
Proceedings Paper:
Dennis, L.A., Aitken, J.M., Collenette, J. et al. (2016) Agent-based autonomous systems and abstraction engines: Theory meets practice. In: Alboul , L., Damian, D. and Aitken, J., (eds.) Towards Autonomous Robotic Systems. 17th Annual Conference, Towards Autonomous Robotics, TAROS 2016, 26 June 26 - 1 July, 2016, Sheffield . Lecture Notes in Computer Science (9716). Springer, Cham, pp. 75-86. ISBN: 9783319403786. ISSN: 0302-9743. EISSN: 1611-3349.

https://doi.org/10.1007/978-3-319-40379-3_8

Agent-based Autonomous Systems and Abstraction Engines: Theory meets Practice

Louise A. Dennis[1], Jonathan M. Aitken[2], Joe Collenette[1], Elisa Cucco[1], Maryam Kamali[1], Owen McAree[2], Affan Shaukat[3], Katie Atkinson[1], Yang Gao[3], Sandor Veres[2], and Michael Fisher[1]

[1] Department of Computer Science, University of Liverpool
[2] Department of Autonomous Systems and Control, University of Sheffield
[3] Surrey Space Centre, University of Surrey

Abstract. We report on experiences in the development of hybrid autonomous systems where high-level decisions are made by a rational agent. This rational agent interacts with other sub-systems via an *abstraction engine*. We describe three systems we have developed using the EASS BDI agent programming language and framework which supports this architecture. As a result of these experiences we recommend changes to the theoretical operational semantics that underpins the EASS framework and present a fourth implementation using the new semantics.

1 Introduction

Translating continuous sensor data into abstractions suitable for use in Beliefs-Desires-Intentions (BDI) style agent programming languages is an area of active study and research [7, 6, 17]. Work in [7] provides an architecture for autonomous systems which explicitly includes an *abstraction engine* responsible for translating continuous data into discrete agent beliefs and for reifying actions from the agent into commands for the underlying control system. The architecture includes an operational semantics that specifies the interactions between the various sub-systems.

This paper reports on three autonomous systems based on this architecture, recommends changes to the semantics and then presents a fourth system using the new architecture. In particular we recommend moving away from a view based on a fixed number of sub-systems, to one based on a variable number of communication channels; and abandoning the idea that the abstraction engine works with data present in a logical form.

2 Agents with Abstraction Engines

Hybrid models of control are of increasing popularity in the design and implementation of autonomous systems. In particular there has been interest in systems in which a software agent takes high-level decisions but then invokes lower level controllers to enact those decisions [14, 15, 20]. In many applications

the ability of a *rational* agent to capture the "reasons" for making decisions is important [11]. As the key programming paradigm for rational agents, the BDI model (*Beliefs-Desires-Intention* [18]) is of obvious interest when designing and programming such systems.

A key problem in integrating BDI programs with control systems is that the data generated by sensors is generally continuous in nature where BDI programming languages are generally based on the logic-programming paradigm and prefer to manage information presented in the format of discrete first order logic predicates. A second problem is that the data delivered from sensors often arrives faster than a BDI system is able to process it, particularly when attempting to execute more complex reasoning tasks at the same time.

Work in [7] proposes the architecture shown in Figure 1 in which an *abstraction engine* is inserted between the rational agent (called the *reasoning engine*) and the rest of the system. The abstraction engine is able to rapidly process incoming sensor data and forward only events of interest to the reasoning engine for decision-making purposes. The abstraction engine mediates between the reason-

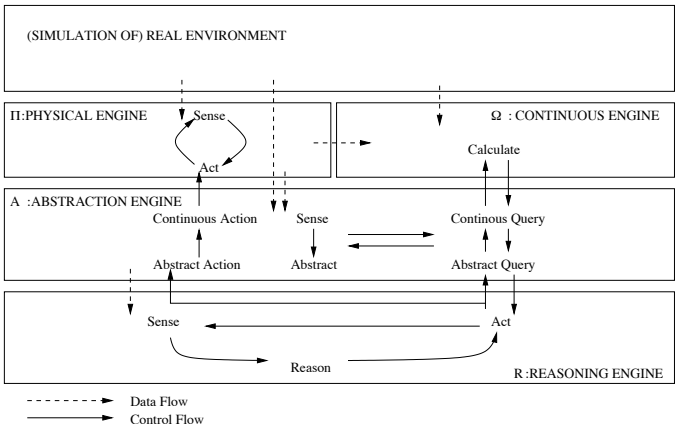

Fig. 1. Abstract Architecture for Agent Based Autonomous Systems

ing engine and a *physical engine* (representing the software control systems and sensors of the underlying autonomous system) and a *continuous engine* which can perform calculation and simulation, specifically, in the original architecture, with a view to path planning and spatial prediction.

The reasoning engine reasons with discrete information. The abstraction engine is responsible, therefore, for abstracting the data from sensors (e.g., real number values, representing distances) to predicates (e.g., *too_close*, etc.). In [7] it is assumed that the abstraction engine works with sensor readings represented in logical form – for instance of the form $distance(2.5)$. A function, fof, is assumed which transforms data from sensors into this representation. The rea-

soning engine makes decisions about actions, but may also request calculations (for instance estimates of whether any collisions are anticipated in the next time window). The abstraction engine is responsible for translating these actions, which it is anticipated will be expressed in a high level fashion, into appropriate commands for either the physical engine or the continuous engine.

A variant of the GWENDOLEN BDI programming language [8], named EASS, implements this architecture. Specifically it provides BDI-based programming structures for both abstraction and reasoning engines (so both are implemented as rational agents). It also provides Java classes for building middleware *environments* for the agents which implement the operational semantics of interaction from [7]. These environments can communicate with external systems by a range of methods (e.g., over sockets or using the Robot Operating System (ROS)[4] [16]) and provide support for transforming data from these systems into first order predicate structures (i.e., providing the *fof* function). We refer to the EASS language and associated environment support as the EASS framework. The EASS framework was used in an extensive case study for the architecture involving satellite and space craft systems [13, 10].

3 Practical Systems

We now discuss three systems built using the EASS framework. The first is a demonstration system in which a robot arm performs *sort and segregate* tasks. The second is a simulation of a convoy of road vehicles. The third is a public engagement activity involving LEGO robots.

3.1 An Autonomous Robotic Arm

The autonomous robotic arm system performs sort and segregate tasks such as waste recycling or nuclear waste management[5] [2, 1]. The system is required to view a set of items on a tray and identify those items. It must determine what should be done with each one (e.g. composted, used for paper recycling or glass recycling, etc) and then move each item to a suitable location.

The system integrates computer vision, a robot arm and agent-based decision making. It is implemented in the Robot Operating System (ROS) [16]. Computer vision identifies items on a tray [19]. These identities and locations are published to a *ROS topic* – a communication channel. The abstraction engine subscribes to this topic and abstracts away the location information informing the reasoning engine what types of object can be seen. The reasoning engine makes decisions about what should be done with each object. These decisions involve, for instance, sending: anything that is plant matter to be composted;

[4] `www.ros.org`

[5] The robotic arm system involves proprietary software developed jointly by the universities of Liverpool, Sheffield and Surrey and National Nuclear Labs. Requests for access to the code or experimental data should be made to Profs Fisher, Veres or Gao.

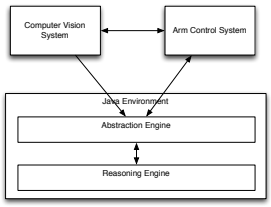

Fig. 2. Architecture for the Robot Arm

paper for recycling; and bricks for landfill. These decisions are published to a different topic by the abstraction engine (adding back the information about object location) to which the robot arm control system subscribes. The control system publishes information about what it is doing which the reasoning engine uses to make sure new instructions are not sent until previous ones have completed.

The architecture is shown in Figure 2. The abstraction and reasoning engine operate as described in [7]. We also show the Java Environment that supports interaction. There is no equivalent to the continuous engine. The physical engine is a combination of the arm control system and the computer vision system. The computer vision and robotic arm sub-systems communicate with each other for coordination when moving objects.

This system has subsequently been extended to deal with the *sort and disrupt* problem [1]. A canister is presented, which must be lifted and placed in a set of v-shaped grooves, before it is opened using a pneumatic cutting tool to inspect the contents which then undergo sort and segregate. This demonstrator consists of a KUKA IIWA arm, with a payload of 7kg. The location of disruption is indicated on a canister via a laser-pen to prevent physical destruction of the test pieces. As well as handling the sort and disrupt task the agent can reason about faulty equipment (simulated using laser-pen failure). The reasoning engine uses a reconfiguration strategy [9] to instruct the arm to use a different laser-pen to complete the task. Similarly the agent reasons about the accuracy and reliability of the information received from the computer vision system.

3.2 Vehicle Convoying

We view an autonomous vehicle convoy as a queue of vehicles in which the first is controlled by a human driver, but subsequent vehicles are controlled autonomously. The autonomously controlled "follower" vehicles maintain a safe distance from the vehicle in front. When a human driving a vehicle wishes to join a convoy they signal their intent to the convoy lead vehicle, together with the position in the convoy they wish to join. Autonomous systems in the lead vehicle then instructs the vehicle that will be behind the new one to drop back, creating a gap for it to move into. When the gap is large enough, the human driver is informed that they may change lane. Once this is achieved, autonomous

systems take control and move all the vehicles to the minimum safe convoying distance. Similar protocols are followed when a driver wishes to leave the convoy.

Maintenance of minimum safe distances between vehicles is handled by two low level control systems. When the convoy is in formation, control is managed using distance sensors and wireless messages from the lead vehicle. These messages inform the convoy when the lead vehicle is braking or accelerating and so allow smooth responses from the whole convoy to these events. This reduces the safe minimum distance to one where fuel efficiency gains are possible. In some situations control uses sensors alone (e.g., during leaving and joining). In these situations the minimum safe distance is larger.

The agent system manages the messaging protocols for leaving and joining, and switches between the control systems for distance maintenance. For instance, if a communication break-down is detected, the agent switches to safe distance control based on sensors alone. The abstraction engine, therefore, is involved primarily in monitoring distance sensors and communication pings.

The system was developed in order to investigate issues in the verification and validation of these convoying systems [12][6]. A simulation of the vehicle control systems was created in MATLab and connected to the the TORCS[7] racing car simulator. The architecture for this system is close to that in [7] with a single physical engine, but still no continuous engine. Several agents connect to the simulator, via the Java environment. The agents use the Java environment for messaging between agents. This architecture is shown in Figure 3.

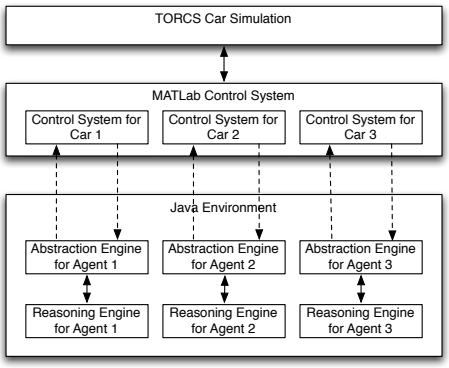

Fig. 3. Architecture for a Simulated Convoying System

Following verification and validation phases based on simulation, we are in the process of transferring the system to Jaguar 4x4 wheeled robots[8] for hardware testing in outdoor situations.

[6] Software available from `github.com/VerifiableAutonomy`

[7] `torcs.sourceforge.net`

[8] `jaguar.drrobot.com`

3.3 LEGO Rovers

The LEGO Rovers system was developed to introduce the concepts of abstraction and rational agent programming to school children[9]. It is used in science clubs by volunteer members of the STEM Ambassador scheme, and has also been used in larger scale events and demonstrations. The activity introduces the user to a teleoperated LEGO robot and asks them to imagine it is a planetary rover. The robot's sensors are explained, the user is shown how the incoming data is abstracted into beliefs such as *obstacle* or *path* using simple thresholds and can then create simple rules, using a GUI, which dictate how the robot should react to the appearance and disappearance of obstacles, etc.

This activity has been through two versions. In the first, the EASS framework was used off-the-shelf with LEGO NXT robots, once again with no continuous engine. The GUI ran on laptops. The system used the leJOS Java-based operating system for Mindstorms robots [4, 5] and classes from this were used for direct communication with the robot. Sensors needed to be polled for data in contrast two our other applications where sensors continuously published data to a stream. While the activity worked well, some issues were observed, particularly the robot's response to rules sometimes lagged more than could be accounted for simply by delays in Bluetooth communication. Investigation suggested that, even when concerned only with processing sensor data, a rational agent was more heavy-weight technology than was required for abstraction. The rational agent used logical reasoning to match plans to events which were then executed to produce abstractions. We discuss an amended version of the system in section 5.

4 An Improved Architecture and Semantics

Practical experience has shown that the provision of a continuous engine is not a fundamental requirement and that it is unrealistic to think of a physical engine as a monolithic entity that encompasses a single input and a single output channel. The use of agent-based abstraction engines that work with first order formulae also seems unnecessary and, in some cases, causes additional inefficiency in the system. This leads us to an adaptation of the semantics in which the purpose of the abstraction engine is to link the reasoning engine to a variety of communication channels which can be viewed as either input or output channels. Input channels include channels which the abstraction engine polls for data so long as it does not execute further until it has received an answer. Communications, like those handled by the continuous engine, in which a request is made for a calculation and, some time later, an answer is received can be handled by placing a request on an output channel and then receiving an answer on an input channel. The abstraction engine (and reasoning engine) can be responsible for matching received answers to requests in an appropriate fashion.

We group these channels into two sets. Π are output channels where the abstraction engine writes information or sends commands to be interpreted by

[9] `www.csc.liv.ac.uk/~lad/legorovers`

other parts of the system. Δ are input channels. Input channels may operate on a request-reply basis but the abstraction engine does nothing between request and reply. Figure 4 shows this new architecture.

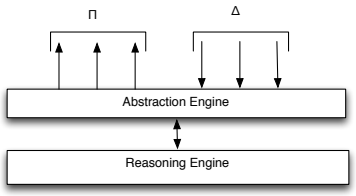

Fig. 4. Refined Architecture for Agent-Based Hybrid Systems with Abstractions

In [7], Π is the physical engine, Ω the continuous engine and Δ is a set of sensor inputs. The semantics also references A (the abstraction engine), R (the reasoning engine) and Σ, Γ and Q, sets of predicates which are used in communication between the abstraction and reasoning engines. Σ is a set of *shared beliefs* which stores abstractions, Γ is a set of commands the reasoning engine has requested for execution and Q is a set of queries from the reasoning engine. The whole system is represented as a tuple. In the modified version of the operational semantics, we no longer consider Ω and Q. Δ and Π are now sets of channels rather than explicit sub-systems. Therefore, we represent the system as a tuple $\langle \Delta, \Pi, A, R, \Sigma, \Gamma \rangle$. The operational semantics specifies a labelled transition system on this tuple. For clarity in discussion, we will sometimes replace parts of this tuple with ellipsis (...) if they are unchanged by a transition.

Most of the rules governing interaction between the abstraction and reasoning engine are largely unchanged. We show these rules in Figure 5 for completeness but do not discuss them further here. We refer the reader to [7] for further discussion and motivation. Three rules involving "queries" (the set Q) intended for the continuous engine have been removed.

We turn our attention to the remaining rules in the semantics. It should be noted that all subsystems may take internal transitions which change their state.

Only one rule involving interaction between abstraction and reasoning engine needs modification. This semantic rule governs transitions the system takes after the reasoning engine has placed a command in the set Γ for the abstraction engine to reify. The original semantics views this reification process as a combination of the transitions taken by the abstraction engine in order to transform the command *and* any subsequent changes to the physical engine. We simplify our view of this so we consider only the change in A when it reads in the command $A \xrightarrow{read_C \gamma} A'$. The abstraction engine may make subsequent internal transitions as it processes the command. The changes to the system when it passes the command on to the physical engine are shown in semantic rule (9). The new

$$\frac{A \xrightarrow{per(\Sigma)} A'}{\langle \ldots, A, R, \Sigma, \ldots \rangle \xrightarrow{per_A(\Sigma)} \langle \ldots, A', R, \Sigma, \ldots \rangle} \quad (1)$$

$$\frac{R \xrightarrow{per(\Sigma)} R'}{\langle \ldots, A, R, \Sigma, \ldots, \rangle \xrightarrow{per_R(\Sigma)} \langle \ldots, A, R', \Sigma, \ldots \rangle} \quad (2)$$

$$\frac{A \xrightarrow{+_\Sigma b} A'}{\langle \ldots, A, R, \Sigma, \ldots \rangle \xrightarrow{+_{\Sigma,A} b} \langle \ldots, A', R, \Sigma \cup \{b\}, \ldots \rangle} \quad (3)$$

$$\frac{A \xrightarrow{-_\Sigma b} A'}{\langle \ldots, A, R, \Sigma, \ldots \rangle \xrightarrow{-_{\Sigma,A} b} \langle \ldots, A', R, \Sigma \backslash \{b\}, \ldots \rangle} \quad (4)$$

$$\frac{R \xrightarrow{+_\Sigma b} R'}{\langle \ldots, R, \ldots \rangle \xrightarrow{+_{\Sigma,R} b} \langle \ldots, R', \Sigma \cup \{b\}, \ldots \rangle} \quad (5)$$

$$\frac{R \xrightarrow{-_\Sigma b} R'}{\langle \ldots, R, \Sigma, \ldots \rangle \xrightarrow{-_{\Sigma,R} b} \langle \ldots, R', \Sigma \backslash \{b\}, \ldots \rangle} \quad (6)$$

$$\frac{R \xrightarrow{do(\gamma)} R'}{\langle \ldots, R, \ldots, \Gamma \rangle \xrightarrow{do_R(\gamma)} \langle \ldots, R', \ldots, \{\gamma\} \cup \Gamma \rangle} \quad (7)$$

Fig. 5. Unchanged Semantic Rules

version of this rule is shown in (8). In this rule when A performs a read on Γ the whole system makes a transition in which A is transformed to A' and γ, the command A has read, is removed from Γ.

$$\frac{\gamma \in \Gamma \qquad A \xrightarrow{read_C(\gamma)} A'}{\langle \ldots, A, \ldots, \Gamma \rangle \xrightarrow{do_A(\gamma)} \langle \ldots\ldots, A', \ldots, \Gamma \backslash \{\gamma\} \rangle} \quad (8)$$

Example Assume a simple BDI agent, A, represented as a tuple, $\langle \mathcal{B}, \mathcal{P}, \mathcal{I} \rangle$ of a set $\mathcal{B}$ of beliefs, a set $\mathcal{P}$ of plans and an intention stack $\mathcal{I}$ of commands to be executed. Assume that when A reads a formula, $\gamma \in \Gamma$ from it places it in $\mathcal{B}$ as $do(\gamma)$. Consider the Lego Rover example and a request from the reasoning engine for the robot to *turn_right* when A has an empty belief set. Before execution of rule 8, $A = \langle \emptyset, \mathcal{P} \rangle$. After the execution of 8, $A = \langle \{do(turn\_right)\}, \mathcal{P}, \mathcal{I} \rangle$.

Our view of the interaction of the abstraction engine with the rest of the system is considerably simplified by the removal of an explicit continuous engine. Two complex semantic rules are removed and we need only consider what happens when the abstraction engine publishes a command to an output channel and when it reads in data from an input channel.

A, the abstraction engine, can place data, γ (assumed to be the processed form of a command issued by the reasoning engine – though this is not en-

forced[10]), on some output channel, π from Π. It does this using the transition $A \xrightarrow{run(\gamma,\pi)} A'$ and we represent the change caused to π when a value is published on it as $\pi \xrightarrow{pub(\gamma)} \pi'$. $\Pi\{\pi/\pi'\}$ is the set Π with π replaced by π'. Rule (9) shows the semantic rule for A publishing a request to π.

$$\frac{\pi \in \Pi \qquad A \xrightarrow{run(\gamma,\pi)} A' \qquad \pi \xrightarrow{pub(\gamma)} \pi'}{\langle \ldots, \Pi, A, \ldots \rangle \xrightarrow{run(\gamma,\pi)} \langle \ldots, \Pi\{\pi/\pi'\}, A', \ldots \rangle} \tag{9}$$

This rule states that if A makes a transition where it publishes some internally generated command, γ, to π, $A \xrightarrow{run(\gamma,\pi)} A'$, then assuming π is one of the output channels and then effect of publishing γ to π is π' then the whole system makes a transition in which A is replaced by A' and π by π'.

Example Returning to our example above, assume that A has plans that reify *turn_right* into a sequence of two commands, one for each engine controlling a wheel on the robot. Internal execution of these plans places these commands on the intention stack[11].

$$A = \langle \{\mathcal{B}, \mathcal{P}, pilot(right.back()) : pilot(left.forward()) : \mathcal{I}\rangle$$

We assume Π consists of two output channels *pilot* (which transmits commands over bluetooth to the leJOS `Pilot` class) and *gui* which sends status information to the GUI controlling the robot. The agent executes the command $pilot(\gamma)$ by placing γ on the *pilot* channel and removing the command from the intention stack. Assume that the *pilot* channel is empty and the GUI channel contains the notification of an obstacle $gui = \{obstacle\}$. So before execution of rule 9, $\Pi = \{\emptyset, \{obstacle\}\}$. After one execution, $\Pi = \{\{right.back()\}, \{obstacle\}\}$ and $A = \langle \{\mathcal{B}, \mathcal{P}, pilot(left.forward()) : \mathcal{I}\rangle$.

Similarly, we indicate the process of abstraction engine, A, reading a value from a channel d by $A \xrightarrow{read(d)} A'$. We represent any change in state on the channel if a value is read from it as $d \xrightarrow{read} d'$. This allows us to simply define a semantics for perception in (10).

$$\frac{d \in \Delta \qquad A \xrightarrow{read(d)} A' \qquad d \xrightarrow{read} d'}{\langle \Delta, \ldots A, \ldots \rangle \xrightarrow{read(d)} \langle \Delta\{d/d'\}, \ldots, A', \ldots \rangle} \tag{10}$$

Example Returning to the example, assume that $A = \langle \emptyset, \mathcal{P}, \mathcal{I} \rangle$ is reads from two input channels: a distance sensor and an RGB colour sensor. Let $\Delta = \{\{50\}, \{\langle 1, 5, 6\rangle\}\}$ (i.e., there is an object 50 centimetres from the robot and the color sensor is looking at a very dark (almost black) surface). Suppose, when A reads data, d, from either channel it removes the reading from the channel and

[10] Particularly since a single command from the reasoning engine can be transformed into a sequence of commands by the abstraction engine.

[11] We use : to indicate concatenation of an element to the top of a stack.

turns it into a belief $distance(d)$ or $rgb(d)$ respectively. If rule 10 is executed and A reads from the distance channel. Then Δ becomes $\{\emptyset, \{\langle 1, 5, 6\rangle\}\}$ and $A = \langle\{distance(50)\}, \mathcal{P}, \mathcal{I}\rangle$.

The main advantage of this new semantics is the simplification of the system by the removal of semantic rules involving calculations and queries and the removal of related components from the state tuple. We also believe it has the pragmatic virtue of representing more accurately the architectures of systems that people actually build using abstraction engines and reasoning agents.

5 Implementation (LEGO Rovers v.2)

The new operational semantics required few changes to the EASS framework, many of which had been implemented incrementally during our experience building the systems in section 3. Support for continuous engines was dropped, simplifying both the language and the Java Environment classes. Support for agent-based abstraction engines was kept, but their use became optional. The implementation had never enforced the use of single input and output channels and the nature of Java meant these were trivial to include as an application required.

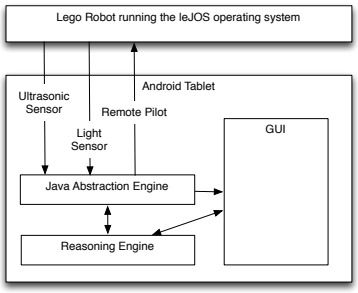

Fig. 6. Architecture for the LEGO Rover System

A second version of the LEGO Rover activity was then developed for LEGO EV3 robots with a GUI running on Android Tablets. This version used the new semantics with the EASS based abstraction engine being replaced by an efficient Java class that handled abstraction of data and reification of commands.

The architecture for the second version is shown in Figure 6. Both abstraction and reasoning engines interact with the GUI which displays the sensor data and the abstractions, and allows the user to define the rules in the reasoning engine. The abstraction engine polls two sensors (ultrasonic and light) for data and publishes commands to the robot. Feedback from demonstrators and teachers suggests that the changes to the activity, including the removal of lag caused by the agent-based abstraction engine, provide a better experience for children.

6 Further Work and Conclusions

In this paper we amended the operational semantics for communication between rational agents, abstraction engines and the rest of a hybrid autonomous system that was presented in [7]. This results in a simplification of the semantics, particularly the removal of an explicit continuous engine and the insistence that abstraction engines handle data in a logical form. The changes are based on extensive practical experience in developing such systems, three of which we have outlined in order both to illustrate the development of agent-based autonomous systems and to motivate the changes to the semantics.

In future, we intend to provide more software support for the implemented framework, particularly for the development of abstraction engines. For instance we aim to expand our abstraction engines, which are currently based almost exclusively on thresholding data, to richer formalisms that would allow abstractions to be formed based on observed complex events and to explore the use of external systems and frameworks to perform the necessary stream processing. In particular it seems desirable to consider abstraction engines either developed in, or incorporating, efficient stream processing tools such as Esper[12] or ETALIS [3].

We now have a refined and simplified a framework for agent-based hybrid systems that use abstraction engines. The framework is well-supported by practical experience in the construction of such systems and represents a strong practical theoretical basis for agent-based hybrid systems.

Acknowledgments

The work in this paper was funded by EPSRC grants Reconfigurable Autonomy (EP/J011770/1, EP/J011843/1, EP/J011916/1) and Verifiable Autonomy (EP/L024845/1, EP/L024942/1, EP/L024861/1) and STFC Grant LEGO Rovers Evolution (ST/M002225/1)

References

1. Aitken, J.M., Shaukat, A., Cucco, E., Dennis, L.A., Veres, S.M., Gao, Y., Fisher, M., Kuo, J.A., Robinson, T., Mort, P.E.: Autonomous nuclear waste management. Robotics and Automation (2016), under Review
2. Aitken, J.M., Veres, S.M., Judge, M.: Adaptation of system configuration under the robot operating system. Proceedings of the 19th world congress of the international federation of automatic control (IFAC) (2014)
3. Anicic, D., Rudolph, S., Fodor, P., Stojanovic, N.: Stream reasoning and complex event processing in etalis. Semant. web 3(4), 397–407 (Oct 2012)
4. Bagnall, B.: Maximum LEGO NXT: Building Robots with Java Brains. Variant Press (2013)
5. Bagnall, B.: Maximum LEGO EV3: Building Robots with Java Brains. Variant Press (2014)

[12] `www.espertech.com`

6. Cranefield, S., Ranathunga, S.: Handling agent perception in heterogeneous distributed systems: A policy-based approach. In: Holvoet, T., Viroli, M. (eds.) Coordination Models and Languages: 17th IFIP WG 6.1 International Conference, COORDINATION 2015. pp. 169–185. Springer International Publishing (2015)
7. Dennis, L.A., Fisher, M., Lincoln, N., Lisitsa, A., Veres, S.M.: Declarative Abstractions for Agent Based Hybrid Control Systems. In: Proc. 8th International Workshop on Declarative Agent Languages and Technologies (DALT). LNCS, vol. 6619, pp. 96–111. Springer (2010)
8. Dennis, L.A., Farwer, B.: Gwendolen: A bdi language for verifiable agents. In: Löwe, B. (ed.) Logic and the Simulation of Interaction and Reasoning. AISB, Aberdeen (2008), AISB'08 Workshop
9. Dennis, L.A., Fisher, M., Aitken, J.M., Veres, S.M., Gao, Y., Shaukat, A., Burroughes, G.: Reconfigurable autonomy. KI-Künstliche Intelligenz 28(3), 199–207 (2014)
10. Dennis, L.A., Fisher, M., Lincoln, N.K., Lisitsa, A., Veres, S.M.: Practical verification of decision-making in agent-based autonomous systems. Automated Software Engineering pp. 1–55 (2014)
11. Fisher, M., Dennis, L.A., Webster, M.P.: Verifying autonomous systems. Commun. ACM 56(9), 84–93 (2013)
12. Kamali, M., Dennis, L.A., McAree, O., Fisher, M., Veres, S.M.: Formal Verification of Autonomous Vehicle Platooning. ArXiv e-prints (Feb 2016), under Review
13. Lincoln, N.K., Veres, S.M., Dennis, L.A., Fisher, M., Lisitsa, A.: Autonomous asteroid exploration by rational agents. Computational Intelligence Magazine 8(4), 25–38 (IEEE, 2013)
14. Muscettola, N., Nayak, P.P., Pell, B., Williams, B.C.: Remote agent: To boldly go where no ai system has gone before. Artif. Intell. 103(1-2), 5–47 (Aug 1998)
15. Patchett, C., Ansell, D.: The development of an advanced autonomous integrated mission system for uninhabited air systems to meet uk airspace requirements. In: Intelligent Systems, Modelling and Simulation (ISMS), 2010 International Conference on. pp. 60–64 (Jan 2010)
16. Quigley, M., Conley, K., Gerkey, B.P., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: ROS: An open-source robot operating system. In: Proc. ICRA Workshop on Open Source Software (2009)
17. Ranathunga, S., Cranefield, S., Purvis, M.: Identifying events taking place in second life virtual environments. Applied Artificial Intelligence 26(1-2), 137–181 (2012)
18. Rao, A.S., Georgeff, M.P.: An Abstract Architecture for Rational Agents. In: Proc. 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR). pp. 439–449 (1992)
19. Shaukat, A., Gao, Y., Kuo, J.A., Bowen, B.A., Mort, P.E.: Visual classification of waste material for nuclear decommissioning. Robotics and Autonomous Systems 75, Part B, 365 – 378 (2016)
20. Wei, C., Hindriks, K.V.: An agent-based cognitive robot architecture. In: Dastani, M., Hübner, J.F., Logan, B. (eds.) Programming Multi-Agent Systems: 10th International Workshop, ProMAS 2012, Valencia, Spain, June 5, 2012, Revised Selected Papers. pp. 54–71. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)