# Architecture for Hybrid Robotic Behavior

David Billington[1], Vladimir Estivill-Castro[2], René Hexel[1], and Andrew Rock[1]

[1] ICT/IIIS, Griffith University, Nathan, QLD, 4111, Australia
{d.billington,v.estivill-castro,r.hexel,a.rock}@griffith.edu.au
www.griffith.edu.au/mipal
[2] Visiting Scholar, Universitat Popeu Fabra, Barcelona, Spain

**Abstract.** Software architectures for agent technology and robots have been polarized between reactive architectures and architectures based on planning and reasoning. Although hybrid architectures have been shown to offer benefits from both, these seem complicated to integrate. In this paper we integrate the reactive nature of finite state machines and the reasoning capabilities of non-monotonic logics to produce intelligent autonomous robots. In particular, we demonstrate this with a robotic poker player. The robotic player integrates vision, sound recognition, motion control and the reasoning to perform competitively as a player in a complex game with incomplete information.

**Keywords:** Non-monotonic logics, finite state machines, software patterns, software engineering, software architecture.

## 1 Introduction

The implementation of the behavior of an autonomous robot is a delicate and sophisticated engineering task. Typically, one would like to produce an architecture that combines a reactive architecture (considered suitable for unknown but simple environments and tasks) with a planning/reasoning approach (suitable for complex worlds which need sophisticated knowledge about the domain and the environment). We present a software architecture that enables behavior designers to specify behaviors using the reactive modeling tool of finite state machines; however, we enable predicates of non-monotonic logics to label transitions. This simplifies the design task because the reasoning component of the logic will resolve conflicts in the description, while the descriptive nature of non-monotonic logics relieves the designer from many of the concerns regarding implementation or the burdens and pitfalls of procedural implementation.

We enable the design of robotic behaviors in terms and formalisms that are accessible to humans. This will become significantly more relevant as collaborative applications for teams of autonomous robots in human environments emerge in the near future. Robots have penetrated carefully controlled industrial environments where they perform well-defined, repetitive tasks. However, the emergence of agent technology and reduced costs of hardware for sensors, batteries, networking, and computational power, suggest robots will be deployed in much

more challenging environments that continuously change, often in unpredictable ways. Today's technologies only handle the complexity of human environments to a very limited extent, but it is expected that in the near future, intelligent integrated systems with the capacity to act within such a complex environment will collaborate with their users in many tasks [1].

There is now an emerging line of research where the human-machine interaction anticipates the ability of all parties to act and co-exist with the environment both in cooperation but also in competition with each other and other collaborative teams [2,3]. Therefore, the area of social robots that interact with people is gaining prevalence [4], together with the area of human-robot interaction [5].

*Cognitive Robotics* aims at programing robots using only high-level actions and relations among actions described by a formal logic. With the situation calculus as foundation, Golog is arguably the most studied high level logical language in this direction [6], and many extensions have appeared in the literature. However, little exists in terms of comparisons and implementations in robots [7]. Non-monotonic logics can and should be incorporated into formalisms for the specification, analysis, and design of behavior [8]. We incorporate them into the central behavioral artifacts provided by state machines. This paper describes how we implemented this approach. A specific non-monotonic logic (*Plausible Logic* [9,10,11] (PL), which is the only non-monotonic logic with an efficient non-looping algorithm [11], was used. This paper describes the infrastructure that enables programmers to design, validate, and deploy graphical models of behavior in autonomous mobile robots. We describe the generic architecture that solves issues of control, interaction with the environment and knowledge representation, and how developers define behaviors using this infrastructure elsewhere [12]. Here, we give an illustration with an application where robots have multi-modal interactions with humans in a game of poker. We believe this will demonstrate the benefits of our approach for intelligent integrated systems that combine capabilities such as reasoning and planning, voice recognition, image analysis, and motion control. Games are considered a suitable methodology for evaluating robot-human interaction [13] while general game playing is the new frontier of artificial intelligence and agent technology [14]. Our robot acts as a multi-modal interface that perceives multiple aspects of the environment and produces diverse types of outputs, such as sounds and gestures, and even acts on the environment. It interacts with a human in a competitive environment with incomplete information.[1] However, the architecture has also been applied to other scenarios where decision-making is complicated because there is incomplete information about a dynamic environment, for example robotic soccer [15] and robots for multi-modal interfaces [16]. Algorithms for signal analysis, computer vision, image processing, and gesture recognition are all involved to capture information from the environment including human speech and human actions.

---

[1] In game theory, making a decision without knowledge of all the values of variables that determine the state of the environment is labeled as *incomplete information*, but also, in the literature of agents this is referred to as an *inaccessible environment*.

## 2    Software Architecture

The most general architecture for an agent interacting with its environment presumes an execution cycle consisting of a phase where the agent collects information through its sensors, decides on an action, and then applies this action [17]. This provides a preliminary answer to the first problem the architecture is to solve — *the interaction problem*, i.e. how does the robot/agent receive information from the environment, and how does it act on it. We use a global architecture that provides a series of services that enable high-level PL descriptions to be compiled and executed directly on board a robot.

### 2.1    Format of the Generic Software Architecture

Our generic software architecture shares many fundamental and structural components with other proposed software architectures for robotics [18]. This will illustrate that our incorporation of non-monotonic reasoning and its tools for visual description and for designing behavior are also applicable to many other architectures. Our architecture has also proven to be a solid framework[2] for development from the software engineering perspective in two important aspects. Modules and subcomponents can be developed by a team of programmers working relatively independently of each other. The architecture facilitates integration and supports a development cycle that consists of regular version refinement and improvement, almost like *Extreme Programming* [20].

How the robot encodes all information collected about the environment, including domain knowledge provided a priori, is the *knowledge representation problem*. Our architecture proposes to have this at several levels of detail. At one level, we use what we call a *whiteboard* where almost all modules write information they have come across, mainly to facilitate module communication. From the perspective of knowledge representation with logics, the *whiteboard* comprises all the facts (including a time-stamp and an author), allowing reasoning associating agency and negotiation. E.g., in soccer we can interpret *whiteboard* messages as "vision believes the ball is in front" and "sonar believes something is ahead", and fuse this to increase our confidence that the ball is ahead.

This basic knowledge representation is complemented with formal logics (and in particular PL) to represent significant issues regarding the domain of operation. Eg. the rules of poker and the strategies to make decisions should be expressed in logic. While this would enable reasoning regarding the action to be performed in a certain state of the environment, the other important aspect is a software engineering concern. Robot control software becomes rather large very quickly, and the analysis of certain situations would be better expressed in a descriptive language close to human understanding enabling much easier analysis of the validity of the knowledge implicitly determining robot's actions.

The nesting, presentation, and meaning of rules encoded in, e.g., C++ or Java becomes complicated and beyond human comprehension. Logic models can be

---

[2] In the sense of 'framework' defined by Larman [19].

created and evolved with mechanisms that abstract the logical inference algorithms, but will facilitate the validation and testing. They should also facilitate their improvement through iterative refinement by humans; much in the way the theory of expert systems approached knowledge elicitation [21].

When addressing the *control problem*, the architecture determines what takes control of the robot's actuators, how a decision on the next action is made, and how progress and the environment is being monitored to adapt the course of action if the current setting is not what we hoped for? There have been many debates between reactive systems and reasoning agents [17]. Reactive agents typically do not build plans, carry out no reasoning, and rarely represent knowledge in any formal logic. Reasoning architectures try to build sophisticated knowledge representations of the environment in order to perform high-level reasoning and to conclude what should be the best action. They may identify goals, build plans, evaluate plan feasibility and monitor progress. Some argue that it is not possible to control a robot in a complex environment unless one applies variants of the subsumption architecture proposed by Brooks. Others [22] suggest a combination of non-monotonic reasoning in reactive systems, placed as "knowledge middleware" to bridge the reactive sensor-based approach and reasoning.

We have a hybrid architecture for the control problem. Like subsumption, it uses priority discrimination. Behaviors are organized to provide a hierarchical structure to the type of behaviors or actions suitable for a certain setting. *External States* characterize some high-level settings in the environment.

Consider a reactive system with behaviors that control the robot and described with state-machines. A behavior consists of *behavior states* and transitions between them. For example, in robotic soccer, a simple ball-chaser behavior could be defined by two states and two transitions. In the state of FOLLOW the robot follows the ball. If the ball goes out of sight, a BALL_NOT_VISIBLE transition moves the behavior to a state of SEARCH where the robot spins around. When the ball becomes visible again, (BALL_VISIBLE transition), the robot changes state back to the state FOLLOW (Fig. 1(a)). This is reactive behavior, because as soon as the conditions that label a transition become effective, the system reacts by performing the actions in the new state.

However, there are elements of planning and reasoning in our agents (robots), as formal logic statements label the transitions between states. Reasoning is performed to establish if a transition should take effect. In the example above, the change to the ball searching state may not be simply the fact that one frame has not identified the ball, but involve a more complex evaluation of other aspects such as path calculation, ball speed and/or distance, as well as the vision module error rate, recent changes of state, or minimum time intervals.

Our architecture has the capability to develop and incorporate complex behaviors while maintaining some grasp on the correctness of them. For this, we enable the programming of a robot in a high-level descriptive language (not just procedural or object-oriented coding in C++ or Java). We describe the knowledge base with a formal logic, and the behaviors are described by a hierarchy of finite state machines. Both of these descriptions are manipulated by
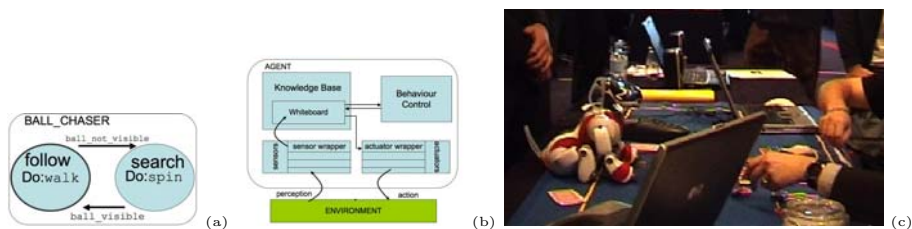
**Fig. 1.** (a) Illustration of simple behavior. (b) Illustration of the architecture operating under an action-perception cycle. (c)The prototype interacting with humans.

programming tools that allow visualization and development by software engineers, while enabling a significant amount of testing in simulators.

## 2.2   Environmental Interaction – The Action-Perception Cycle

The action-perception cycle is perhaps an issue of control, but we discuss it further since it defines the interface with the environment. More importantly, it defines the possibility of the same agent to operate on different hardware (and possibly a different environment). This is because the specific sensors interact with behavior control and the knowledge base through wrappers[3] that can have common interfaces. Consider the diagram in Fig. 1(b). Sensors collect information about the environment and deposit this on the *whiteboard*. The behavior control deposits messages on the *whiteboard* that actuator wrappers grab if the message belongs to them and in turn operate their actuator. A particular sensor (e.g. a camera, gyroscope, sonar sensor, or microphone) may in fact be accessible to a series of software layers. Also, the sensor wrapper may be in itself a sophisticated layered or pipelined architecture (e.g. a vision module representing a pipeline for image segmentation, edge detection and object recognition).

The wrapper allows replacing the physical environment with a virtual environment (e.g. through connecting to a networking socket instead of a real sensor) or portability between different hardware platforms in line with considerations by Kim[24]. In this way, the intelligence (and arguably the personality) of the agent remains the same, operating unaware of a virtual or a physical environment.[4] This idea also illustrates that the majority of the architecture is isolated from the actual physical platform it operates on. Thus, from the software engineering perspective, this means that we can test and verify a large portion of our internal modules by providing simulators for the sensors and the actuators.

## 2.3   Knowledge Representation – The Whiteboard

The *whiteboard* is an abstract data structure where a module can deposit a message. Each message has a type and a time stamp, and is signed by the module

---

[3] The design pattern *wrapper* [19, Page 418] is also named *facade* [23, Page 185].

[4] We have denoted this capability *the matrix* in honor of the series of 3 science-fiction films where the sensors of humans are bridged to a virtual environment, but in the literature on robotics and agents there are other names for this.

that deposits the message. Modules can read all or just the most recent message of a given type. Modules can also retrieve messages sorted by the time-stamp.

The *whiteboard* is inspired by the blackboard architectures for Distributed Artificial Intelligence [25] and by the publish/subscribe and similar software engineering patterns [26,19]). This eliminates the need to create a more complex module communication mechanism. Recall that historically, the first model of communication was a master-slave model best illustrated by the notion of subroutine. While this enabled procedural abstraction, the master must know how and when to call the subroutine. The client-server model provides a step forward due to the independence of flow of control to each module. Nevertheless, the client must be aware of the server interface. The *whiteboard* allows a further level of decoupling. The provider may supply information for unknown consumers who may not even be active. There is no need to be aware of the consumer's interface, only the interface to the *whiteboard* is necessary.

Sensory information in robotic applications is noisy, and may lead to false beliefs about the ground truth. Information with the same time-stamp in the *whiteboard* may in fact not be synchronous in the environment. A typical example of this is the challenge of reading angles for joints in the head of a SONY Aibo and associate them with an image from the camera. A moving head can result in images whose associated angles for head-joints are more than 12 degrees off. Of course this can be eliminated by commanding the head to stay still and then grabbing the image, but in robotic soccer, this would slow down participation in the game beyond any level of competitive performance. Other software modules need to perform the corresponding sensor fusion (data fusion) to build a reasonable picture of the environment. The *whiteboard* can be considered a series (a table) of facts of the form "at time $X$, module $Z$ believed $Y$". Every sensor wrapper will deposit into the *whiteboard* whatever information it can report. Using non-monotonic logic solves the sensor fusion issues by integrating the different beliefs about the environment.

However, there are some challenges with the *whiteboard*. For efficiency reasons, some message may hold pointers (references) to other objects[5] that were not replicated (cloned). E.g. with the images from the vision module, it is too costly (in terms of memory and CPU time) to copy the image pixels. Therefore, the potential exists for the publisher to have deleted the object(s) pointed to by the message. Vision may need to free memory more frequently than other modules as it handles much larger objects. Reference counting offers a solution, but in C++ requires retro-fitting into all affected classes. Nonetheless, the *whiteboard* offers an interface to test references that no longer have a footprint.

## 2.4   Reasoning and Planning

The large list of facts that are available on the *whiteboard* need further processing by reasoning. We have shown that non-monotonic logic can reason about the landmarks reported by vision [27,28]. We have also used this to construct

---

[5]  Instances of classes in the sense of the object-oriented model.

behaviors for triggering alarms if certain conditions are observed in the environment [29]. The first point that the above examples of reasoning illustrate is that the reasoning engine can sit independently of the perception-action cycle.

1. It may sit somewhere in the list of activities that are performed when we respond to the sensor that has brought us to the start of the cycle (e.g. analyzing sightings after vision has processed an image).
2. It may sit as part of some behavior. That is, the conditions that label the transitions in the finite state machine that describes the behavior.

The implementation of PL required the development of a logic programing language (DPL) and a reasoning engine in C. The DPL representation of a set of rules is parsed into a binary representation that can be uploaded onto the robot or used in a simulator for testing. The very same reasoning engine runs on both the robot and the simulator, allowing us to test and debug logic models in a controlled environment outside the robot.

Glue code between the C++ implementation on the robot and the reasoning engine allows us to use predicates that we know can be implemented more efficiently in the native programming language than in logic itself. For example, it may be very laborious to describe integer arithmetic in logic. Asking if an integer is larger than another (which may be necessary for testing if an object is perceived above another) or computing the angle between to perceived objects is best performed in the SONY Aibo in C++. Therefore, we expect to naturally construct logic models for which some predicates consist of collecting a bunch of facts from the *whiteboard* (and thus perhaps from many other modules) and formulating a new fact by posting a new message onto the *whiteboard*.

We have a direct mechanism to place automatically generated C++ code (from a PL) in a *Framework* using a *Template Method*[6]. The template method has three main phases. The first phase (initialization) sets all the variables of the logic to false, ensuring that, by default, all facts are unknown. Then, those predicates that are calculated in the native language are evaluated once using the information on the *whiteboard* (in the simulator, they are set by the operator of the simulator). The resulting values modify the Boolean variables that were initialized in the first step. The logic engine can now be invoked with one or more formulas, and as a result, some new facts can be posted to the *whiteboard*, with the signature of the module using the engine.

## 2.5   Modeling Behaviors

We now provide a description of the tools and our approach to the implementation and the programming of behaviors. As we already mentioned, this was originally implemented on a SONY Aibo for several applications. Easy migration to other platforms has been demonstrated by a Mac OS X (Cocoa) simulator implementation, and a recent port to Aldebaran's Nao robot.

---

[6] Larman describes frameworks using the design pattern *Template Method* [19].

Our sense of subsumption is not strictly in the sense of its proponent [30,31]. The highest level in the hierarchy is what we call *external states*. Actions issued by external states have the highest priority. They are modeled by finite state machines (a programmer can define the external states and its transitions with a finite state machine). We use the term state to reflect again the notion of state in finite state machines, or OMT/UML state diagrams [26,19] but in our nomenclature external state refers to overarching states where *behaviors* take place. An external state is a general top-level, *easily perceived*, long lasting condition of the robot. The robot changes from these external states because an external event produces some stimulus. Our external states correspond to the states designed for the application at hand. These are also what external agents will believe are states of the application (this is what we mean by "easily perceived"). Eg., for playing soccer, the external states were `ready`, `playing`, `off-field`, `booting`, `set-team`, `getting-up`, and `returning-home` (perhaps a better name is "meta-states" or "modes"). External states have the highest controller priority. For example, if gyroscopes tell the robot that it has fallen over, such meta-state components may send instructions and issue joint commands to recover from falling over. These actions may dispute the control over "behaviors" under that meta-state (this is where meta-states take priority over behaviors and is our analogy for "subsumption" [30]). For poker and dominoes, the indication that a player's turn has arrived, will constitute a transition of external states.

`Behavior`s can be implemented under the assumption that pre-conditions of the `State` are met. For example, the designer of a `Behavior` may assume to a significant extent that the robot is standing (because if not the meta-state would have taken over and performed the transition to get it standing, despite the fact that the `Behavior` will still execute if the robot is up-side-down; however, none of the commands will be able to overrule the commands issued by the external state). This point is important because the programmer of the behavior cannot totally assume for example that it will always have images as if standing up. In fact, some variables may have values from an up-side-down robot, and this may create a programming bug in the behavior. The true assumption for behavior designers is that their command would have no effect if the robot is up-side-down. However, programming with the aid of non-monotonic reasoning facilitates operation under these contexts, because again, any sensor information can be incorporated in the from of "sensor X believes Y" and thus, it does not matter that it is inconsistent with other information. The logic model should account for this, we will illustrate this point further with an example.

A `Behavior` is a long lasting activity, (however, the code will only do a little bit of work in every call), a personality that defines what the robot does under this behavior. We must emphasize that a behavior must make sense for each of the possible external states. Behaviors are constructed from actions (or commands) to the wrappers of actuators (if they are basic behaviors) or by sub-behaviors (if they are composite behaviors). Commands and actions are thus detailed motions, while behaviors are intermediate between commands and external states. An example of the benefits of this design is our RoboCup soccer implementation,

by which the state component maintains whether the robots are playing on the blue team or on the red team. Mi-Pal played both halves of a match with the same memory stick even before the league considered this.

Behaviors operate under a `BehaviorControl` container (a singleton[7] object). `BehaviorControl` has different `Behaviors` and decides which to run based on the current external `State`. If the external `State` has changed, then we will call `SwitchBehavior` in `BehaviorControl`; this will select the `Behavior` for that `State`. Thus, fundamentally, states are aware of behaviors, but behaviors should be unaware of states. This should enable the same behavior to be used in a different state. However, behaviors will be aware that an external state transition has occurred. Each external state should have at least one behavior responsible for controlling the robot in that external state. Therefore, behaviors act in parallel. When the external state is established (because of a external-state switch) the method `DoBehavior` in `BehaviorControl` will perform what we call an update of all behaviors in all states, namely `UpdateBehavior()` is called. This makes every behavior aware of changes in the environment, even if they are not "active".

## 3   Illustration and Conclusions

Interpreting the information from sensors (or from other reasoning modules) as beliefs for combining them for common-sense decision making has several benefits. We use the example of a poker player to illustrate that both backward chaining and forward chaining are possible. It is also possible to consider a model of (multiple) agency within a single robot. We illustrate first the use modalities to fuse inputs or expertise from different sub-modules. For example, the ranking of the 169 possible initial hands of Texas-Hold'em Poker is subject to contention. Therefore, perhaps it is better, after recognizing its hold, that the robot represents this fact as "vision believes our hand-strength pre-flop is in the top 50% of all possible hands". Sensors supply contradictory information. In soccer, vision on the Aibo regularly reports a distance to the ball different from what the chest-sonar sensor reports. Thus, a fact such as "sensor X says temperature is above 20" can be modeled as a simple belief using the power of plausible rules. That is, the temperature of the environment in unknown, but we can have a rule that says "if sensor X says temperature is above 20, then usually the temperature is above 20". So evaluations and measurements of the environment, and comparisons for specific constants and values, can be handled with more flexibility. We incorporate sensor information on the same aspect of the environment from more than one sensor by plausible rules and a priority relation on the rules. Similarly, we can incorporate two or more personalities into a poker player that give potentially contradictory advice on the next action. That is, our poker strategy is by default a tight aggressive strategy. However, we monitor the opponent's moves. If we learn that the opponent is tight and passive, we become even tighter and more aggressive.

---

[7] A singleton is the only object of this class in the system [19, Page 413][23, Page 125].

In many interactions for games it is not uncommon to have a state diagram leading to the same set of outputs. For example, the design demands a behavior that has few final options as its completion. This is very common in games like poker or dominoes, where the *external state* of IN_MY_TURN has very few ways to finish the behavior. For example, an active poker player can only *call*, *raise*, or *fold* (and in some situations, only a subset of these options). The corresponding state-diagrams do not have transitions back. When this happens (no transitions back), our tools that evaluate the large system of PL modules can operate in two formats that are analogous to the two types of automatic reasoning (namely forward-chaining and backward chaining). In the forward chaining approach, the execution follows the execution that the finite state machine would follow. That is, the PL expression guarding the initial state is evaluated, and a new state is determined as a result. Once again, the new state is taken as the current state and the expressions guarding transitions are used to determine the next state. This is repeated until a final state is determined. In the poker player, when it is its turn, the robot runs the PL module that determines if the type of opponents is to be changed, and according to the outcome we may now be in a new state to play tighter, it tosses a coin to perhaps bluff, and finally, after deciding on the personality it is going to go with, it uses the PL of the personality it has decided upon to make the final decision whether to call, fold or raise.

Backward chaining is also possible, that is, to execute all the PL modules leading to final states asynchronously (but completely). Then the ones on the previous level synthesize from the current level, and so on, traversing backwards in the transition chain until the initial state is reached. In the the poker player, we may run all styles of play, and get "suggestions" on how to play (as if these were experts on the next move). Then, we synthesize back, so that now we can take the advice from the strategy that best counters the type of opponent we believe we are facing, and directly select among the suggested actions to call, raise, or fold. While backward chaining may seem more wasteful, we have found that typically the run time of the embedded PL reasoning engine is negligible within the action cycle of the robot. We can therefore easily execute all PL modules, rather than selectively execute them. But, we have left this as an optional alternative as we believe both forms may be useful in different behaviors.

Moreover, this also illustrates that the architecture supports a multi-agent model, which is proposed as a post-object-orientation paradigm for software development [17]. The agent model suggests negotiation, perhaps interaction through auctions or regulators. Since behaviors are composed of loosely coupled modules capable of non-monotonic reasoning demonstrates that we can model and support several agents who may arrive at rather contradictory conclusions or bid for possibly incompatible actions. The overall system will mediate between them for a global, well-defined behavior. This was illustrated with the example of a poker player modeled as several personalities that would not necessarily suggest the same action on a particular scenario for the game. Nevertheless, all can execute and are mediated by a non-monotonic reasoning regulator.

# References

1. Wichert, G.V., Lawitzky, G.: Man-machine interaction for robot applications in everyday environments. In: IEEE Int. Workshop on Robot and Human Interactive Communications, Bordeaux, Paris, pp. 343–346 (2001)
2. Clarkson, J., Dowland, B., Cipolla, R.: The use of prototypes in the design of interactive machines. In: Int. Conf. Engineering Design ICED 1997, Tampere (1997)
3. Tzafestas, S., Tzafestas, E.: Human-Machine interaction in intelligent robotic systems: A unifying consideration with implementation examples. J. Intelligent and Robotic Systems 32(2), 119–141 (2001)
4. Breazeal, C., Takanishi, A., Kobayashi, T.: Social robots that interact with people. Springer Handbook of Robotics, pp. 1349–1369. Springer, Berlin (2008)
5. Sankar, N. (ed.): Human-Robot Interaction. I-Tech Education, Vienna (2007)
6. Vassos, S., Levesque, H.: Progression of situation calculus action theories with incomplete information. In: 20th IJCAI, Hyderabad, India, pp. 2024–2029 (2007)
7. Trevizan, F.W., de Barros, L., Corrêa da Silva, F.: Designing logic-based robots. Inteligencia Artificial 10(31), 11–22 (2006)
8. Antoniou, G.: Nonmonotonic Reasoning. MIT Press, Cambridge (1997)
9. Billington, D., Rock, A.: Propositional plausible logic: Introduction and implementation. Studia Logica 67, 243–269 (2001)
10. Rock, A., Billington, D.: An implementation of propositional plausible logic. In: 23rd Australasian Computer Science Conf. of Australian CSC, vol. 22(1), pp. 204–210 (2000)
11. Billington, D.: The proof algorithms of plausible logic form a hierarchy. In: Zhang, S., Jarvis, R. (eds.) AI 2005. LNCS (LNAI), vol. 3809, pp. 796–799. Springer, Heidelberg (2005)
12. Billington, D., Estivill-Castro, V., Hexel, R., Rock, A.: Plausible logic facilitates engineering the behavior of autonomous robots (submitted)
13. Xin, M., Sharlin, E.: Playing games with robots — a method for evaluating human-robot interaction. In: Sankar, N. (ed.) Human-Robot Interaction, Vienna, Austria, ch. 26, pp. 469–480. I-Tech Education and Publishing (2007)
14. Schiffel, S., Thielscher, M.: Fluxplayer: A successful general game player. In: Twenty-Second AAAI Conf. on Artificial Intelligence, pp. 1191–1196. AAAI Press, Menlo Park (2007)
15. Lovell, N.: Machine Vision as the Primary Sensory Input for Mobile, Autonomous Robots. PhD thesis, School of ICT, Griffith University, Nathan, QLD (2006)
16. Estivill-Castro, V., Seymon, S.: Mobile robots for an e-mail interface for people who are blind. In: Lakemeyer, G., Sklar, E., Sorrenti, D.G., Takahashi, T. (eds.) RoboCup 2006: Robot Soccer World Cup X. LNCS, vol. 4434, pp. 338–346. Springer, Heidelberg (2007)
17. Wooldridge, M.: An Introduction to MultiAgent Systems. John Wiley, NY (2002)
18. Liu, T.X.W., Baltes, J.: An intuitive and flexible architecture for intelligent mobile robots. In: 2nd Int. Conf. Autonomous Robots and Agents, NZ, pp. 52–57 (2004)
19. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. Prentice-Hall, NJ (1995)
20. Jeffries, D., Anderson, A., Hendrickson, C.: Extreme Programming Installed. Addison-Wesley, MA (2001)
21. Compton, P., et al.: Ripple down rules: possibilities and limitations. In: 6th Banf AAAI Knowledge Acquisiiton for Knowledge Based Systems Workshop (1991)

22. Heintz, F., Rudol, P., Doherty, P.: Bridging the sense-reasoning gap using dyknow: A knowledge processing middleware framework. In: Hertzberg, J., Beetz, M., Englert, R. (eds.) KI 2007. LNCS, vol. 4667, pp. 460–463. Springer, Heidelberg (2007)
23. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, MA (1995)
24. Kim, J.H., Lee, K.H., Kim, Y.D.: The origin of artificial species: Generic robot. Int. J. Control, Automationa, and Systmes 3(4), 564–570 (2005)
25. Hayes-Roth, B.: A blackboard architecture for control. In: Distributed Artificial Intelligence, pp. 505–540. Morgan Kaufmann, San Francisco (1988)
26. Rumbaugh, J.R., Blaha, M.R., Lorensen, W., Eddy, F., Premerlani, W.: Object-Oriented Modeling and Design. Prentice-Hall, NJ (1991)
27. Billington, D., Estivill-Castro, V., Hexel, R., Rock, A.: Non-monotonic reasoning for localisation in robocup. In: Australasian Conf. on Robotics and Automation, Sydney, Australian Robotics and Automation Association (2005)
28. Billington, D., Estivill-Castro, V., Hexel, R., Rock, A.: Using temporal consistency to improve robot localisation. In: Lakemeyer, G., Sklar, E., Sorrenti, D.G., Takahashi, T. (eds.) RoboCup 2006: Robot Soccer World Cup X. LNCS, vol. 4434, pp. 232–244. Springer, Heidelberg (2007)
29. Billington, D., Estivill-Castro, V., Hexel, R., Rock, A.: Non-monotonic reasoning on board a sony AIBO. In: Lima, P. (ed.) Robotic Soccer, ch. 3, Vienna, Austria, pp. 45–70. I-Tech Education and Publishing (2007)
30. Brooks, R.: Intelligence without reason. In: 12th ICJAI, Sydney, Australia, pp. 569–595. Morgan Kaufmann, San Francisco (1991)
31. Brooks, R.: How to build complete creatures rather than isolated cognitive simulators. In: Architectures for Intelligence, pp. 225–239. Lawrence Erlbaum, Mahwah (1991)