Declarative Abstractions for Agent Based Hybrid Control Systems*

Louise A. Dennis[1], Michael Fisher[1], Nicholas K. Lincoln[2], Alexei Lisitsa[1], and Sandor M. Veres[2]

[1] Department of Computer Science, University of Liverpool, UK
[2] School of Engineering, University of Southampton, UK
Contact: `L.A.Dennis@liverpool.ac.uk`

Abstract. Modern control systems are limited in their ability to react flexibly and autonomously to changing situations by the complexity inherent in analysing environments where many variables are present. We aim to use an agent approach to help alleviate this problem and are particularly interested in the control of satellite systems using BDI agent programming as pioneered by the PRS.
Such systems need to generate discrete abstractions from continuous data and then use these abstractions in rational decision making. This paper provides an architecture and interaction semantics for an abstraction engine to interact with a hybrid BDI-based control system.

1 Introduction

Modern control systems are limited in their ability to react flexibly and autonomously to changing situations. The complexity inherent in analysing environments where many continuous variables are present, and dynamically changing, has proved to be a challenge. In some situations one control system may need to be swapped for another. This, quite severe, behavioural change is very difficult to handle just within the control systems framework.

We approach the problem from the perspective of satellite control systems. Consider a single satellite attempting to maintain a geostationary orbit. Current systems maintain orbits using feedback controllers. These implicitly assume that any errors will be minor and easily corrected. In situations where more major errors occur, e.g. caused by thruster malfunction, it is desirable to change the controller or modify the hardware configuration. The complexity of the decision task has proved to be a challenge to the type of imperative programming approaches traditionally used within control systems programming.

There is a long standing tradition, pioneered by the PRS system [17], of using agent languages (and other logic programming approaches – e.g. [29]) to control and reason about such systems. We consider a satellite to be an *agent* which consists of a discrete (rational decision making) engine together with a continuous (calculation) engine. The rational engine uses the *Belief-Desire-Intention*

* Work funded by EPSRC grants EP/F037201/1 and EP/F037570/1

(BDI) theory of agency [23] to make decisions about appropriate controllers and hardware configurations for the satellite. It is assisted by the continuous engine which can perform predictive modelling and other continuous calculations.

In order for such an architecture to be clear and declarative it is necessary to generate discrete abstractions from continuous data. It is also necessary to translate discrete actions and queries back into continuous commands and queries. In order to do this we introduce an *Abstraction Engine* whose purpose is to manage communication between the continuous and discrete parts of the system in a semantically clear way. (See Figure 1, later.)

This paper provides an architecture and interaction semantics which describe the way an Abstraction Engine interacts with a hybrid BDI-based control system. We present a case study and discuss its implications for the choice and design of declarative languages for hybrid control systems.

This paper is organised as follows. Section 2 provides some background material. Section 3 provides an architecture for a hybrid control system with explicit abstraction. Section 4 provides an operational semantics for interaction between the major components of such a system. Section 5 presents a prototype implementation of the architecture and semantics and Section 6 discusses a case study performed in this system. Section 7 draws some preliminary conclusions and discusses the further work motivated by the prototype and case study.

2 Background

2.1 Control Systems

Satellite systems are typically governed by feedback controllers. These continuously monitor input sensors and compare the values to a desired state. They then alter various aspects of the system accordingly, for instance by increasing or decreasing power or adjusting direction. Typically the actual controller is specified using differential equations and operates in a continuous fashion.

A *hybrid system* is one in which the desired controller is a function which not only has continuous regions but also distinct places of discontinuity between those regions, such as the moment when a bouncing ball changes direction on impact. In practical engineering contexts, such as satellites, it is frequently desirable to change feedback controllers at such points. Appropriate control mechanisms for such hybrid systems is a very active area of research [26; 8; 12].

2.2 BDI Agents

We view an agent as an *autonomous* computational entity making its own decisions about what activities to pursue. Often this involves having goals and communicating with other agents in order to accomplish these goals [30]. *Rational agents* make decisions in an explainable way and, since agents are autonomous, understanding *why* an agent chooses a particular course of action is vital.

We often describe each agent's *beliefs* and *goals* which in turn determine the agent's *intentions*. Such agents make decisions about what action to perform, given their current beliefs, goals and intentions. This approach has been popularised through the influential BDI (Belief-Desire-Intention) model of agency [23].

2.3 The Problem of Abstraction

Generating appropriate abstractions to mediate between continuous and discrete parts of a system is the key to any link between a control system and a reasoning system. Abstractions allow concepts to be translated from the quantitative data necessary to actually run the underlying system to the qualitative data needed for reasoning. For instance a control system may store precise location coordinates, represented as real numbers, while the reasoning system may only be interested in whether a satellite is within reasonable bounds of its desired position.

The use of appropriate abstractions is also important for verification techniques, such as model checking, for hybrid systems [2; 15; 25; 21] and potentially for declarative prediction and forward planning [19]. These require the continuous search space to be divided into a finite set of regions which can be examined.

Ideally the generation of such abstractions should itself be declarative. This would make clear say, that a decision to change a fuel line corresponds directly to the activation of certain valves within the satellite system.

3 Architecture

Our aim is to produce a hybrid system embedding existing technology for generating feedback controllers and configuring satellite systems within a decision making part based upon agent technologies and theories. The link is to be controlled by a semantically clear *abstraction layer*. At present we consider a single agent case and leave investigation of multi-agent scenarios to future work.

Figure 1 shows an architecture for our system. Real time control of the satellite is governed by a traditional feedback controller drawing its sensory input from the environment. This forms a *Physical Engine* (Π). This engine communicates with an agent architecture consisting of an *Abstraction Engine* (A) that filters and discretizes information. To do this A may a use a *Continuous Engine* (Ω) to make calculations involving the continuous information. Finally, the *Rational Engine* (R) contains a "Sense-Reason-Act" loop. Actions involve either calls to the Continuous Engine to calculate new controllers (for instance) or instructions to the change these controllers within the Physical Engine. These instructions are passed through the Abstraction Engine for reification.

In this way, R is a traditional BDI system dealing with discrete information, Π and Ω are traditional control systems, typically generated by MatLab/Simulink, while A provides the vital "glue" between all these parts.

4 Semantics of Interaction

We assume a hybrid control system consisting of a Physical Engine (Π), a Continuous Engine (Ω), an Abstraction Engine (A) and a Reasoning Engine (R).

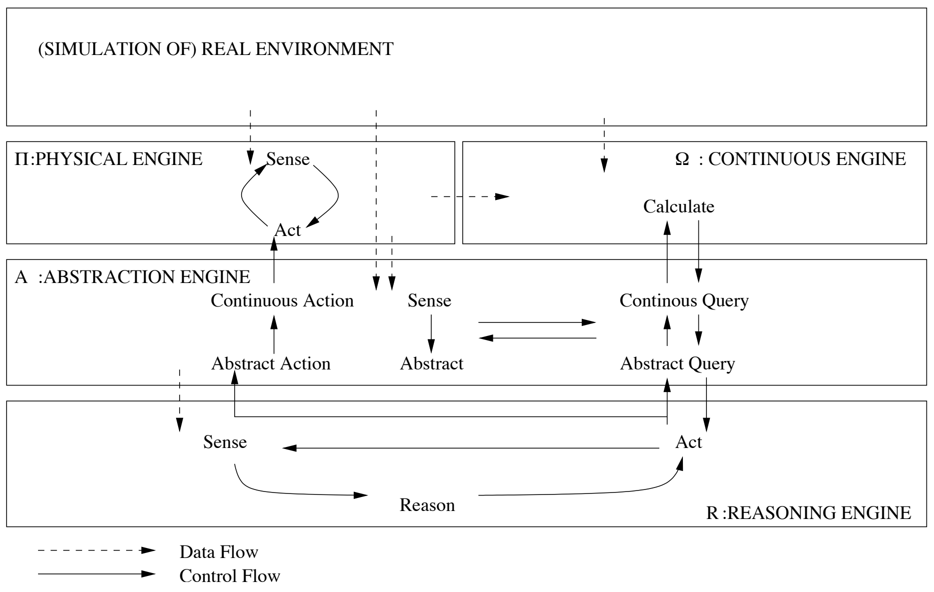

Fig. 1. Hybrid Agent Architecture

We present an operational semantics for the interaction of these engines. This semantics makes minimal assumptions about the internals of the engines, but nevertheless places certain constraints upon their operation and the way they interact with the external environment. This semantics is designed to allow a *declarative abstraction language* to be developed for the Abstraction Engine, A. An implementation of the architecture and the semantics is discussed in Section 5 and a case study using the implementation is discussed in Section 6. The implementation and case study influenced the development of the semantics and serve as additional motivation for the design of its components.

We assume the Abstraction Engine has access to four sets of data. These are Δ (description of the real world and Physical Engine), Σ (beliefs/abstractions shared with the Reasoning Engine), Γ (abstract actions the Reasoning Engine wishes the Physical Engine to take), Q (abstract queries the Reasoning Engine wishes to make of the Continuous Engine). Σ, Γ and Q are all assumed to be sets of ground atomic formulae. Therefore, we can represent the entire system as a tuple $\langle \Pi, \Omega, A, R, \Delta, \Sigma, \Gamma, Q\rangle$. For clarity, in the semantics we will replace parts of this tuple with ellipsis $(\ldots)$ if they are unchanged by a transition.

4.1 Abstraction and Reification

We assume that the Abstraction Engine, A, contains processes of *abstraction* (*abs*) and *reification* (*rei*) and that these form the primary purpose of A. Indeed we use the reification process in the semantics via the transition $A \xrightarrow{rei(p)} A'$

which indicates any internal changes to the abstraction engine as it reifies some request p from the Rational Engine. For instance converting a thruster change request, to a sequence of valve and switch activations, or adding additional information about the current real-valued position of the satellite to a request for the calculation of a new feedback controller to move the satellite along a path.

Implicitly we assume that *abs* represents a function from Δ to the shared beliefs Σ. Similarly we assume that reification takes Γ and Q and converts them into sequences of instructions for the Physical Engine or calls for calculations from the Continuous Engine.

4.2 Internal Transitions

We assume all four engines may take internal transitions which we represent as $\xrightarrow{?}$ to indicate some unknown internal state change. So, for instance,

$$\frac{\Pi \xrightarrow{?} \Pi'}{\langle \ldots, \Pi, \ldots \rangle \xrightarrow{?_{\Pi}} \langle \ldots, \Pi', \ldots \rangle} \tag{1}$$

represents an internal state change in the Physical Engine which leaves the rest of the system unaltered. Similar rules exist for the three other engines.

4.3 Perception

We assume that both the Abstraction Engine and Reasoning Engine incorporate a perception mechanism by which they can "read in" data represented as first-order predicates and represent this information internally as, for instance, beliefs. We write $A \xrightarrow{per(S)} A'$ as the process by which A reads in first-order data S. Similarly we write $R \xrightarrow{per(S)} R'$ for the Reasoning Engine's perception process. We represent this as a transition since the Reasoning Engine and Abstraction Engine will change state (e.g., adding beliefs and/or events) during perception.

We have data, Δ, that arrives from the Physical Engine. This data might not be represented in first-order form. We require a function $\mathit{fof}(\Delta)$ that transforms the language of Δ into appropriate ground atomic predicates (though these may represent real numbers). This is the first stage in an abstraction process.

Furthermore we assume that A keeps a log, L, of snapshots of the current state of the physical system, as represented by Δ. So A can be represented as (L, A_r) where A_r represents all of A's internal data structures apart from the log. We treat the log as a list with ':' as the `cons` function.

This allows us to define a semantics for perception as follows:

$$\frac{A_r \xrightarrow{per(\mathit{fof}(\Delta))} A'_r}{\langle \ldots, (L, A_r), R, \Delta, \ldots \rangle \xrightarrow{per_A(\Delta)} \langle \ldots, (\mathit{fof}(\Delta) : L, A'_r), R, \emptyset, \ldots \rangle} \tag{2}$$

The Abstraction and Rational engines my also perceive the shared beliefs.

$$\frac{A \xrightarrow{per(\Sigma)} A'}{\langle \ldots, A, R, \Delta, \Sigma, \ldots \rangle \xrightarrow{per_A(\Sigma)} \langle \ldots, A', R, \Delta, \Sigma, \ldots \rangle} \tag{3}$$

$$\frac{R \xrightarrow{per(\Sigma)} R'}{\langle \ldots, A, R, \Delta, \Sigma, \ldots, \rangle \xrightarrow{per_R(\Sigma)} \langle \Pi, \Omega, A, R', \Delta, \Sigma, \ldots \rangle} \tag{4}$$

4.4 Operating on Shared Beliefs

Both the Abstraction Engine and Reasoning Engine can operate on the memory they share. We assume that both these engines can perform transitions $+_{\Sigma}b$ and $-_{\Sigma}b$ to add and remove shared beliefs where b is a ground first-order formula. We represent internal changes to the engines caused by such actions as transitions.

$$\frac{A \xrightarrow{+_{\Sigma}b} A'}{\langle \ldots, A, R, \Delta, \Sigma, \ldots \rangle \xrightarrow{+_{\Sigma,A}b} \langle \ldots, A', R, \Sigma \cup \{b\}, \ldots \rangle} \tag{5}$$

$$\frac{A \xrightarrow{-_{\Sigma}b} A'}{\langle \ldots, A, R, \Delta, \Sigma, \ldots \rangle \xrightarrow{-_{\Sigma,A}b} \langle \ldots, A', R, \Sigma \backslash \{b\}, \ldots \rangle} \tag{6}$$

$$\frac{R \xrightarrow{+_{\Sigma}b} R'}{\langle \ldots, R, \Delta, \ldots \rangle \xrightarrow{+_{\Sigma,R}b} \langle \ldots, R', \Sigma \cup \{b\}, \ldots \rangle} \tag{7}$$

$$\frac{R \xrightarrow{-_{\Sigma}b} R'}{\langle \ldots, R, \Delta, \Sigma, \ldots \rangle \xrightarrow{-_{\Sigma,R}b} \langle \ldots, R', \Sigma \backslash \{b\}, \ldots \rangle} \tag{8}$$

Note that the abstraction process employed by the Abstraction Engine is intended to be one of transforming the predicates generated via $\mathit{fof}(\Delta)$ into a set of shared beliefs which are then added to Σ. One of our interests is in finding ways to present this transformation in as expressive and declarative a fashion as possible. We discuss this further in Sections 6 and 7.

4.5 Calculation

We allow the Abstraction Engine to transform the predicate representation of the calculation, p, into the input language of the Continuous Engine. This is similar to the inverse of the operation performed by fof and so we term it fof^{-1}. Usually this involves trivial changes (e.g. $\mathit{set\_valve}(x)$ becomes $\mathit{set\_x\_valve}$ – translating between the parameterised form used by the Rational Engine and the non parameterised form used by the Physical and Continuous Engines).

When the Abstraction Engine requests a calculation from the Continuous Engine it could wait for an answer. However such an answer may take time to calculate and the Abstraction Engine may need to continue handling incoming data. Some agent languages (such as *Jason* [7]) allow intentions to be suspended while other parts of an agent may continue to run. We follow this approach and represent requesting and receiving the answer to a calculation via two rules. We indicate the process of requesting a calculation by $A \xrightarrow{calc(p,V)} A'(V)$, where we write $A'(V)$ to indicate that the state of the Abstraction Engine contains a free variable, V, that is awaiting instantiation. We represent the change in state from Ω when it is not performing a calculation to when it is via $\Omega \xrightarrow{calc} \Omega(fof^{-1}(p))$.

$$\frac{A \xrightarrow{calc(p,V)} A'(V) \qquad \Omega \xrightarrow{calc} \Omega(fof^{-1}(p))}{\langle \ldots, \Omega, A, \ldots \rangle \xrightarrow{calc(p,V)} \langle \ldots, \Omega(fof^{-1}(p)), A'(V), \ldots \rangle} \tag{9}$$

$$\frac{\Omega(fof^{-1}(p)) = t \qquad A(V) \xrightarrow{V=t} A'(t)}{\langle \ldots, \Omega, A(V), \ldots \rangle \xrightarrow{V=t} \langle \ldots, \Omega, A'(t), \ldots \rangle} \tag{10}$$

When the Reasoning Engine, R, wishes to request a continuous calculation it places a request in the query set, Q, which A then reifies. We implicitly assume that the reification will include one or more calculation requests to the Continuous Engine but that the only change to the overall system state is to the internal state of A. We write reification as a transition $\langle \ldots, A, \ldots \rangle \xrightarrow{rei(q,V)} \langle \ldots, A', \ldots \rangle$.

$$\frac{(q, V) \in Q \qquad \langle \ldots, A, \ldots \rangle \xrightarrow{rei(q,V)} \langle \ldots, A', \ldots \rangle}{\langle \ldots, A, R, \ldots, Q \rangle \xrightarrow{qcalc(q,V)} \langle \ldots, A', R, \ldots, Q\{V/t\} \rangle} \tag{11}$$

As with the Abstraction Engine, we split the processes of requesting a calculation and receiving an answer in the Reasoning Engine:

$$\frac{R \xrightarrow{rcalc(q,V)} R'(V)}{\langle \ldots, A, R, \ldots, Q \rangle \xrightarrow{rcalc(q,V)} \langle \Pi, \Omega, A, R'(V), \ldots, \{(q, V)\} \cup Q \rangle} \tag{12}$$

$$\frac{\langle \ldots, A, R(V), \ldots, Q \rangle \xrightarrow{rei(q,V)} \langle \ldots, A', R(V), \ldots, Q' \rangle \quad (q, t) \in Q' \quad R(V) \xrightarrow{V=t} R'(t)}{\langle \ldots, A, R(V), \ldots, Q \rangle \xrightarrow{V=t} \langle \Pi, \Omega, A', R'(t), \ldots, Q' \backslash \{(q, t)\} \rangle} \tag{13}$$

4.6 Performing Tasks

Finally, A can request that Π makes specific updates to its state.

$$\frac{A \xrightarrow{run(\gamma)} A' \qquad \Pi \xrightarrow{fof^{-1}(\gamma)} \Pi'}{\langle \Pi, \Omega, A, \ldots \rangle \xrightarrow{run(\gamma)} \langle \Pi', \Omega, A', \ldots, \rangle} \tag{14}$$

R can request changes to Π, but A reifies these requests. The reification may involve several calls to $run(\gamma)$ and these are all amalgamated into one system transition: $\langle \Pi, \Omega, A, \ldots \rangle \xrightarrow{rei(\gamma)} \langle \Pi', \Omega, A', \ldots, \rangle$.

$$\frac{R \xrightarrow{do(\gamma)} R'}{\langle \ldots, R, \ldots, \mathit{\Gamma}, Q\rangle \xrightarrow{do_R(\gamma)} \langle \ldots, R', \ldots, \{\gamma\} \cup \mathit{\Gamma}, Q\rangle} \tag{15}$$

$$\frac{\gamma \in \mathit{\Gamma} \qquad \langle \mathit{\Pi}, \mathit{\Omega}, A, \ldots\rangle \xrightarrow{rei(\gamma)} \langle \mathit{\Pi}', \mathit{\Omega}, A', \ldots, \rangle}{\langle \mathit{\Pi}, \mathit{\Omega}, A, \ldots, \mathit{\Gamma}, Q\rangle \xrightarrow{do(\gamma)} \langle \mathit{\Pi}', \mathit{\Omega}, A', \ldots, \mathit{\Gamma}\backslash\{\gamma\}, Q\rangle} \tag{16}$$

5 Implementation

We have implemented a prototype system to explore the requirements for the Abstraction Engine. The simulated environment, Physical and Continuous Engines are all implemented in MatLab using the Simulink tool kit.

The Abstraction Engine and Reasoning Engine are both written in the JAVA-based Gwendolen agent programming language[3] as separate agents. Requests for calculations or actions from the Reasoning Engine are read into the Abstraction Engine as 'perform' goals Therefore the plans for handling these goals are equivalent to the function *rei* in the abstract semantics and execution of those plans is equivalent to the transition $A \xrightarrow{rei(p)} A'$. The Continuous Engine may, as a side effect of its calculations, place configuration files in the shared file system for use by the Physical Engine. Communication between the JAVA process and the two MatLab processes is via JAVA sockets and exists in a thin JAVA "Environment" layer between the Abstraction Engine and the MatLab parts of the system.

The Physical Engine is assumed to have direct access to a satellite's sensors. At present the information is transmitted to the Abstraction Engine in the form of a simple string tag (which relates to the way the data values flow around the Simulink model), followed by a number of arguments which are mostly real numbers. These tags and values are then converted to predicates by the Abstraction Engine. For instance the Physical Engine groups data by 'thruster' and tags them, for instance "xthruster1" (for the data from the first thruster in the X direction) followed by values for the *current*, *voltage* and *fuel pressure* in the thruster (say C, V and P). It is more natural, in the Abstraction Engine, to represent this data as a predicate $\mathit{thruster}(x, 1, C, V, P)$ than as the predicate $\mathit{xthruster}1(C, V, P)$. At the moment the JAVA environment handles the necessary conversion which is equivalent to the *fof* function from the semantics.

The JAVA environment also handles all four data sets, $\mathit{\Delta}$, $\mathit{\Sigma}$, $\mathit{\Gamma}$ and Q and sends predicates to the relevant agents at appropriate moments. $\mathit{\Gamma}$ and Q are treated as messages with performatives indicating the type of goal they should be transformed into by the Abstraction Engine.

When the Abstraction Engine requests calculations from the Continuous Engine it requests that an `M-file` (MatLab function) is executed. It sends the Continuous Agent the name of the `M-file` followed by any arguments the `M-file`

[3] The choice of language was dictated entirely by convenience. One of the purposes of this case study is to explore the desirable features of a BDI-based control language.

requires. Gwendolen allows intentions to be suspended until some event occurs. We use this explicitly in both engines to force the agent to wait until it perceives the result of calculation. In particular this allows the Abstraction Engine to continue processing new information even while waiting for a result. Once the `M-file` has been executed the Continuous Engine of the agent returns the resulting data to the Abstraction Engine. (We are exploring whether the Continuous Engine should also sense data from the system model to assist its calculations.)

Finally, at present the Abstraction Engine only keeps the most recent snapshot of Δ and discards older information rather than keeping it as a log.

5.1 sEnglish for data abstractions and programming

Both the Physical Engine and Continuous Engine need to work with the Abstraction Engine to produce abstractions for the Reasoning Engine. To make the meanings of abstractions clear, concise and easy to remember for the programmer of the agent system, a high-level notation called *system English* (sEnglish, [24; 27; 28]), is used to represent the MatLab code which then generates the `M-files` used by the Continuous Engine and parts of the Physical Engine. sEnglish provides a natural link between a predicate style formulation and the underlying MatLab code. It uses an ontology to organise modelling objects that can be arguments for the predicates stated by both the Physical Engine and Continuous Engine. Both can be partially programmed in sEnglish (indeed, the Continuous Engine represents a generic system entirely configurable by sEnglish) and each produced predicate has an equivalent, and natural, sEnglish description. This use of natural language programming provides human insight into the data abstractions of the agent. This makes the programming of the Physical Engine and Continuous Engine more structured and easier to maintain.

In future work we hope to be able to extend the use of sEnglish to provide the language of communication between the continuous and first-order parts of the system. This would give a principled structure to the fof and fof^{-1} functions in the Abstraction Engine. For instance, in our case study (see Section 6) the Physical Engine sends thruster information as a string "xthruster1" (for the first thruster in the x thruster bank) followed by the data about that thruster. This is then converted to the predicate $\mathit{thruster}(x, 1, C, V, P)$ using simple rules that match "xthruster1" to "CthrusterD" where C represents a single character and D a single digit. If no specific translation is supplied then the stream tag followed by a stream args is converted automatically to the predicate $\mathit{tag}(\mathit{args})$.

6 Case Study: Geostationary Orbit

A geostationary orbit (a GEO orbit) is characterized as an equatorial orbit (zero inclination), with near zero eccentricity and an orbital period equal to one sidereal day. A satellite maintaining such an orbit will remain at a fixed longitude at zero degrees latitude. Thus, with respect to an Earth based observer, the satellite will remain in a fixed overhead position. Numerous benefits follow from the use of geostationary orbits, the principal one of these being highlighted originally in [1]: three geostationary satellites stationed equidistantly are capable of providing worldwide telecommunications coverage.

While telecommunications is an obvious application area for such orbits, observation satellites and other applications also make heavy use of them. Consequently the geostationary orbit represents prime real-estate for satellite platforms. GEO locations are hotly contested and their allocation is politicized. This makes it important that such locations are used optimally and that satellites do not stray far from their assigned position.

Once placed in a GEO orbit at a specified longitude, *station keeping procedures* are used to ensure that the correct location is retained. Such station keeping procedures are required because the disturbances caused by solar radiation pressure (SRP), luni-solar perturbations and Earth triaxiality naturally cause an object to move from an orbit location in which it has been placed. These disturbances result in changes to the nominal orbit which must be corrected for. A standard feedback controller is able to handle these tasks.

6.1 Scenario

We implemented a Simulink model of a satellite in geostationary orbit. A MatLab function (an `M-file` written in sEnglish) was created to calculate whether a given set of coordinates were within an acceptable distance of the satellite's desired orbital position (comp_distance). A second function (plan_approach_to_centre), based on [20], was also written to calculate a new feedback controller to steer the satellite along a path back to its desired orbital position (for use if the satellite strayed out of bounds – e.g. because of fuel venting rom a ruptured line). These functions were made available to the agent's Continuous Engine.

Controls were made available in the Physical Engine which could send a particular named *activation plan* to the feedback controller (set_control), switch thrusters on and off (set_x1_main, set_x2_main, set_y1_main, etc.), control the valves that determined which fuel line was being utilised (set_x1_valves , etc.) and change the thruster being used in any direction (set_x_bank, etc.).

The satellite was given thrusters in three directions (X, Y and Z) each of which contained two fuel lines. This enabled the agent to switch fuel line in the event of a rupture (detectable by a drop in fuel pressure). We also provided up to five redundant thrusters, allowing the agent to switch to a redundant thruster if both fuel lines were broken.

6.2 The Abstraction Engine

The Abstraction Engine code in the case of one redundant thruster is as follows:

Code fragment 6.1 Geostationary Orbit:Abstraction Engine

```
+location(L1, L2, L3, L4, L5, L6) : {𝓑 bound_info(V1)} ←
          calculate (comp_distance(L1, L2, L3, L4, L5, L6), Val),
          ∗result (comp_distance(L1, L2, L3, L4, L5, L6), Val),
          −result(comp_distance(L1, L2, L3, L4, L5, L6), Val),
          +bound_info(Val);

+bound_info(in) : {𝓑 proximity_to_centre(out)} ←
          −bound_info(out),
          −Σproximity_to_centre(out), +Σproximity_to_centre(in);

+bound_info(out) : {𝓑 proximity_to_centre(in)} ←
          −bound_info(in),
          −Σproximity_to_centre(in), +Σproximity_to_centre(out);

+!maintain_path : {𝓑 proximity_to_centre(in)} ← run(set_control(maintain));
+!execute(P)    : {𝓑 proximity_to_centre(out)} ← run(set_control(P));

+!plan_approach_to_centre(P) : {𝓑 location(L1, L2, L3, L4, L5, L6)} ←
          calculate (plan_approach_to_centre(L1, L2, L3, L4, L5, L6), P),
          ∗result (plan_approach_to_centre(L1, L2, L3, L4, L5, L6), P),
          −result(plan_approach_to_centre(L1, L2, L3, L4, L5, L6), P),
          +Σplan_approach_to_center(P);

−broken(X) :
    {𝓑 thruster_bank_line(X, N, L), 𝓑 thruster(X, N, C, V, P), P1 < 1} ←
          +Σ(broken(X));

+thruster(X, N, C, V, P):
    {~ 𝓑 broken(X), 𝓑 thruster_bank_line(X, N, L), P1 < 1} ←
          +Σbroken(X);
+thruster(X, N, C, V, P):
    {𝓑 broken(X), 𝓑 thruster_bank_line(X, N, L), 1 < P1} ←
          −Σbroken(X).

+!change_fuel_line(T, 1) : {𝓑 thruster_bank_line(T, B, 1)} ←
          run(set_valves(T, B, off, off, on, on)),
          −Σthruster_bank_line(T, B, 1),
          +Σthruster_bank_line(T, B, 2),
          −Σbroken(T);
+!change_bank(T) : {𝓑 thruster_bank_line(T, B, L)} ←
          B1 is B + 1;
          run(set_bank(T, B1)),
          run(set_main(T, B, off)),
          run(set_main(T, B1, on)),
          −Σthruster_bank_line(T, B, L),
          +Σthruster_bank_line(T, B1, 1),
          −Σbroken(T);
```

We use a standard BDI syntax: $+b$ indicates the addition of a belief; $!g$ indicates a perform goal, g, and $+!g$ the commitment to the goal. A plan $e : \{g\} \leftarrow b$ consists of a trigger event, e, a guard, g, which must be true before the plan can be executed and a body b which is executed when the plan is selected.

Gwendolen allows plan execution to be suspended while waiting for some belief to become true. This is indicated by the syntax $*b$ which means "wait until b is believed". This is used in conjunction with '*calculate*' to allow the engine to continuing producing abstractions from incoming data while calculation occurs. The new belief is then immediately removed so that further calls to '*calculate*' suspend as desired. Ideally, a language would handle this more cleanly without the awkward "call-suspend-clean-up" sequence.

Abstraction and Reification. Ideally we would like to be able to clearly derive the functions *abs* and *rei* from the Abstraction Engine code.

In the above the *abs* process is represented by plans triggered by belief acquisition. For instance the code in lines 30–32 represents an abstraction from the predicate $thruster(X, N, C, V, P)$, where C, V and P are real numbers, to the predicate $broken(X)$. However, it is harder to see how the acquisition of location data (line 1) generates abstractions about "proximity to centre".

The reification of the abstract query "plan_approach_to_centre(P)" (line 20), converts it to a call with real number arguments (the current location) and then causes the intention to wait for the result of the call. Similarly the code in lines 42–49 shows the reification of the predicate, $change\_bank(T)$, into a sequence of commands to set the bank and turn the relevant thrusters off or on, but this is obscured by housekeeping to manage the system's beliefs.

An area of further work is to find or develop a language for the Abstraction Engine that expresses these two functions in a clearer way.

6.3 The Reasoning Engine

The reasoning engine code is as follows:

Code fragment 6.2 Geostationary Orbit: Reasoning Engine

```
+proximity_to_centre(out) : {⊤} ← −proximity_to_centre(in),
        +!get_to_centre ;
+proximity_to_centre(in) : {⊤} ← −proximity_to_centre(out),
        perform(maintain_path);

+!get_to_centre   : {ℬ proximity_to_centre(out)} ←
        query(plan_approach_to_centre(P)), ∗plan_approach_to_centre(P),
        perform(execute(P)),
        −Σplan_approach_to_centre(P);

+broken(X): {ℬ thruster_bank_line(X, N, 1)} ←
        perform(change_fuel_line(X, N));
+broken(X): {ℬ thruster_bank_line(X, N, 2)} ←
        perform(change_bank(X, N));
```

We use the same syntax as we did for the Abstraction Engine. Here the actions, 'perform' and 'query', request that the Abstraction Engine adopt a goal.

The architecture lets us represent the high-level decision making aspects of the program in terms of the beliefs and goals of the agent and the events it observes. So, for instance, when the Abstraction Engine observes that the thruster line pressure has dropped below 1, it asserts a shared belief that the thruster is broken. When the Reasoning Engine observes that the thruster is broken, it then either changes fuel line, or thruster bank. This is communicated to the Abstraction Engine which then sets the appropriate valves and switches.

7 Conclusions

This paper has explored creating declarative abstractions to assist the communication between the continuous and discrete parts of a hybrid control system.

We believe that it is desirable to provide a clear separation between abstraction and reasoning processes in hybrid autonomous control systems. We believe this is beneficial not only for the clarity of the code, but also for use in applications such as forward planning and model checking.

We have created a formal semantics describing how such an Abstraction Engine would interact with the rest of the system, and discussed a prototype BDI based Abstraction Engine and the issues this raises in terms of a suitable language for generating discrete abstractions from continuous data. We believe that this is the first work linking autonomous agents and control systems via a formal semantics.

7.1 Future Work

The work on hybrid agent systems with declarative abstractions for autonomous space software is only in its initial stages and considerable further work remains to be investigated.

Further Case Studies. We are keen to develop a repertoire of case studies, beyond the simple one presented here, which will provide us with benchmark examples upon which to examine issues such as more sophisticated reasoning tasks, multi-agent systems, forward planning, verification and language design.

In addition we aim, next, to investigate a case study involving multiple satellites attempting to maintain and change formation in low Earth orbit. This presents significant planning challenges.

Custom Language. At the moment the BDI language we are using for the Abstraction Engine is not as clear as we might like. In particular the functions of abstraction and reification are not so easy to "read off" from the code and are obscured somewhat by housekeeping tasks associated with maintaining consistent shared beliefs about which thrusters are in operation.

A further degree of declarativeness can be achieved within the Abstraction Engine by separation of abstraction evaluation and the control features. Due to the dynamic setting in which abstraction is performed "on-the-fly" reacting to incoming sensory data, it can be naturally seen as query processing for data streams [13; 14]. This viewpoint would provide a clean semantics for abstraction evaluation, based on the theory of stream queries [14] and would hopefully avoid the need to devote too much space to storing data logs. We also aim to investigate the extent to which techniques and programming languages developed for efficient data stream processing (from e.g. [3; 18]) can be re-used within the Abstraction Engine. It is possible that something similar might be used for the reification process as well, although this is more speculative.

We are interested in investigating programming languages for the Reasoning Engine – e.g. languages such as *Jason* [7] or 3APL [9] are similar to Gwendolen, but better developed and supported. Alternatively it might be necessary to use a language containing, for instance, the concept of a *maintain* goal. Nuch of a satellite's operation is most naturally expressed in terms of *maintaining* a state of affairs (such as a remaining on a particular path).

Planning and Model Checking. At present the `M-file` employed to create a new controller that will return the satellite to the desired orbit uses a technique based on hill-climbing search [20]. We are interested in investigating temporal logic and model-checking based approaches to this form of planning for hybrid automata based upon the work of Kloetzer and Belta [19].

Model checking techniques also exist [6] for the verification of BDI agent programs which could conceivably be applied to the Reasoning Engine. Abstraction techniques would then be required to provide appropriate models of the continuous and physical engines and it might be possible to generate these automatically from the abstraction and reification functions.

There is also a large body of work on the verification of hybrid systems [2; 15] which would allow us to push the boundaries of verification of such systems outside the limits of the Reasoning Engine alone.

Multi-Agent Systems. We are interested in extending our work to multi-agent systems and groups of satellites that need to collaborate in order to achieve some objective. In particular there are realistic scenarios in which one member of a group of satellites loses some particular functionality meaning that its role within the group needs to change. We believe this provides an interesting application for multi-agent work on groups, teams, roles and organisations [10; 16; 11; 22], together with the potential for formal verification in this area.

Since the individual agents in this system will be discrete physical objects and will be represented as such in any simulation we don't anticipate major challenges to the architecture itself from moving to a multi-agent scenario. However we anticipate interesting challenges from the point of view of coordination and communication between the agents.

Bibliography

[1] A. C. Clarke. Extra-Terrestrial Relays: Can Rocket Stations Give World-wide Radio Coverage? *Wireless World*, pages 305–308, 1945.

[2] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The Algorithmic Analysis of Hybrid Systems. *Theor. Comput. Sci.*, 138(1):3–34, 1995.

[3] A. Arasu, S. Babu, and J. Wisdom. The cql continuous query language: Semantic foundations and query execution. Technical Report 2003-67, Stanford, 2003.

[4] R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications*. Springer-Verlag, 2005.

[5] R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah-Seghrouchni, editors. *Multi-Agent Programming: Languages, Tools and Applications*. Springer-Verlag, 2009.

[6] R. H. Bordini, L. A. Dennis, B. Farwer, and M. Fisher. Automated Verification of Multi-Agent Programs. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 69–78, L'Aquila, Italy, September 2008.

[7] R. H. Bordini, J. F. Hübner, and R. Vieira. Jason and the Golden Fleece of Agent-Oriented Programming. In Bordini et al. [4], chapter 1, pages 3–37.

[8] M. S. Branicky, V. S. Borkar, and S. Mitter. A Unified Framework for Hybrid Control: Model and Optimal Control Theory. *IEEE Transactions on Automatic Control*, 43(1):31–45, 1998.

[9] M. Dastani, M. B. van Riemsdijk, and J.-J. C. Meyer. Programming Multi-Agent Systems in 3APL. In Bordini et al. [4], chapter 2, pages 39–67.

[10] J. Ferber and O. Gutknecht. A Meta-model for the Analysis and Design of Organizations in Multi-agent Systems. In *Proc. Third International Conference on Multi-Agent Systems (ICMAS)*, pages 128–135, 1998.

[11] M. Fisher, C. Ghidini, and B. Hirsch. Programming groups of rational agents. In *Proc. International Workshop on Computational Logic in Multi-Agent Sytems (CLIMA)*, volume 3259 of *LNAI*. Springer-Verlag, November 2004.

[12] R. Goebel, R. Sanfelice, and A. Teel. Hybrid dynamical systems. *IEEE Control Systems Magazine*, 29(2):28–93, April 2009.

[13] M. Grohe, Y. Gurevich, D. Leinders, N. Schweikardt, J. Tyszkiewicz, and J. V. den Bussche. Database query processing using finite cursor machines. In *Proceedings of the International Conference on Database Theory (ICDT 2007)*, volume 4343 of *LNCS*, pages 284–298, 2007.

[14] Y. Gurevich, D. Leinders, and J. V. den Bussche. A theory of stream queries. In M. Arenas and M. I. Schwatzbach, editors, *DBPL 2007*, volume 4797 of *LNCS*, pages 153–168, 2007.

[15] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: A Model Checker for Hybrid Systems. *International Journal on Software Tools for Technology Transfer*, 1(1-2):110–122, 1997.

[16] J. F. Hübner, J. S. Sichman, and O. Boissier. A Model for the Structural, Functional, and Deontic Specification of Organizations in Multiagent Systems. In *Proc. Sixteenth Brazilian Symposium on Artificial Intelligence (SBIA)*, pages 118–128, London, UK, 2002. Springer-Verlag.

[17] F. F. Ingrand, M. P. Georgeff, and A. S. Rao. An Architecture for Real-Time Reasoning and System Control. *IEEE Expert: Intelligent Systems and Their Applications*, 7(6):34–44, 1992.

[18] N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Aetintemal, M.Cherniack, R. Tibbetts, and S. Zdonik. Towards a streaming sql standard. In *Proceedings of VLDB*, pages 1397–1390, Auckland, New Zealand, August 2008.

[19] M. Kloetzer and C. Belta. A Fully Automated Framework for Control of Linear Systems From Temporal Logic Specifications. *IEEE Transactions on Automatic Control*, 53(1):287–297, 2008.

[20] N. Lincoln and S. 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.

[21] P Tabdada. *Verification and Control of Hybrid Systems: A Symbolic Approach*. Springer, 2009.

[22] D. V. Pynadath, M. Tambe, N. Chauvat, and L. Cavedon. Towards Team-Oriented Programming. In *Intelligent Agents VI — Proc. Sixth International Workshop on Agent Theories, Architectures, and Languages (ATAL)*, volume 1757 of *LNAI*, pages 233–247. Springer-Verlag, 1999.

[23] A. S. Rao and M. Georgeff. BDI Agents: From Theory to Practice. In *Proc. First International Conference on Multi-Agent Systems (ICMAS)*, pages 312–319, San Francisco, USA, June 1995.

[24] S.M. Veres. *Natural language programming of agents and robotic devices: Publishing for humans and machines in sEnglish*. SysBrain Ltd, 2008.

[25] A. Tiwari. Abstractions for hybrid systems. *Formal Methods in Systems Design*, 32:57–83, 2008.

[26] P. Varaiya. Design, simulation, and implementation of hybrid systems. In *Proceedings of the 20th International Conference on Application and Theory of Petri Nets*, pages 1–5, London, UK, 1999. Springer-Verlag.

[27] S. Veres and N. Lincoln. Sliding Mode Control of Autonomous Spacecraft — in sEnglish . In *Proc. Towards Autonomous Robotics Systems (TAROS)*, Edinborough, UK, 2008.

[28] S. Veres and L. Molnar. Publishing Documents on Physical Skills for Intelligent Agents in English. In *Proc. Tenth IASTED International Conference on Artificial Intelligence and Applications (AIA)*, Innsbruck, Austria, 2010.

[29] R. Watson. An application of action theory to the space shuttle. In G. Gupta, editor, *Practical Aspects of Declarative Languages, First International Workshop, PADL '99, San Antonio, Texas, USA, January 18-19, 1999, Proceedings*, volume 1551 of *Lecture Notes in Computer Science*, pages 290–304. Springer, 1999.

[30] M. Wooldridge and N. R. Jennings. Intelligent Agents: Theory and Practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.