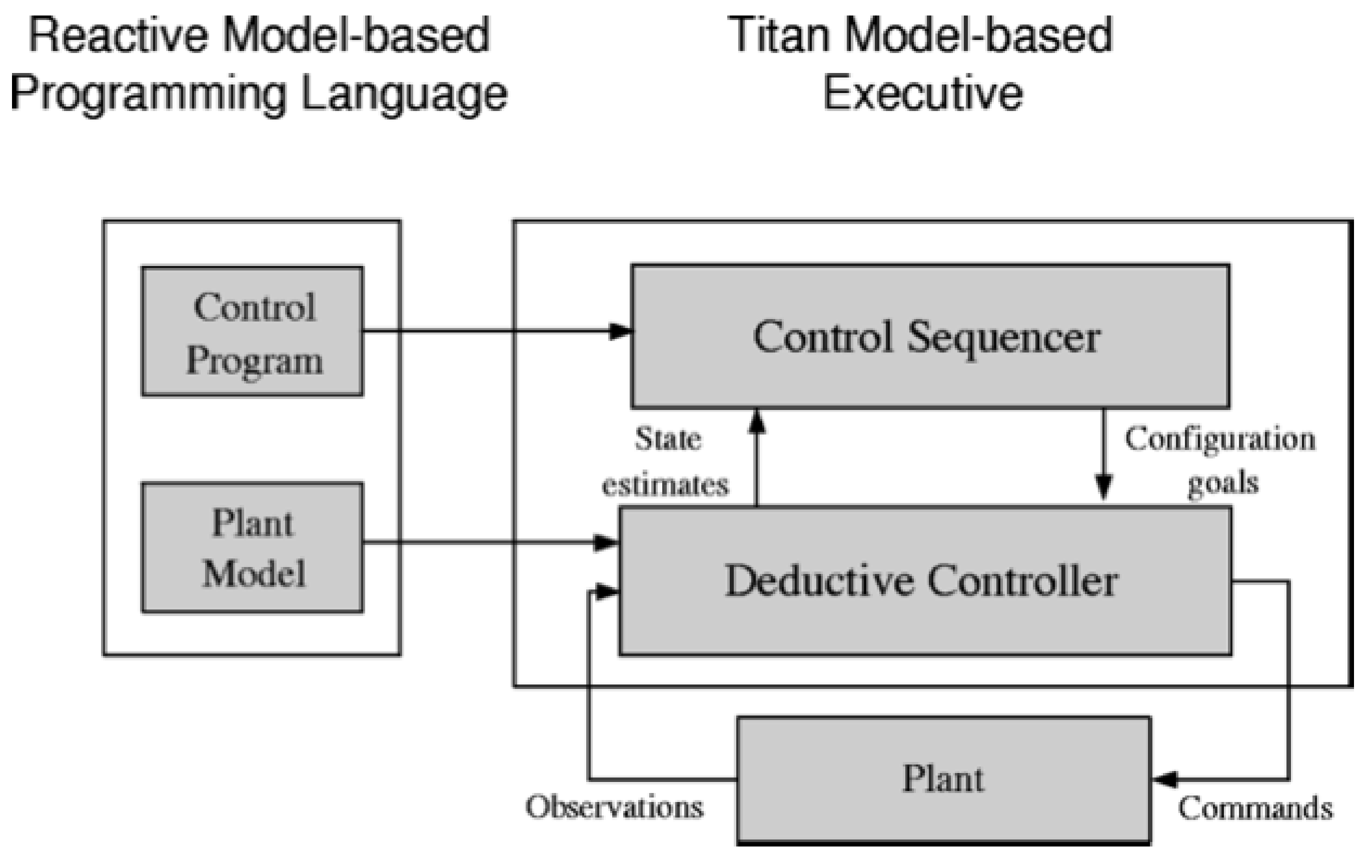
RMPL/TITAN - 2002

Not unlike RAPS, it engages in goal-directed tasking and monitoring capabilities, but covers fully synchronous programming to unify the executive with the real-time language and the deductive controller interfaces with more applications.

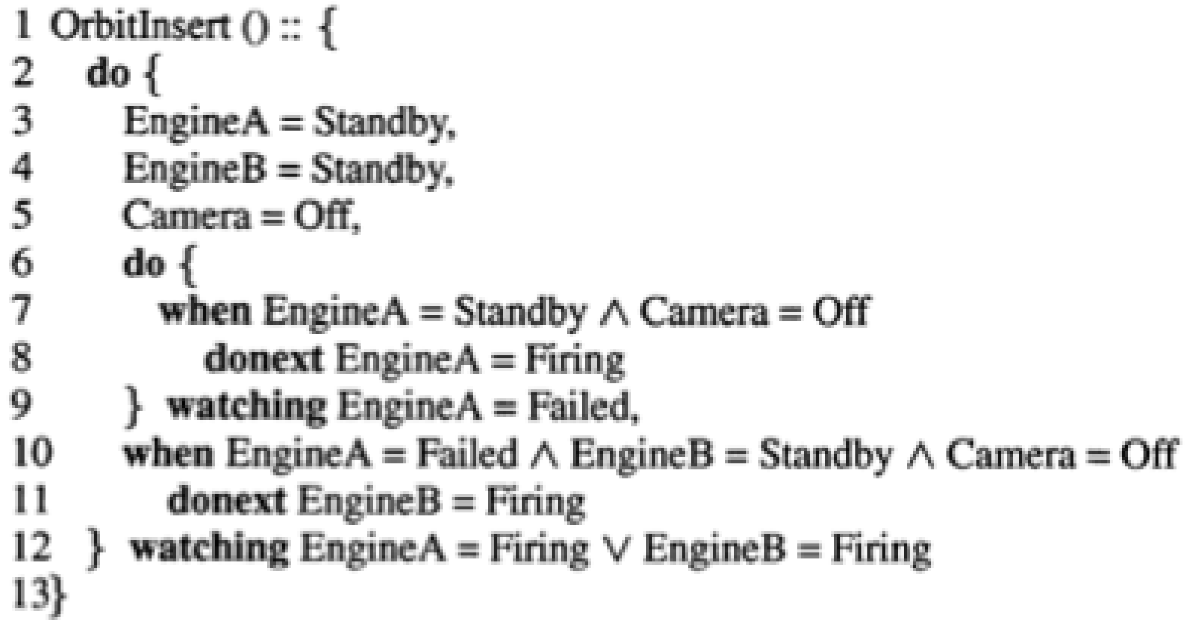
- Written in C++, LISP, plant models are written in MPL.
- uses the OpSat Optimal Constraint Satisfaction Engine.
- MBP is already extended to include fast temporal planning.

- Applications: Mission scenarios for NASA's MESSENGER, demonstrated on the MIT SPHERES spacecraft inside the International Space Station; Titan is a superset of Livingstone, which was demonstrated on the DS-1 mission and a Mars rover prototype.

RMPL/TITAN Agent Architecture



RMPL - Reactive Model-Based Programming Language



```
OrbitInsert () :: {
   do {
      EngineA = Standby,
      EngineB = Standby,
      Camera = Off,
      do {
         when EngineA = Standby ∧ Camera = Off
            donext EngineA = Firing
      }  watching EngineA = Failed,
      when EngineA = Failed ∧ EngineB = Standby ∧ Camera = Off
         donext EngineB = Firing
   }  watching EngineA = Firing ∨ EngineB = Firing
}
```

SPARK - SRI Procedural Agent Realization Kit - 2004

- Influenced by PRS, it possesses a more formal semantic model and greater emphasis on engineering issues.
- Scales to large-scale real-world applications but maintains the clean, consistent semantics of a formal agent framework. Modular construction.
- Capable as a controller in a dynamic world, applying predefined libraries to perform tasks and respond to novel events. However, rather than use various program variables to characterize world state, SPARK uses a general declarative KB.
- Its procedural language provides great flexibility and expressiveness to handle world complexity.
  - a wide range of control structures,
  - clear, well-defined formal semantics supporting reasoning techniques, procedure validation and repair.
- supports introspection/reflection about agent's knowledge and execution state, which enables system validation, effective situational awareness, and event horizon projection.

SPARK - SRI Procedural Agent Realization Kit - 2004

- meta-level reasoning capabilities for enhanced agent control.
- flexible plan execution mechanism interleaves goal-directed activity and reactivity to changes in its execution environment.
- Advisability techniques support user direct-ability.
- Can adapt behavior at run-time.
- Well-defined failure mechanisms through meta-level predicates.

- Written in Python.

- Applications: a personal assistant for a office manager; is the process representation and execution mechanism for CALO, a large AI agent system.

SPARK Agent Architecture

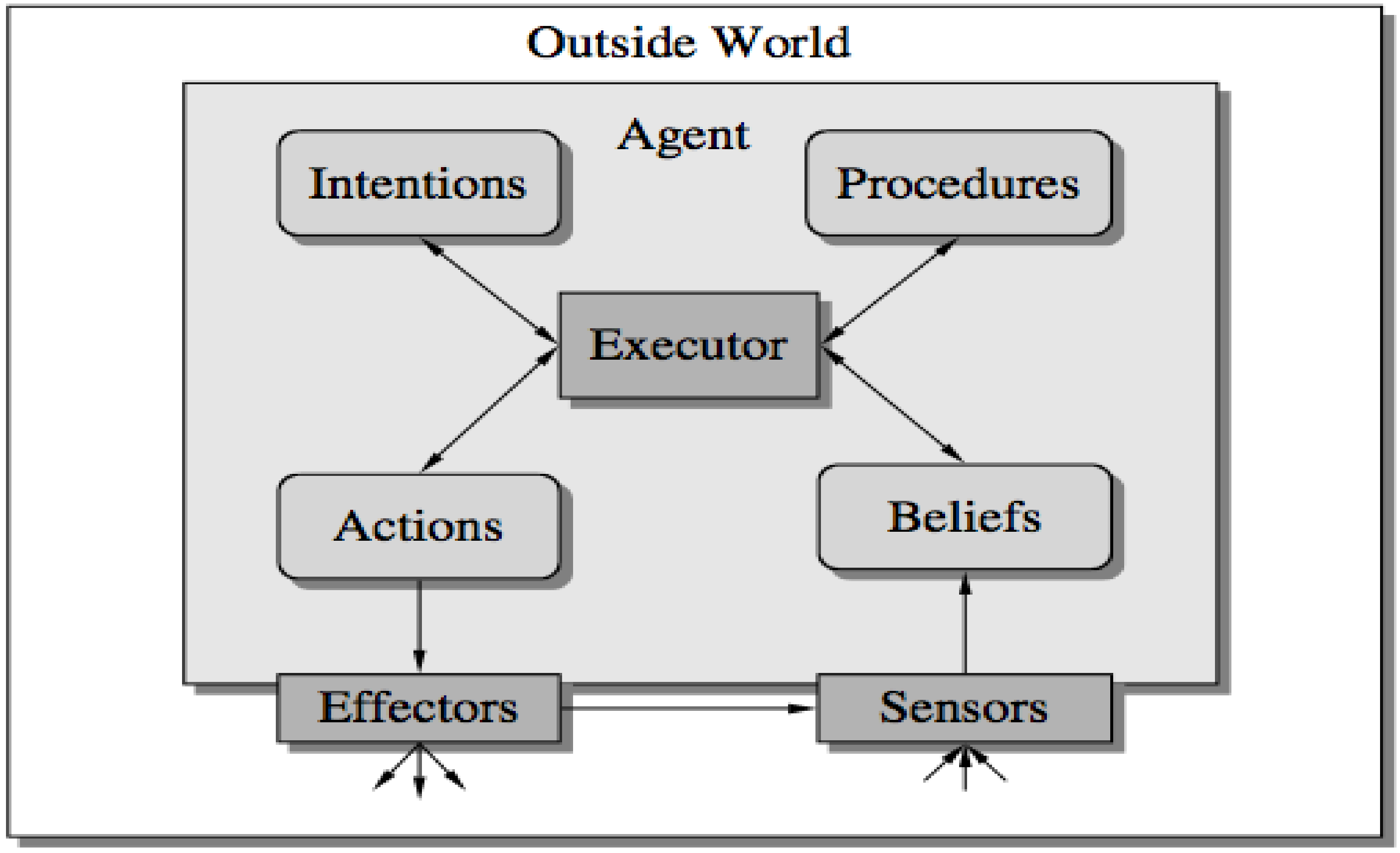
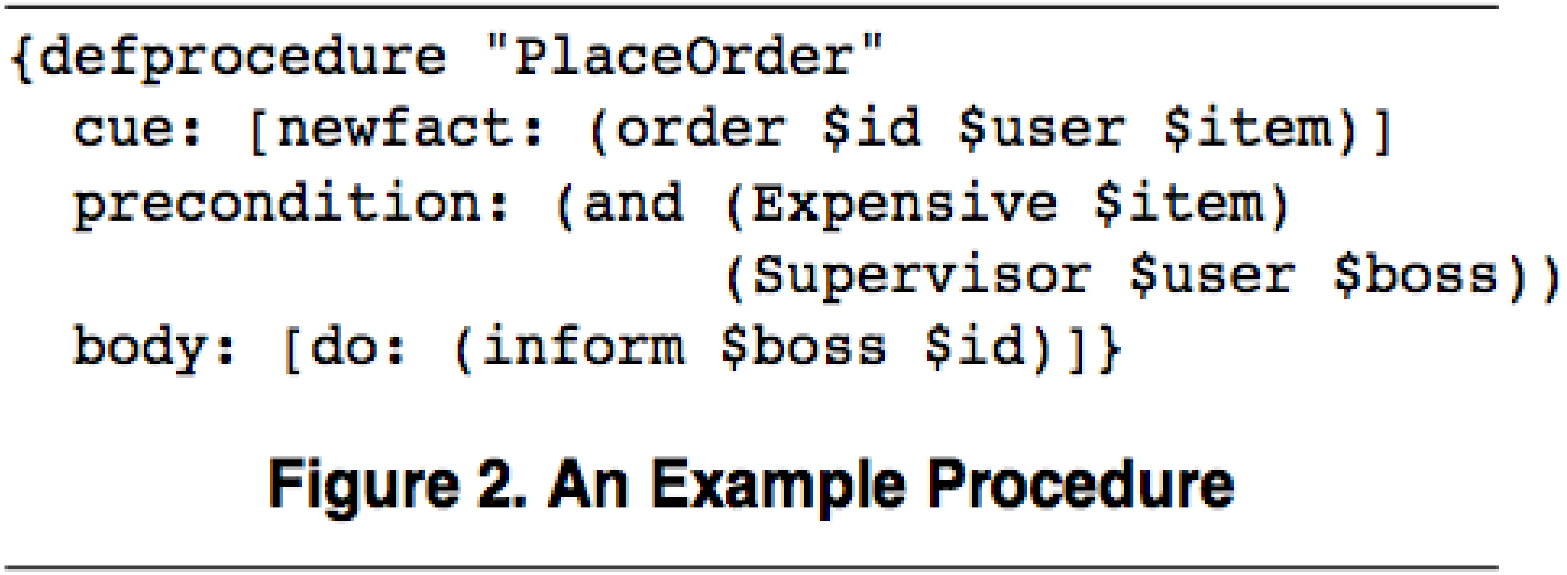


Figure 1. SPARK Agent Architecture

SPARKS Description Language



```
{defprocedure "PlaceOrder"
  cue: [newfact: (order $id $user $item)]
  precondition: (and (Expensive $item)
                     (Supervisor $user $boss))
  body: [do: (inform $boss $id)]}
```

Figure 2. An Example Procedure

UE/PLEXIL - 2007

An execution engine software tool, Universal Executive (UE), and an associated language, PLEXIL (Plan Execution Interchange Language) implemented for spacecraft operations. PLEXIL is light-weight, well-defined, predictable, verifiable, and capable of integration with high-level planners. Universal Executive executes PLEXIL plans for various platforms and environments with varying levels of autonomy and coordination with other systems including humans.

- Unlike most general frameworks, UE/PLEXIL is not monolithic and slow.
- UE is lightweight – 1.5 MB.

- Written in C++
- Applications: Manually generated command upload and execution; ground-based (external) automatic plan; plan in-situ; mixed manual and automated planning; Robotic inspection for Lunar surface operations with the SCOUT and K10 rovers; power system management on ISS.

UE/PLEXIL - 2007

- Fault-tolerant command sequence plan generation
- execution and monitoring can be generated autonomously by an internal, closed-loop process or can utilize an external planner.
- Complex computation can be done via function call.
- PLEXIL and UE simplify operations by unifying interfaces to operators and other systems.
- UE is reusable as the core executive in a variety of applications involving different underlying control software and different interactions with decision-making capabilities.
- It can operate in different ways at different times and in different contexts.
- Its PLEXIL control plans can specify what to do in all anticipated situations including how to respond to failures and problems.

UE/PLEXIL - 2007

- PLEXIL has a well-defined, expressive formal semantics for expressing plans that control complex systems, including guarantees of unambiguous responses for any given plan and situation via plan validation. Though structurally simple, it is capable of expressing complex control constructs.

- Control constructs include:
  - conditionals – if/else,
  - branches and shortcuts,
  - floating contingencies,
  - loops – for/while,
  - event-driven control,
  - time-based control.

UE/PLEXIL Example System Structures

- UE/PLEXIL SS's are massively reconfigurable according to the application.

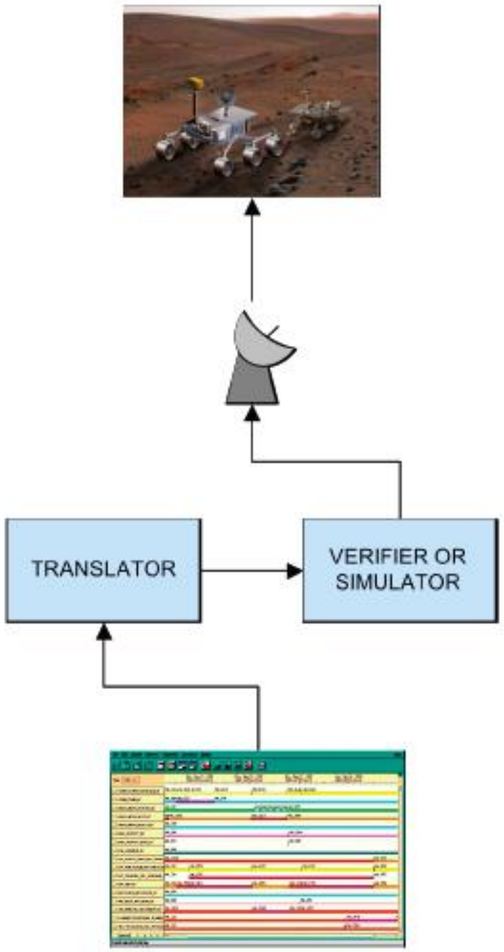


Figure 2. Automatically created external plans.

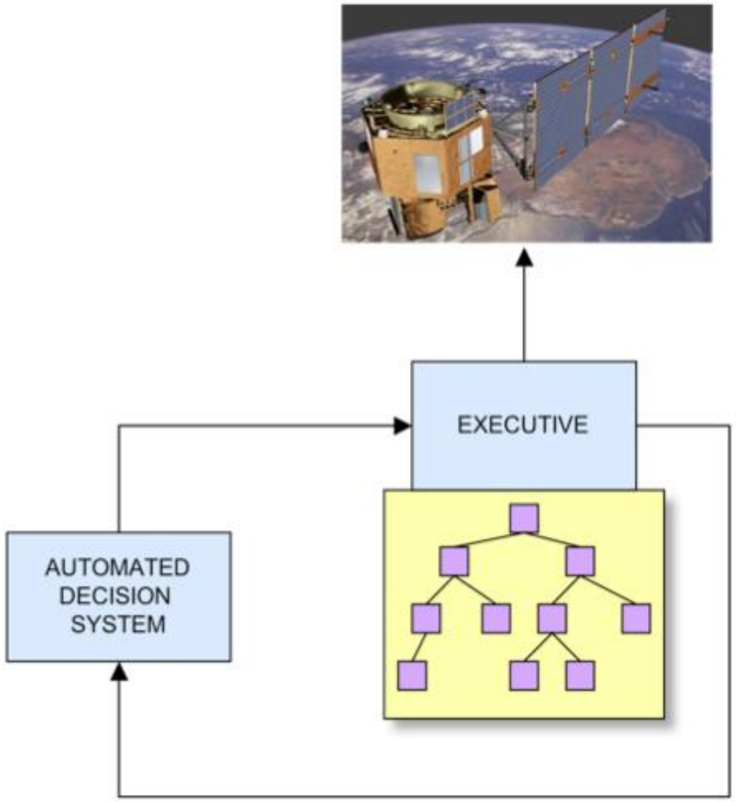


Figure 3. Plans created in-situ. *Spacecraft picture*

PLEXIL/UE - Verification and Validation

- LTSA (Labeled Transition System Analyzer) - automated formal verification of software components modeled as finite state machines – thus, PLEXIL and UE are highly conformant.

- Verification of PLEXIL plans: deadlocks, internal infinite action loops, plan conformity.

PLEXIL Example Plan

```
DriveToTarget: {
      Boolean drive_done = false, timeout = false;

      NodeList: {
      Command: rover_drive(10);

        When AbsoluteTimeWithin (10, POSITIVE_INFINITY) {
        Command: rover_stop()
        Assignment: timeout = true;
      }

      When LookupWithFrequency ("target_in_view", 10) {
        Command: rover_stop();
        Assignment: drive_done = true;
      }

      When timeout Command: take_navcam();
 When drive_done Command: take_pancam();

      While true {
        When (LookupOnChange("temperature") < 0)
                Command: turn_on_heater();
        When (LookupOnChange("temperature") > 10)
                Command: turn_off_heater();
      }
}
```

Final Thoughts

It was often a bit unclear how the agents in their environment were accurately maintaining their world model. In a dynamic uncertain environment, things are changing all the time. Sensors and low-level tasks assigned to those sensors register/record changes. The changes are placed on some form of queue.

The executor must pull off all changes in the queue and modify the world model before selecting the next task to perform. As that task performs, it changes the world model as well. And the sensors and their associated tasks are still recording all world changes.

So, then after every step of the task there must be a world update, an evaluation to see if that task is still the best choice to be performed, and a possible undo operation.

Whether or not many of these agents did this (well) was unclear from their representative paper. Conversely, at least one of these systems dispensed with maintaining a world model entirely through modeling all possible likely states given the plant model pre-run-time (TITAN).