An Agent Based Framework for Adaptive Control and Decision Making of Autonomous Vehicles

Nicholas K. Lincoln*, Sandor M. Veres *
Louise Dennis, Michael Fisher, Alexei Lisitsa**

**School of Engineering Sciences, University of Southampton, UK*
e-mail:s.m.veres@soton.ac.uk
***Department of Computer Science, University of Liverpool, UK*

Abstract: The paper addresses the problem of defining a theoretical physical agent framework that combines rational agent decision making with abstractions from predictions and planning of the future of the physical environment. The objective of the new framework is to *reduce complexity* of logical inference of agents controlling autonomous vehicles and robots in space exploration, deep underwater exploration, defense reconnaissance, automated manufacturing and household automation. An essential feature of the framework is automated *realtime evaluations of abstractions on the effects of future actions*. Comparison is made with hybrid automaton based solutions in terms of computational complexity.

Keywords: Control of hybrid systems, intelligent physical agents, autonomous control.

1. INTRODUCTION

Adaptive and reconfigurable control systems are often considered in isolation for a set of control inputs and outputs to be controlled, without regard to integration into an overall logic based decision making process. Autonomous vehicles, robots, utility control agents and manufacturing artificial agents often need to integrate controllers into their overall operations. In this context reasoning about priorities and consideration of system goals has often been done by intelligent agents [1,2,3,4]. This paper reconsiders the abstraction problem by agents to handle the solution of complex reconfigurable control. Following a brief introduction of intelligent physical agents for the engineering reader, the problem this paper addresses is presented.

It is widely believed that agent systems are most appropriately described by the intentional stance, wherein the agent is an entity subject to anthropomorphism [1]. Whilst, to date, there is no panacea for agent theory, significant contributions have been made relating to what properties an agent should have and how these properties should be formally represented and reasoned about [1,2,5,6,3]. Application areas for agent systems have been defined within the realms of software, industry and autonomous control systems [1,3,7]. Agent architectures are divisible into three groups: reactive, multi-layered and deliberative. Reactive architectures operate through a mapping of sensor data or current state to action, in a stimulus-response fashion, as exhibited by Brooks subsumption architecture [8]. Subsumption architectures present intelligence as an emergent property resulting from a hierarchy of finite state machines such that there is a behavioural response resulting from goal oriented desires; there is no form of planning. Such simplistic architectures are advantageous in dynamic environments, though in addition to the difficulty of engineering a set of behaviours to achieve a particular task, undesired behaviours also evolve. Moreover, omitting the ability for such agents to reason eliminates the possibilities of planning and learning: both highly desirable proactive traits for agents [9,10].

A logical framework wherein beliefs, desires and intentions (BDI) are primitive attitudes, as formulated by Rao and Georgeff, has proven to be the most popular: the three BDI primitives are highly cited within literature and used within numerous agent programming languages including the logic based PRS, Jason and 3APL, as well as the Java based implementations of JADE, Jadex and JACK [2,5,6].

Systems aiming to combine the timely nature of reactive architectures and the mathematical rigor of deliberative architectures are termed as hybrid, or layered architectures. Layered systems, as their name would suggest, involve a hierarchy of interacting subsystem layers; these layers may be horizontal or vertical. Complexities of information bottlenecks and the need for internal mediation within horizontal architectures are partially alleviated within vertical architectures, though these structures do not provide for fault tolerance. Both horizontal and vertical architectures have been implemented within Touring Machines and InterRRaP respectively [11].

Possibly the most widely known BDI implementation is PRS [4, 12]. PRS is a situated real time reasoning system that has been applied to the handling of space shuttle malfunctions, threat assessment and the control of autonomous robots. Within the PRS framework a knowledge database, detailing how to achieve particular goals or react to certain situations, is interfaced by an interpreter. Conceptually this is similar to a horizontally layered Touring machine, though notions of intention are contained and consequently place PRS as a deliberative BDI model. Whilst PRS is certainly a notable and proven methodology, the knowledge database is fixed

and invoked upon perceived condition triggers for the particular knowledge item of relevance. Here we propose the interleaving of physical stances within the intentional stances used to abstract and reason about the system in order to enrich the reasoning process.

This paper addresses the problem of defining a theoretical physical agent framework that combines rational agent decision making with abstractions from predictions and planning of the future of the physical environment. The framework can be an extension of BDI agent architectures. The paper argues that logic based decision making is more complex, or indeed correct decisions are impossible to achieve, without the agent simulating (predicting) the near future. An agent framework, permitting automated *timely* evaluation of abstractions on the effects of future actions, is presented. Programming of abstractions and a unified system ontology is facilitated by natural language programming (NLP) in sEnglish.

2. A SIMPLE MOTIVATING EXAMPLE

Assume that a point mass robot *A* is navigating through a 2D environment. It is able to sense its position relative to a target location but does not have a map of the environment (though it can build a map of the environment where it has already passed trough). It is also assumed that *A* is able to scan a range map ahead of it within a 90° angle and *A* has a fixed set of manoeuvres, *M*, it can make at any time. Now the question is as follows: is it sufficient to choose the next manoeuvre of *A*, based on the current situation of the agent by dividing the state space of the current kinetic state *x* of *A* into a set of *abstractions* and taking into account the shape of the current range map *R* to define a safe next manoeuvre function $N(x,R)\in M$? This approach is characterised by the following:
(1) Ability to plan ahead in time, to only move into a position from where there is a guaranteed manoeuvre to proceed further, without being trapped on the basis of the available range map *R*.
(2) It can be approximated from a fixed set of abstractions, for *x* and *R*, to decide on the next manoeuvre to be executed.
The question is whether such pre-computed planning is sufficient for robot control? We argue in this paper that for most practical applications this is not applicable and this is only useful in ideal situations where vehicle dynamics does not change. In most practical systems there are *state-space changes that render pre-planned decision making inapplicable*. These include actuator degradation and external disturbances affecting the manoeuvres. A possible solution may be to *extend the discretisation principle* of [13,14]: detect the actuator degradation *D*, or disturbance *d*, abstract it into propositions and define an extended manoeuvre selection function $N(x,R,D,d)\in M$. This appears feasible, though this has now become a function of 4 arguments and the number of cases to consider can be very large. The number of different cases we discriminate for *x,R,D,d* shall be called the (logical) *complexity of pre-planned decision making on the next manoeuvre*. Even if we discriminate for only a low number of cases for each of *x,R,D,d*, e.g. 10 each, one will need 10,000 cases to pre-evaluate and store for decision making.

There is an alternative to this type of pre-computation: we may equip the agent with the ability of online optimisation of its required inputs over a future time horizon that takes into account *x, R, D* and *d*. This scheme is generally known as adaptive *model predictive control.* We argue predictive control substantially reduces the complexity of the rational agent's decision making about an essentially continuous time process. In this paper we introduce a family of agent architectures for efficient use of predictive control methods instead of discretizing the state space of the agent as in [13,14] that can lead to large number of cases or logic rules to consider. However the benefits of the new agent architecture moves much more beyond the use of predictive control. We introduce and agent family that can handle adaptive MPC and performs realtime decision making via abstractions and logic:
(1) Execution of feedback loops is delegated to processes or threads on what we term here as a *physical engine*
(2) An *abstraction process* provides symbolic inputs for logic based decision making within a *rational engine*
(3) *Simulations of the future* by the agent to explore/try and solutions using its *continuous engine*.
(4) Potential for *formal verifiability* for safety of operations
(5) *Abstraction programming* via the use of natural language programming (NLP) in sEnglish.

The point mass robot example in this section has only been used to motivate the need for timely predictive control computations instead of trying to construct abstractions of cases in state space that may result in inefficient use of computational resources. The remaining features of the architecture family proposed in this paper serve a similar purpose: making the agent adaptable and fast in an unknown unstructured environment.

Publications in [15,13,14] aim to bridge the gap between continuous dynamics and discrete logic by synthesizing controllers from temporal logic formulae. The dynamical models are fixed and all discrete transitions precomputed. The motivating example given in this section highlights the need for dynamical adaptivity due to actuator changes and disturbances. The controller synthesis schemes of these publications can suffer from uncertainties and computational complexity to similar degree as was indicated within the presented example.

3. A FRAMEWORK FOR UNSTRUCTURED ENVIRONMENTS

A physical agent system (PAS) is a 3-tuple consisting of the agent, the environment in which the agent exists and a coupling between the agent and environment.

Definition A *physical agent system*, α, is defined by $\alpha = \langle A|E|C\rangle$ where A is a set of agents, $A = \{A^i | i \in S\}$, E is a set of environment objects, $E = \{E^i | i \in K\}$, and C is a set of couplings $C \subseteq \{\Gamma\langle a,b\rangle | a,b \in A \cup E\}$ between any members of the joint set $A \cup E$. Couplings between members of A are called *communications*, $\boldsymbol{C}$, and couplings between members of E are called *physical interactions*, $\boldsymbol{P}$.

Couplings between members of A and E are called *action and sensing*, Æ, in the environment.
A coupling Γ is an abstract object at this stage of our discussion that will be specified later for its types, attribute and models (see Section 4). In this paper we are interested in physical agents that are not pure software agents but ones that have a set of environmental objects associated with them, referred to as the "body" of the agents. Strictly speaking a body is a set of environmental objects associated with a set of agents.

Definition The *body association* of agents is a map, β, defined by splitting A and E into a disjoint sets of agents and sets of environmental objects, respectively:
$A = \cup A_k$, $E = \cup E_j$,
and allocating to each A_k a component of E : $A_k \overset{\beta}{\rightarrow} E_j$.

In exceptional circumstances the $E = \cup E_k$ may be allowed not to be disjoint, meaning that agent teams can "share some body parts", but this is not the norm. For logical clarity one should normally organize teams of agents, such that $E = \cup E_k$ is a disjoint union.

In the physical agent definition, that we introduce below, we separate computational units for
(1) Receiving signals from sensors and producing signals for actuators and forming symbolic first-order logic expressions (abstractions) for these signals and their patterns (unit Π) .
(2) For simulating continuous events of the future and the past and abstracting symbolic first-order logic expressions of implications for these hypothetical events (unit Ω).
(3) Manipulating symbolic expressions of behavior constraints, goals, past and future hypothetical events using first-order logic derivations via proof theoretic techniques (unit Σ) to decide upon actions to be taken.

Definition. A single *physical agent* is a tuple $A = \langle \Pi | \Sigma | \Omega \rangle$ consisting of the three main constituents: its physical engine, Π, its rational behavior engine, Σ, and its continuous engine, Ω. The physical engine can be a complex process or set of processes. It can contain a multitude of devices, including communication, sensor and actuator devices, as well as sensor data abstractors (SDAs), control data developers (CDDs) and feedback processes (FBPs) for realtime processing of perception and action signals of the agent. The SDAs are to symbolize events for Σ, the CDDs are to develop symbolic action into control signals for the actuators and the FBPs are needed for linking CDD and SDA pairs into realtime feedback loop executors.

Definition. A *physical engine* is a quad tuple, $\Pi = \langle \gamma | \sigma | \alpha | \Delta \rangle$, where γ is a set of communication devices, σ is a set of sensor data abstractors, α is a set of control data developers and Δ is a set of realtime feedback loop executors. For practical reasons any of σ, α, Δ may be empty but $\gamma \neq \emptyset$ that represents communication devices and as such is not allowed to be empty.

Definition. A *software agent* is a physical agent that has only communication devices present within its physical engine, i.e. all of σ, α, Δ are empty.

The rational behaviour engine, Σ, is formally defined so as to permit the accommodation of simple reactive subsumption based architectures as well as deliberative belief-desire-intention, i.e. BDI agent architectures. Whatever agent architecture is implemented, they must handle sensed event abstractions, agent action abstractions and decision making processes (DMPs). Components of DMP can be:

- Planning of movements in the physical world using motion primitives
- High level planning of goal achievement using various resolutions of world maps
- Planning to achieve goals other than movements

The following definition grasps the minimum of what the DMP must contain for a capable physical agent, without specifying the actual mechanisms of operation, i.e. no "agent architecture" is specified.

Definition. A rational behavior engine, Σ, is a tuple, $\Sigma = \langle W | M_p | M_g | C_s | G \rangle$, that contains a granulated multi-resolution and physical multi-domain symbolic world models *W*, abstract physical skills memory M_p, goal achievement memory (problem solving memory) M_g , abstract formulation of behavior constraints C_s, abstract formulation of short and long term goals *G*.

The question of modelling structures relating to the continuous world, and their role in planning and decision making, has so far been neglected. It is a fact however, that decisions can be influenced by hypothetical planning: forming a decision and subsequently planning for this decision may lead to less than ideal outcomes. This fact and that initial abstractions in σ, as prescribed by the designer of the agent, may not be sufficient to capture the essence of the changing world correctly, leads us to defining the "Continuous engine" of the physical agent.

Definition. The continuous engine, Ω, is a tuple $\Omega = \langle M | S | O | B | L \rangle$ where *M* is a set of approximate continuous models of the world, *S* is a continuous time simulator that uses analytical and empirical data based dynamic models to predict future state of the world, *O* is an optimizer that can optimize continuous time planning of actions, *B* is a Boolean evaluator of propositions in terms of σ statements and *L* is a library of useful numerical computations in terms of continuous variables.

As *W* is the primary, symbolic model of the world with relationships stored on symbols, the *M* is related to *W*. The *M* is a collection of models from various physical domains (geometric, mechanical, electrical, gravitational, heat, fluid flow, etc…) that make symbolic descriptions in *W* either more precise in numerical or in qualitative, time evolution sense. Even without precise numerics, the agent may perform a simulation using the simulation tools in *S* to see

possible qualitative outcomes in terms of σ, α, Δ abstractions. These qualitative outcomes of predictions can be used to make the right decisions and planning by Σ.

The *O* is a set of optimization tools for continuous problems. The agent can perform planning in continuous time and can set up a problem formulation. Then it searches in *O* for a suitable optimization tool. The results are formulated and used at the symbolic level of σ, α, Δ. Optimization for path planning, robust feedback control or a low complexity numerical dynamical model may all be tools that are available in *O*.

B is important since Boolean values of primitives in σ in prediction (simulation) outcomes must be inferred by some continuous computations that cannot be performed elsewhere except in the continuous engine Ω. *B* contains a Boolean valued functional, by example "will the agent body be within the allowed boundary if it carries on moving the same way for the next 10s" is an evaluation that is not done in symbolic computation but using *M*, *S* and *B*: the statement is converted into a symbolic first-order logic statement for Σ.

Finally *L* is a library of auxiliary computations with continuous quantities: for instance equation solvers for linear systems of equations, nonlinear equation solvers and generic nonlinear optimizers. Use of *L* by the agent assumes that the agent is capable of setting up problems in Σ by first abstracting them and picking a solution tool from *L*.
It is evident that Σ and Ω run hand-in-hand, i.e. the Σ frequently delegates computations to Ω, and then uses the output from Ω to continue deliberation of prescribe action.

There are many ways to implement the agent architecture described above. Ideally a single language should be used for all the components to permit multi-platform implementation but there are both practical and legacy issues that make realization biased towards software systems that have tools available and which would be far too expensive to reproduce. For instance the MATLAB family of toolboxes provides a rich set of numerical processing, optimization and simulation algorithms that can be exploited. Similarly the Java environment has numerous software components in the area of agent reasoning that may be implemented directly. The following table provides some possible options for the software platform to be used for the implementation of the agent architecture outlined.

Π platform	**Σ platform**	**Ω platform**
C	C	C
C++	C++	C++
Java	Java	Java
MATLAB	Java	MATLAB
SIMPOL	SIMPOL	SIMPOL

4. J2M IMPLEMETATION

In the following we describe a possible software implementation in terms of MATLAB+Java+MATLAB (J2M) that has the advantage of minimal programming effort due to the rich set of programming tools presently available. Within the J2M implementation, the Π and Ω platforms are developed within MATLAB, whilst the Σ platform is exclusively developed using Java. Although it is theoretically possible to form a rational engine within a MATLAB framework, since MATLAB does not allow for multi-threaded code execution, this would be a restricting factor. Also, to do so would neglect the existing frameworks which have been developed in more suitable languages. A description of the J2M will follow, within which we are assuming the implementation of a physical agent, that is one for which σ, α, Δ are not empty.

Sensor data, σ, may relate to position and velocity information given in some coordinate frame, as would be required for roving vehicles such as UAVs or AUVs. σ may also relate to other functional data types such as temperature and pressure information, extracted from relevant sensors. The way in which σ is communicated to Σ is critical, since Σ must be in the position to update W correctly based upon information received via γ. W is updated as a consequence of σ, from which Σ will evaluate the appropriate actions and this may instantiate invocations of Ω.

During the rational agent deliberation cycle, a point may be reached wherein calls to the continuous engine are required. Instances of these calls have already been entered upon; here we shall concentrate on the data flow. The continuous engine is constructed within MATLAB and consequently all functionality of this system is achieved through use of the appropriate m-files. Σ requests execution of a component from Ω, either *S*, *O*, *B* or *L*, dependent upon the solution sought by the deliberation cycle occurring within Σ. Ω needs to return abstracted first-order logic statements about the future under the conditions of some actions hypothetically being taken. Here logical statements are arrived at not solely by logic derivations but by (1) continuous time simulations (2) abstracting first-order logic statements of the form "if I do this then that happens" format. The logic statements obtained by Ω can then be used by Σ for deliberative planning and commitment to actions.

The physical engine, Π, is an element within an environment; this environment may be purely numerical, instantiated within a virtual world or a true dynamic real world environment. Regardless of construct, the physical engine is capable of extracting relevant abstract data from the inhabited world, using σ, and communicate those to the rational engine. This is not the only interaction mechanism Π has with Σ: the Σ can activate α, Δ components if it concludes that by logical inference. Within the developed implementation, the physical engine continually passes sensory information to the rational engine, which is used to update W, though the rational engine may chose to ignore information it receives.

Actions executed by Ω are conditional upon input from Σ, and in turn Σ expects some form of result. Consequently we are presented with the need for more elaborate communications protocols to ensure efficient interaction

between Σ and Ω. Messaging from Σ is of the form <m_c,c_c,r_c>, whereby m_c is a character string specifying the particular m-file to be executed, based upon the data encapsulated within c_c and expecting the number of returned evaluations to be the number specified within r_c. Return dialog from Ω to Σ is in a similar format, of the form <m_r,r_r>, wherein m_r signifies the m-file which was executed and r_r the result of the routine.

In the same way that Π acts to 'push' data to Σ, Σ prescribes action for Π: here Σ prescribes the initiation of Δ which acts directly upon α. Δ executes without additional action from Σ, using internal feedback devices, though Σ is capable of over-riding the execution of a Δ.

All engines run concurrently: Π is continuously feeding data to Σ, which prescribes action to Π and invokes intermittent communication with Ω. A schematic of the J2M construct, indicating interaction of the three agent engines, is given within Figure 1.

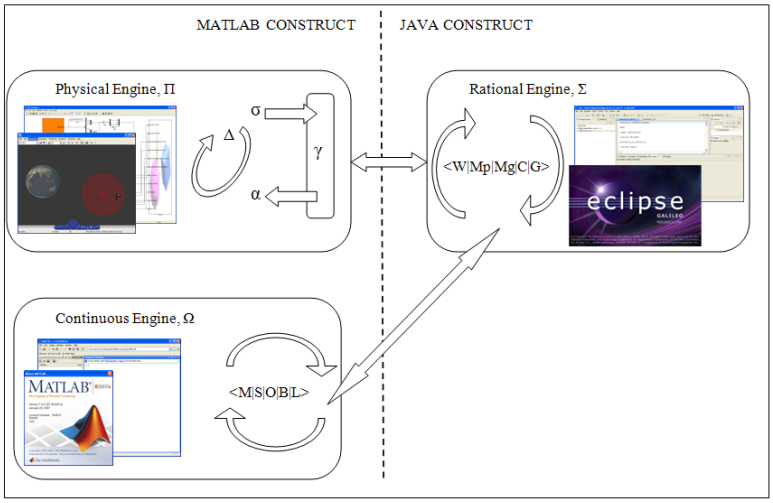

Fig 1: J2M Construct and data flow between agent engines

5. ABSTRACTIONS USING NLP

Knowledge representation within a hybrid-software agent system is an important concern since encapsulated knowledge must be uniform across the system. The process of abstraction for use within an agent system, commonly developed via anthropomorphism, may be aided using natural language programming techniques. NLP may be used not only to help link abstraction and agent deliberation, but to link the physical and continuous engines to a unified ontology. As an illustration, the following tables list some of the $\sigma, \alpha, \Delta, \gamma$ abstractions available to 6DOF spacecraft, in natural language programming (NLP) produced in sEnglish. The sentences listed are sEnglish code that unambiguously compile into MATLAB.

Some perception abstractions of a GEO satellite agent:

σ_1	Inside target region Bt.
σ_2	Moving away from the Earth.
σ_3	Being approached by debris from direction D

Some open-loop action abstractions of a GEO satellite agent:

α_1	Applying thrust vector Uv for period T.
α_2	Switching to new control configuration Cc2.
α_3	Executing plan to approach target region

Some closed-loop action abstractions of a GEO satellite agent:

Δ_1	Regulating my attitude to constant Av0.
Δ_2	Using thruster feedback control to track orbit O within cluster frame Cf.
Δ_3	Using sliding mode feedback control to regulate position Pa.

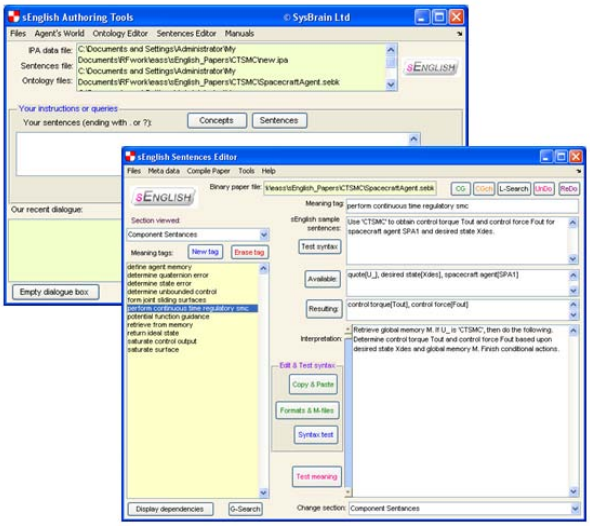

Fig 2: sEnglish Sentences Editor: Natural Language Programming for agents via structured sentences.

Some communications abstractions of a GEO satellite agent:

γ_1	Sending message M to ground control centre Co.
γ_2	Receiving message Mr from ground control centre Co.
γ_3	Receiving message Mr from agent A32.

Linking the NLP abstractions to the agent reasoning engine, developed in Java, is based upon exploiting the NLP abstractions and ontology developed within the sEnglish authoring suite[16, 17].

The implemented Java parser utilizes a similar BDI syntax to that of the Jason language [5] and is presented within [18]. For further details the interested reader is referred to the publication, here it is intended to highlight the link between such a language and the sEnglish ontology and abstractions. The simple instance of the agent using a percept update to evaluate if it is within a target region, expressed by the listed σ_1 abstraction as "inside target region Bt" may be completed by:

```
+stateinfo(L) : {True} <-
        calculate(comp_distance(L), Val),
                *result(comp_distance(L), V1),
                +bound_info(V1);

+bound_info(in) : {B inside target region Bt (out)} <-
        -bound_info(out),
        remove_shared(inside target region Bt (out)),
        assert_shared(inside target region Bt (in));
```

The above may be read as: *If it is believed that state information L has been received, then do the following: call the function 'comp_distance' with L as the argument and request a return value. Assign the returned value to V1 and use this to assess bound_info. Upon the belief addition that bound_info is 'in' and under the condition that previously it was believed 'inside target region Bt' was false, remove the belief addition of bound_info(in), remove the shared belief 'inside target region Bt (out)' and assert that the shared belief 'inside target region Bt (in)'.*

Implementation of an achieve goal, for instance the execution of an open loop control plan that links to the α_3 abstraction "Executing plan to approach target region", may be completed by:

```
+!get_to_centre[perform]:{B inside target region Bt(out)}
<-
        query(plan_to_approach_target_region(P)),
        * plan_to_approach_target_region(P),
        perf(execute(P)),
```

which may be read as: *given the achieve goal addition of 'get to centre', under the condition that it is believed that the agent is outside of the target region, then do the following: query the availability of an existing and valid plan to approach target region. Wait until the literal representing the plan exists and then execute plan to approach target region.*

These examples of linking NLP developed abstractions to the Java agent programming lead to the actual implementation of a Σ request such as *'execute plan to approach target region(P)'* by Π. Here sEnglish is exploited to enable this link: the same ontology used to provide for abstraction may be used within NLP to develop concise, sentence based, executable code using the Sentence Editor shown in Figure 2. Upon compilation, a meaningful set of linked functions are generated from structured sentences and these may be directly implemented. For this example, structured sentences were formulated relating to the required agent skills using a shared ontology: upon being initiated with plan P, the mfile 'execute_plan_to_approach_target_region', will commence implementation of plan P. Further details of sEnglish development and implementation may be found in [16, 17].

6. AN EXAPLE IMPLEMENTATION

An example implementation of the presented J2M agent framework has been developed based upon a spacecraft agent, tasked with acquiring and tracking a geostationary orbital location upon a failed orbital insertion. The spacecraft agent is availed with numerous possible control methodologies, afforded by the considerable amount of literature available on the subject of spacecraft control, inclusive of planning, adaptive and reactive control systems [16,19-24]. It is the task of the reasoning engine to select an appropriate control methodology to deal with the presented scenario, perform the appropriate control and analyse the resultant output in order to improve performance. Concurrent to the control requirements, the spacecraft agent is tasked with monitoring internal systems and reacting accordingly to any diagnostics made. Within simulation, the actuators present the agent with situations of gain reduction and propellant supply issues resulting from supply valves being stuck open/closed and a fuel-line breach.

The agent world is developed within Simulink (MATLAB), and is inclusive of disturbances resulting from Earth oblateness (triaxiality), solar radiation pressure and Luni-Solar third-body perturbations [25]. The agent is not aware of these impacting factors: within the continuous engine, Ω, the relevant physics engine for internal simulation is based upon the Hill equations, a simplification of the orbital dynamics with respect to the desired orbital location [26]. Whilst not essential, it was chosen to include a VRML world to aid visualization of the dynamical processes occurring as a result of agent action or inaction. The agent world is initialized with a state representative of a failed orbit insertion, in a circular orbit with a 10km position error from the nominal geostationary point.

Upon activation, the agent system adopts the 'achieve goal' of reaching a specified geostationary point. There are numerous methods to achieve this, involving various degrees of control complexity. Within the implementation, the agent was provided with the ability to perform model based (fuel) optimal path planning under varying actuator bounds; the output being a fuel optimal path and a set of thruster sequences to achieve this. The agent may directly implement the open loop set of thruster sequences or use a path following scheme applied to the optimal path. The agent was also availed with a sliding mode feedback controller for point tracking.

The reactive logical actions dealing with discrete thruster fault instances such as valve failure are not the prime focus here; we wish to examine the nature and potential of the continuous engine and this is highlighted primarily within the reduced thruster gain and total thruster loss scenarios. Both of these scenarios represent cases where generated plans may

become obsolete due to an inability to output a desired thrust level. Whilst path following schemes may be executed using traditional feedback control methods, the prescribed path may no longer be achievable; either in physical trace or within a temporal bound.

Online evaluation of plan suitability may be achieved via the continuous engine: here we test the validity of an existing plan within rapid simulation, based upon perceived actuator capabilities. Such testing actions may endow the agent with temporal reachability knowledge; essential in scenarios where timeliness is critical, such as avoidance manoeuvres. An example algorithm of this process (note this is not the agent reasoning cycle) may be as follows:

1) Agent obtains optimal path from continuous engine
2) Agent executes plan
3) Agent perceives non-nominal thruster gain(s)
4) Agent tests current plan, based upon perceived thruster gains, using the continuous engine
   a) If agent enters target region within the temporal constraint then plan execution continues
   b) If not then the situation must be anaylsed to seek a new solution based upon available resources
5) Upon entering target bound, regulatory control is initiated

Item 4 represents the most elaborate usage of the continuous engine; it is where the agent may evaluate both current implementations and possible solutions. Alternative solutions, in the event of initial plan failure, may involve extending the time horizon forward or investigation of control reconfiguration to circumvent usage of a faulty thruster. Upon obtaining an appropriate solution, this may then be implemented by the agent. Example output of plan testing under varying actuator efficiencies is displayed within Figures 3 and 4. For each test, a Boolean result is returned to indicate to the rational engine if a particular plan is still valid, under the current system capabilities.

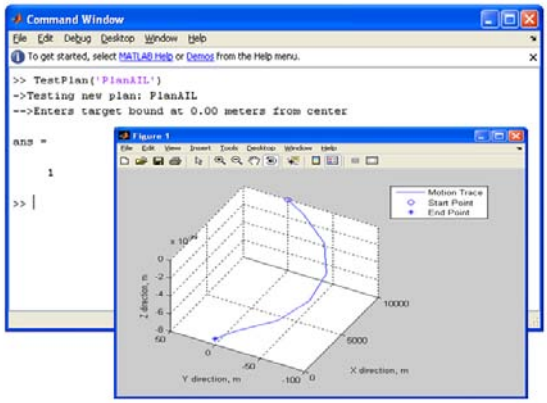

Fig 3: Test pass for existing plan to enter target bound: a boolean 1 is returned to Σ by Ω

Within implementation, the Simulink environment was configured to randomly generate thruster failure modes; in addition a particular actuator was configured to exhibit a continually decreasing gain. Upon agent activation, plan generation was triggered; once this plan was complete, the agent implemented the plan whilst dealing with any thruster failure modes. During plan execution, the low gain thruster was detected, triggering a plan test. Thrusters less than 98% efficient were perceived to 'miss' the target region when implementing the optimal open loop thrust sequences, initiating an alternative plan generation.

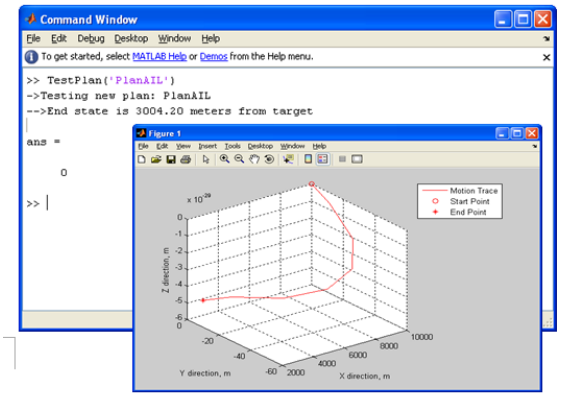

Fig 4: Test fail for existing plan to enter target bound: a boolean 0 is returned to Σ by Ω

The path following control method was observed to be more robust to gain reduction, however it was observed to fail temporal requirements under very low thruster efficiencies. Within the geostationary example, temporal constraints were not overly stringent: this resulted in an initial favoured agent response of extending the permissible time horizon for target region intersection. However, upon diagnosing a continually degrading thruster, control reconfiguration was initiated and plans reformatted for the new configuration. Once the path following control scheme was completed, either using open loop or feedback methods, the control type was switched to that of point tracking and the agent remained within the desired orbital bound. Figure 5 shows the VRML output for the agent system upon initialisation and entering the point tracking mission phase; note that scales have been altered within the VRML display.

Throughout the implementation, the Java based reasoning engine prescribed action based upon abstractions formed through an interface to the physical agent; further abstractions were evaluated by complex processes within the continuous engine. These abstractions were used within a logical framework to generate appropriate decisions, whilst concurrently maintaining an explicit knowledge base of action, intent and capability.

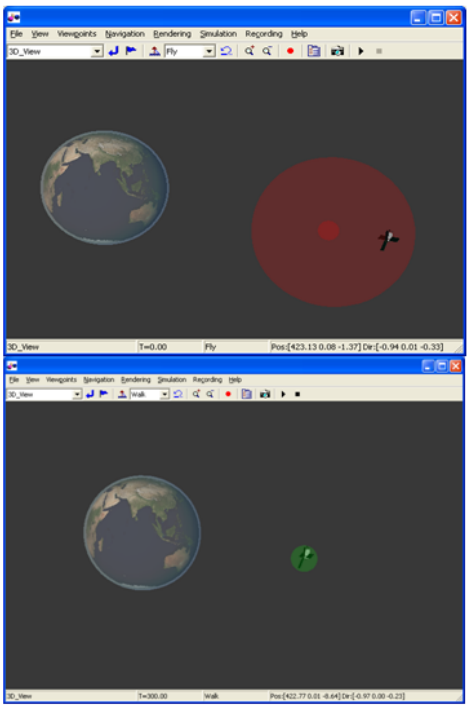

Fig 5: VRML output depicting pre and post activation of agent based control system

CONCLUSIONS

This paper has discussed and presented a theoretical physical agent framework that uses agents with the ability to predict and abstract out the future for decision making. This new framework brings improvements to prior agent based approaches in terms of:

1. *Reduced computational* complexity under changing dynamics, including system degradation.
2. *Aided abstraction programming* and potential verifiability of the resulting hybrid system of agent-environment interaction.
3. Improved control performance and the *ability to handle hybrid system situations* not considered by the agent programmer.

REFERENCES

[1] Michael Wooldridge and Nicholas R. Jennings, *Intelligent Agents: Theory and Practice*. Knowledge Engineering Review Journal, 1995, volume 10, pp 115-152.

[2] Anand S. Rao and Micheal P. Georgeff, *Formal Models and Decision Procedures for Multi-Agent Systems*. Technical note 61: Australian Artificial Intelligence Institute, June 1995.

[3] H. Van Dyke Parunak, *Practical And Industrial Applications of Agent-Based Systems*, 1998.

[4] Georgeff, M. P. and Lansky, A. L. (1987). Reactive Reasoning and Planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 677-682, Seattle, WA.

[5] Rafael H. Bordini, Jomi F. Hubner and Micheal Wooldridge, *Programming Multi-Agent Systems in AgentSpeak Using Jason*. Published by John Wiley and Sons, 2007.

[6] Fabio Bellifemine, Giovanni Caire and Domonic Greenwood, *Developing Multi-Agent Systems With Jade*. Published by John Wiley and Sons, 2007.

[7] Sandor M. Veres, *Autonomous Control Systems Using Agents: An Introduction*. In, Proceedings of IEE Workshop on Agent Based Control Systems. IET (IEE), 1-10 (2005).

[8] R. Brooks, *A Robust Layered Control System for a Mobile Robot*. Robotics and Automation, IEEE Journal in Robotics and Automation,Vol. 2, No. 1, March 1986.

[9] Sandor M. Veres and Aron G Veres, *Learning and Adaptation in Physical Agents*. In, *9th* IFAC Workshop *Adaptation and Learning in Control and Signal Processing (ALCOSP'07), St Petersburg, Russia, 29-31 Aug 2007*.

[10] Sandor M. Veres and Aron G. Veres, *Learning and Adaptation of Skills in Autonomous Physical Agents*. In, *17th World Congress of International Federation of Automatic Control, Seoul, Korea, 6-10 Jul 2008*. Seoul, Korea, Elsevier - Pergamon Press, 6pp, 2671-2676.

[11] Michael Wooldridge, *An Introduction to MultiAgent Systems*. Published by John Wiley and Sons, 2002.

[12] K. L. Myers, *Procedural Reasoning System User's Guide: A Manual for Version 2.0*. Technical report, Artificial Intelligence Center, SRI International, Menlo Park, CA, 2001.

[13] Belta, C., V. Isler, and G. Pappas, *Discrete Abstractions for Robot Motion Planning and Control in Polygonal Environments*. IEEE Transactions on Robotics, 2005. **21**(5): p. 864-874

[14] Fainekos, G.E., S.G. Loizou, and G.J. Pappas. *Translating Temporal Logic to Controller Specifications*. in *Proceedings of the 45th IEEE Conference on Decision & Control*. 2006. San Diego, CA, USA,

[15] Marius Kloetzer and Calin Belta, A Fully Automated Framework for Control of Linear Systems from LTL Specifications. In Hybrid Systems: Computation and Control (2006), pp. 333-347.

[16] Sandor M. Veres and Nicholas K. Lincoln, Sliding Mode control for Agents and Humans. In Proceedings, *TAROS'08, Towards Autonomous Robotic Systems 2008, Edinburgh, UK, 1-3 Sept 2008*. Edinburgh, UK, IC London, 13pp, 61-73.

[17] Sandor M Veres, *Natural Language Programming of Agents and Robotic Devices: Publishing for humans and machines in sEnglish*. Published by SysBrain, London, 2008, ISBN 978-0-9558417-0-5.

[18] Louise A. Dennis, Michael Fisher, Nicholas Lincoln, Alexei Lisitsa and Sandor M. Veres, *Agent Based Approaches to Engineering Autonomous Space Software*. In Proc. Workshop on Formal Methods for Aerospace. Electronic Proceedings in Theoretical Computer Science 20. March 2010.

[19] S. M. Veres, S. B Gabriel, D. Q Mayne and E. Rogers, *Analysis of Formation Flying Control of a Pair of Nano-satellites*. AIAA Journal of Guidance, Control, and Dynamics, 25 (5). pp. 971-974.

[20] R. Pongvthithum, S. M. Veres, S. B. Gabriel and E. Rogers, *Universal Adaptive Control of Satellite Formation Flying*, International Journal of Control, Volume 78(1), January 2005

[21] D. Ya. Rokityanski and S.M. Veres, Application of Ellipsoidal Estimation to Satellite Control. *Mathematical and Computer Modelling of Dynamical Systems*, 11, (2), pages 239-249 (2005).

[22] N.K. Lincoln and S.M. Veres. *Components of a Vision Assisted Constrained Autonomous Satellite Formation Flying Control System*. International Journal of Adaptive Control and Signal Processing, 21(2-3):237–264, October 2006.

[23] S.M. Veres, (2006) Autonomous formation flying of satellite robots: the mechanical control layer. In Proceedings, *TAROS 2006: Towards Auton. Robotic Systems, Guildford, UK, 4-6 Sep 2006.*

[24] S.M. Veres, Thanapalan, K., Gabriel, S. and Rogers, E. (2006) Reconfigurable Controller Design For Operational Safety in Satellite Formation Flying. In Proceedings, *International Control Conference 2006, Glasgow, Scotland, UK, 30 Aug - 1 Sept, 2006.*

[25] David A. Vallado and Wagne D. McClain, *Fundamentals of Astrodynamics and Applications (Space Technology Library)*. Published by Springer Link, 2001.

[26] Marcel J. Sidi, *Spacecraft Dynamics and Control*, Published by Cambridge University Press,1997.