Reducing Code Complexity in Hybrid Autonomous Control Systems

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

* Department of Computer Science, University of Liverpool, UK
e-mail: L.A.Dennis@liverpool.ac.uk; mfisher@liverpool.ac.uk; A.Lisitsa@liverpool.ac.uk

** School of Engineering, Univeristy of Southampton, UK
e-mail: S.M.Veres@soton.ac.uk; N.K.Lincoln@soton.ac.uk

Abstract

Modern control systems are limited in their ability to react flexibly and autonomously to changing situations by the complexity inherent in handling situations where many variables are present.

We present an architecture based on a combination of agent programming and hybrid systems for managing high level decisions in such systems. Our preliminary case study concerns satellites maintaining geo-stationary orbits. This case study suggests that the complexity of the code of such a system increases much more slowly in the face of increasing complexity of the scenario, than in a more traditional approach based on finite state machines over controller options.

1 Introduction

Modern control systems are limited in their ability to react flexibly and autonomously to changing situations by the complexity inherent where many variables are present. Additional mechanisms are required to select between low-level controllers when significant changes occur.

We are particularly interested in the control of autonomous satellite systems. Consider the problem of a single satellite attempting to maintain a geostationary orbit. Current satellite control systems maintain orbits using feedback controllers. These implicitly assume that any errors in the orbit will be minor and easily corrected. In situations where more major errors occur, e.g. caused by thruster malfunction, or where changes in mission priorities occur, it is desirable to modify or change the controller or other aspects of the physical system. The complexity of this decision task is a challenge to the imperative programming approach.

There is a long standing tradition, pioneered by the PRS system [14], of using agent languages (and other logic programming approaches – e.g. [25]) to control and reason about such systems. We therefore approach the problem from the perspective of rational agents and hybrid systems. We consider a satellite to be an *agent* which consists of a discrete (rational decision making) engine and a continuous (calculation) engine. The rational engine uses the *Belief-Desire-Intention* (BDI) theory of agency [20] to both generate discrete abstractions from continuous data and to use these abstractions to govern the high level decisions about when to generate new feedback controllers or modify hardware. The continuous, calculational engine is used to derive controllers, perform predictive simulations and to calculate information from continuous data which can be used in forming abstractions.

Our particular aim in this paper is to assess whether it is *beneficial* to use rational agent languages in such hybrid systems. Such an approach surely improves clarity, but does it have other practical benefits? For example, is the code size/complexity significantly improved through this approach? This we aim to explore in the rest of the paper.

1.1 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 [26]. *Rational agents* make decisions in an explainable way, having explicit motivations for the choices made. This makes debugging, diagnosis and the monitoring processes to account for an agent's actions at a high level, much easier to provide.

Following BDI theory [20], we often describe each agent's *beliefs* and *goals*, which in turn determine the agent's *intentions* (a set of actions it intends to take). Such agents make decisions about what action to perform next, given their current beliefs, goals and intentions.

1.2 Control Systems

A fundamental component of control systems technology is the *feedback controller*. This measures, or estimates, the current state of a system through a dynamic model and produces subsequent feedback/feedforward control signals. In many cases difference/differential equations may be used to elegantly manage the process. These equations of complex dynamics make changes to the input values of sub-systems and monitor the outcomes on various sensors.

We are investigating systems that require some decision making system to be integrated with such feedback controller(s). It is by now well established that using a separate *discrete* and *logical* decision making process for this aspect is preferable to greatly extending the basic con-

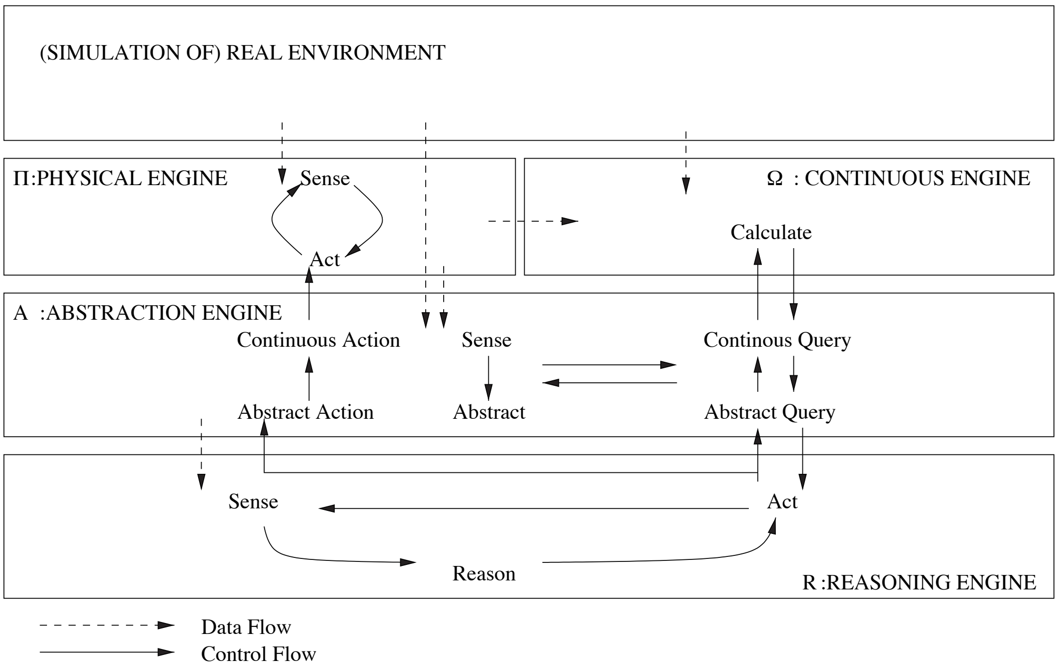

Figure 1. Hybrid Agent Architecture

trol system [1, 2]. Overall systems with these characteristics are often referred to as *hybrid control systems*, in that they integrate discrete, logical decision processes with physical system dynamics.

Unfortunately, the control of hybrid systems using traditional programming methods can become increasingly unwieldy. Often the decision process is represented as an inflexible tree (or graph) of possible situations. Execution then involves tracing through a branch of this (potentially infinite) tree that matches the current situation and then executing the feedback controller (or making other changes to the system) found at the relevant leaf of the tree.

Programming these decisions from state to state is often time-consuming and error prone and can lead to the duplication of code where the same actions need to be taken again but in slightly different situations.

2 Architecture

Our aim is to produce a hybrid system embedding existing technology for generating feedback controllers and configuring satellite systems within a decision making element, based upon agent technologies and theories. The link between the discrete and continuous elements is to be controlled by an *abstraction layer*, which converts data between continuous values appropriate for real time control and discrete values appropriate for reasoning.

Figure 1 shows the basic architecture of 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, in turn, communicates with an agent architecture consisting of an *Abstraction Engine* (*A*) that filters and discretizes information. To do this, the *Abstraction Engine* may a use a *Continuous Engine* (Ω) to make calculations involving continuous information. Finally, the *Rational Engine* (*R*) contains a "Sense-Reason-Act" loop typical of rational agents. Actions involve either calls to the Continuous Engine, for instance to calculate new controllers, or instructions to change the hardware configuration of the Physical Engine. These instructions are passed through the abstraction layer for translation back to continuous values.

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.

3 Scenario: Maintaining Geostationary Orbit with Thruster Failure

A Simulink model of a satellite in a geostationary orbit [18], was implemented. MatLab functions, composed via sEnglish [21, 22, 23], were made available to the continuous part of the agent. These functions are capable of completing trivial computations such as whether a given set of coordinates are within an acceptable distance of the satellite's desired orbital position, `comp_distance`, as well as more complex processing tasks such as computing a fuel optimal return path to a desired orbital position, `plan_approach_to_centre`.

The satellite was simulated with three thruster in an orthogonal X, Y, Z arrangement, and each thruster fed by two fuel lines; one of these fuel lines was redundant enabling the agent to switch fuel lines if the other was determined to be ruptured. Redundant thrusters (up to five in the X direction) were also introduced, allowing the agent to switch to a redundant thruster if both fuel lines appeared broken.

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

A BDI-style rational agent language was developed, based on the Gwendolen programming language [8] and this was used to program both the abstraction and reasoning engines. A key feature of this style of programming is that it allows reactions to several events, or circumstances, to be handled in an interleaved fashion. This permits the system to continue monitoring of incoming data while performing alternative calculations and can react to, in this instance, the malfunction of two thrusters without needing to specify the precise order in which the malfunctions are dealt with.

The agent programming language was implemented in Java and communication between the MatLab and Java parts of the system was managed using sockets. MatLab sent information over the socket consisting of a tag followed by a stream of numbers, on the Java side this was constructed into a predicate to be used by the abstraction engine.

A semantics for interaction between the components of the system was implemented, based on that outlined in [9]. This included a set of *shared beliefs* that were accessible from both the abstraction and reasoning engines.

3.1 The Abstraction Engine

The Abstraction Engine code consisted of two parts: a generic part, used in all examples in the case study, and a specific part, which was modified each time a new thruster was added.

A (cleaned up) version of the generic code is as follows:

Code fragment 3.1 Geostationary Orbit:Abstraction Engine

```
1   +location(L1, L2, L3, L4, L5, L6) : {$\mathcal{B}$ bound_info(V1)} ←
2           calc(comp_distance(L1, L2, L3, L4, L5, L6), Val),
3           +bound_info(Val);
4
5   +bound_info(in) : {$\mathcal{B}$ proximity_to_centre (out)} ←
6           −bound_info(out),
7           $-_\Sigma$ proximity_to_centre (out),
8           $+_\Sigma$ proximity_to_centre (in);
9
10  +bound_info(out) : {$\mathcal{B}$ proximity_to_centre (in)} ←
11          −bound_info(in),
12          $-_\Sigma$ proximity_to_centre (in),
13          $+_\Sigma$ proximity_to_centre (out);
14
15  +!maintain_path : {$\mathcal{B}$ proximity_to_centre (in)} ←
16          run( set_control (maintain));
17  +!execute(P)    : {$\mathcal{B}$ proximity_to_centre (out)} ←
18          run( set_control (P));
19
20  +! plan_approach_to_centre (P) :
21     {$\mathcal{B}$ location (L1, L2, L3, L4, L5, L6)} ←
22     calc( plan_approach_to_centre (L1, L2, L3, L4, L5, L6), P),
23          $+_\Sigma$ plan_approach_to_center (P);
24
25  −broken(X) :
26     {$\mathcal{B}$ thruster_bank_line (X, N, L),
27      $\mathcal{B}$ thruster (X, N, C, V, P), P1 < 1} ←
28          $+_\Sigma$(broken(X));
29
30  + thruster (X, N, C, V, P):
31     {~ $\mathcal{B}$ broken(X),
32      $\mathcal{B}$ thruster_bank_line (X, N, L), P1 < 1} ←
33          $+_\Sigma$ broken(X);
34  + thruster (X, N, C, V, P):
35     {$\mathcal{B}$ broken(X),
36      $\mathcal{B}$ thruster_bank_line (X, N, L), 1 < P1} ←
37          $-_\Sigma$ broken(X).
38
39  +! change_fuel_line (T, 1) :
40     {$\mathcal{B}$ thruster_bank_line (T, B, 1)} ←
41          run( set_valves (T, B, off, off, on, on)),
42          $-_\Sigma$ thruster_bank_line (T, B, 1),
43          $+_\Sigma$ thruster_bank_line (T, B, 2),
44          $-_\Sigma$ broken(T);
45  +!change_bank(T) : {$\mathcal{B}$ thruster_bank_line (T, B, L)} ←
46          B1 is B + 1;
47          run( set_bank (T, B1)),
48          run( set_main (T, B, off )),
49          run( set_main (T, B1, on)),
50          $-_\Sigma$ thruster_bank_line (T, B, L),
51          $+_\Sigma$ thruster_bank_line (T, B1, 1),
52          $-_\Sigma$ broken(T);
```

We here use a standard BDI syntax: $+b$ indicates the addition of a belief; $!g$ indicates a 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. The use of $\mathcal{B}b$ in a plan guard indicates a test that b is believed by the agent.

In addition to regular BDI syntax we use $+_\Sigma b$ and $-_\Sigma b$ to indicate the addition and removal of *shared beliefs* which are used by both the Abstraction and the Reasoning engines. The actions `calc` and `run` trigger communication with the MatLab processes. `calc` requests the calculation of a value from the Continuous Engine achieved by calling an `M-file` with the appropriate name, while `run` activates controls in the physical engine.

The Abstraction Engine performs two functions, converting the data from the Physical Engine and Continuous Engine into a form suitable for reasoning (e.g. the location information is converted into the abstract judgment of whether the satellite is within bounds in lines 1–13 and judgments over whether a thruster is broken are made in lines 30–37) and converting requests from the Reasoning Engine into instructions for the Physical Engine or Continuous Engine (e.g. the `change_bank` request is converted into a sequence of three `run` instructions in lines 45–52).

Requests from the reasoning engine are modelled as goal commitments. So `+!change_fuel_line(T, 1)` indicates that the abstraction engine has received a request from the reasoning engine to change a fuel line.

For example, the code in lines 45–52, describes how to change a thruster in bank T following a request from the reasoning engine, provided the thruster used by the bank is believed to be B. The Physical engine is instructed to set the bank to thruster $B+1$ (`set_bank(T, B1)`), switch off thruster B, switch on thruster $B+1$, and then change the shared beliefs such that it no longer believes that the bank is using thruster B but is using thruster $B+1$. At the same time it removes any beliefs that the thruster is broken.

The code in fragment 3.1 was the same in all versions of the system, but for each additional thruster we had to add code to convert from the input data about that thruster to a more generic predicate. Below is the code used for the 1st thruster in the X bank.

Code fragment 3.2 Geostationary Orbit:X Thruster 1 Code

```
+xthruster1 (L11, L21, P1, Volt1, Curr1):
   {~ ℬ thruster (x, 1, L2, L1, P, V, C)} ←
        +thruster (x, 1, L11, L21, P1, Volt1, Curr1);
+xthruster1 (L11, L21, P1, Volt1, Curr1):
   {ℬ thruster (x, 1, L2, L1, P, V, C)} ←
        −thruster (x, 1, L2, L1, P, V, C),
        +thruster (x, 1, L11, L21, P1, Volt1, Curr1);
```

As can be seen here, the data coming from the Physical Engine tags each thruster's data with a label specific to the thruster (`xthruster1` in this case) but the Abstraction Engine and Reasoning Engine wish to apply the same reasoning to all thrusters and so convert this into a predicate, `thruster`, that is parameterised by the bank (`x` in this case) and the thruster within that bank (`1` in this case). Two cases are needed, depending on whether or not the Abstraction Engine already has a belief about this thruster.

3.2 The Reasoning Engine

The reasoning engine code is as follows and remained the same for all numbers of redundant thrusters:

Code fragment 3.3 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)),
          perform(execute(P)),
          −Σ plan_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 here as we did for the Abstraction Engine: the actions '`perform`' and '`query`' request that the Abstraction Engine forward an instruction to the Reasoning Engine or a calculation to the Continuous Engine respectively.

The architecture allows representations of the high-level decision making aspects of the program in terms of the beliefs and goals within the rational agent and the events it observes. Hence, when the Abstraction Engine observes that the thruster line pressure has dropped inexplicably, it asserts a shared belief that the thruster line is broken. When the Reasoning Engine observes that the thruster line is broken, it then either changes the fuel line, or the thruster bank. This is communicated to the Abstraction Engine, which then sets the appropriate valves and switches.

4 Comparison to Traditional Hybrid Control Systems

As well as constructing a BDI style controller for thruster malfunction we, in parallel, constructed a traditional finite state machine controller using MatLab's stateflow package. As we added additional redundant thrusters we were able to compare how the size of the code increased in the two systems, and hence how both the programming burden and the probability of error increased.

As can be seen from Figure 2, increase in code size for the BDI (i.e. rational agent) system is linear (the additional seven lines of code shown in fragment 3.2 which convert specific thruster predicates into more general predicates) while the FSM approach increases exponentially as more redundant thrusters are added.

5 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.

5.1 Further Case Studies.

We are keen to develop a repertoire of case studies which will provide us with benchmark examples upon which to examine issues such as more sophisticated reasoning tasks, multi-agent systems, forward planning, formal verification and language design.

We have already started work on a more sophisticated study involving a group of satellites operating cooperatively in a low Earth orbit.

5.2 Custom Language.

Currently the BDI language being used for the Abstraction Engine is not as clear as we might like and it may prove that the BDI paradigm is not so appropriate for this abstraction task, since it is not one based around decision making. We are investigating the use of stream processing technologies (from e.g. [3, 15]) and the use of temporal logic statements as a better mechanism for forming abstractions.

We are also interested in investigating other programming languages for the Reasoning Engine – e.g. languages such as *Jason* [6] or 3APL [7] are similar to that currently employed, but are more highly developed and better supported. Alternatively it might be necessary to extend the custom language with, for instance, the concept of a *maintain* goal. This is because much of a satellite's

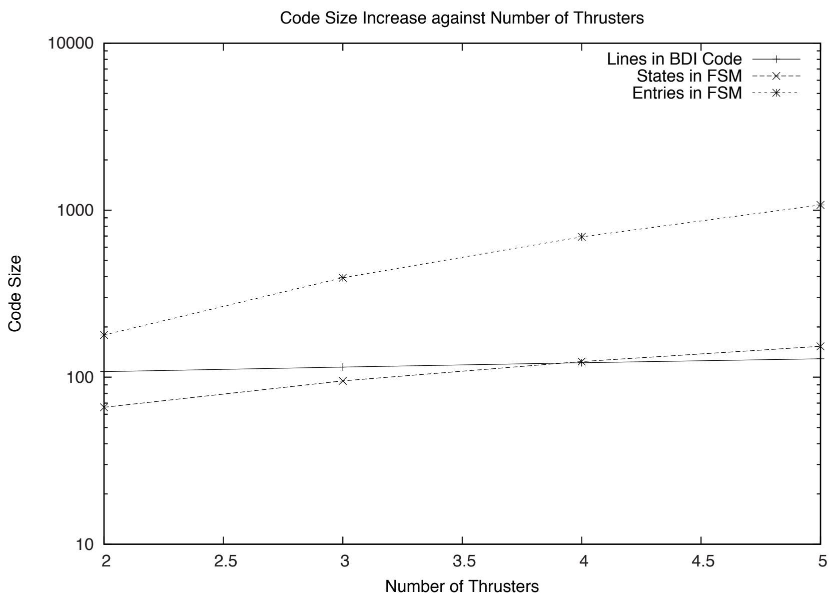

Figure 2. Comparing how Code complexity scales (Logscale on y axis)

operation is most naturally expressed in terms of *maintaining* a state of affairs (such as a remaining on a particular orbital path).

5.3 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 [17]. We are interested in investigating the use of temporal logic and model-checking based approaches to this form of planning for hybrid automata, for example based upon the work of Kloetzer and Belta [16]. We are also interested in the use of simulation as a form of predictive modelling that can assist in the agent's decision making.

Model checking techniques also exist [5] for the verification of autonomous agent programs which could conceivably be applied to the Reasoning Engine. Abstraction techniques would then be required to provide appropriate models of the Continuous Engine and Physical Engine and it might be possible to generate these automatically from the abstraction engine.

There is also a large body of work on the verification of hybrid systems [1, 12] which would allow us to push the boundaries of verification of such systems outside the limits of the Reasoning Engine alone.

5.4 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. For instance, there are realistic scenarios in which one member of a group of satellites loses a particular functionality, meaning that its role within the group must change and the group itself must adapt accordingly. We believe this also provides an interesting application for multi-agent work on groups, teams, roles and organisations [10, 13, 11, 19]. Such instances also provide an interesting test bed for using forward planning and simulation techniques to inform the decision making process.

5.5 Implementation in Hardware

We aim to evaluate our software on a physical satellite simulation environment developed at the University of Southampton [24]. Although this environment constrains the satellites to operate with 5 degrees of freedom, it permits the software to be tested in a real physical environment and thus assess its ability to handle decision-making outside of an entirely virtual implementation. This will be of particular interest when evaluating the predictive simulation aspects of the system, since the ability to handle differences between the simulated result of some action and the actual result of some action, will be a key requirement.

6 Conclusion

This paper has presented a hybrid-style architecture for the autonomous control of satellite systems.

A simple case study is presented demonstrating how this style of programming copes with the increasing complexity of the underlying system better than more traditional approaches to hybrid system programming, based on finite state machines. This reduced complexity follows from the system's ability to make use of parameterised sub-tasks and to specify that sub-tasks are triggered by specific system states and to allow several sub-tasks to be executed in an interleaved fashion.

6.1 Acknowledgment

This work was funded by EPSRC through research grants EP/F037201/1 and EP/F037570.

References

[1] 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.

[2] R. Alur, T. A. Henzinger, G. Lafferriere, George, and G. J. Pappas. Discrete abstractions of hybrid systems. In *Proceedings of the IEEE*, pages 971–984, 2000.

[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, 2005.

[5] 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.

[6] 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.

[7] 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.

[8] L. A. Dennis and B. Farwer. Gwendolen: A BDI Language for Verifiable Agents. In B. Löwe, editor, *Logic and the Simulation of Interaction and Reasoning*, Aberdeen, 2008. AISB. AISB'08 Workshop.

[9] L. A. Dennis, M. Fisher, N. K. Lincoln, A. Lisitsa, and S. M. Veres. Declarative abstractions for agent based hybrid control systems. In A. Omicini, S. Sardina, and W. Vasconcelos, editors, *Declarative Agent Languages and Technologies (DALT'10)*, May 2010.

[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, November 2004.

[12] 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.

[13] 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.

[14] 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.

[15] 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 Very Large Databases*, pages 1397–1390, Auckland, New Zealand, August 2008.

[16] 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.

[17] 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.

[18] M.J. Sidi. *Spacecraft Dynamics and Control: A Practical Engineering Approach*. Cambridge University Press, 2002.

[19] 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, 1999.

[20] 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.

[21] S.M. Veres. *Natural Language Programming of Agents and Robotic Devices: Publishing for Humans and Machines in sEnglish*. SysBrain Ltd, 2008.

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

[23] 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.

[24] S. M. Veres, N. K. Lincoln, and S. B. Gabriel. Testbed for satellite formation flying control system verification. In *Proceedings of the AIAA InfoTech in Aerospace 2007*, Rohnert Park, CA, USA, 2007.

[25] R. Watson. An Application of Action Theory to the Space Shuttle. In G. Gupta, editor, *Proceedings of Practical Aspects of Declarative Languages, First International Workshop (PADL '99)*, volume 1551 of *Lecture Notes in Computer Science*, pages 290–304. Springer, 1999.

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