Legolog: Inexpensive Experiments in Cognitive Robotics

Hector J. Levesque and **Maurice Pagnucco**[1]

Abstract. Researchers and instructors of Cognitive Robotics have long lamented the lack of affordable hardware platforms upon which to demonstrate their art. Even with the advent of more recent mobile robotic platforms that have begun to bring prices within reach, the amount of low-level programming that needs to be performed means that significant time must be spent before work can begin on the problems upon which they would like to focus. Recently LEGO® introduced the MINDSTORMS™ Robotics Invention System™—a construction kit equipped with a programmable microprocessor that can accept inputs and control outputs. Together with the vast number of original LEGO® pieces it is now possible to construct all manner of controllable robotic devices.

In this paper we describe our implementation of Legolog; a system that uses a controller from the Golog family of planners to control a MINDSTORMS™ robot. Legolog is capable of dealing with primitive actions, exogenous actions and sensing. Moreover, the Golog controller is easily replaced with an alternate planner. In this way, our aim is to demonstrate that practical cognitive robotics is already within reach, and to provide a (Prolog-based) system for cognitive robotics practitioners to use.
Key words: Cognitive robotics, Golog, LEGO® MINDSTORMS™ Robotics Invention System™.

Advances in mobile robot technology have led to a plethora of such devices becoming commercially available. However, the price of many of these platforms remains prohibitive for many researchers whose work could benefit from their ready availability. This factor is even more pertinent when it comes to the use of mobile robots for pedagogical purposes.

In this paper we describe Legolog,[2] a system developed to allow experimentation with and demonstration of a Golog [6] planner on the LEGO® MINDSTORMS™ Robotics Invention System™ (RIS). We describe the main components of this system and how they interact. While the use of Golog was our initial motivation, one important consideration during the design was to make it possible to substitute Golog with any planner and to run the underlying Prolog on different platforms in a reasonably seamless manner.

The RIS augments the standard LEGO® components with a RCX (Robotic Command Explorer) "brick" containing a microprocessor capable of accepting three inputs and controlling three outputs. Furthermore, the RCX has an infrared port for communicating with other RCXs and an infrared "tower" that can be attached to the serial port of a personal computer. The RCX allows one to build a vast array of different robots of reasonable sophistication and write programs to control them. Currently, the cost of a MINDSTORMS™ kit is about $US 200. One of the main advantages of the RIS over more expensive mobile robot platforms is the ability to quickly and easily modify the design of a robot to try different experimental scenarios.

The main aim of this paper is to *demonstrate how practical robotics can be brought within the reach of cognitive robotics researchers and educators and to provide one such system that is available to be used and easily adapted for such a purpose.*

In the next section we briefly describe the RIS and the particular flavour of Golog used in our implementation of Legolog. In Section 2 we describe the various components of Legolog. A discussion of the main contributions follows in Section 3 with conclusions in Section 4.

1 Platform Components

Legolog is based on two main components: a robot constructed using the RIS and a Golog robotic controller. The RIS grew out of research conducted on the Programmable Brick [17] at the MIT Media Laboratory. Golog has its origins in the situation calculus [11, 16] and has been used to control quite sophisticated robots in real-world environments. We briefly overview these two components before focussing on their specific roles in Legolog.

1.1 LEGO® MINDSTORMS™ Robotics Invention System™

The RIS represents a significant advance on the types of constructions possible with traditional LEGO® kits. At the heart of the RIS is the RCX (Robotic Command Explorer). The RCX contains a Hitachi H8/3297 microprocessor [4]. It allows for up to three input ports and is capable of controlling three output ports. A variety of sensors can be attached to the three input ports. Currently LEGO® has light, temperature, rotation and pushbutton sensors available. However, enthusiasts have shown how to construct inexpensive additional sensors [3][5, Chapter 11]. The outputs are primarily for the control of motors but lights are also available. The RCX is also equipped with an infrared port that can be used to download programs from a standalone computer via an infrared tower attached to the computer's serial port and also allows communication with other RCXs.

The basic idea behind the RIS is that programs are written on a desktop or laptop machine and downloaded to the RCX via the infrared tower which is attached to the machine's serial port. There are a number of options when it comes to programming the RCX. LEGO® provides some basic software—known as *firmware*—implementing a virtual machine that can be downloaded to the RCX. This virtual machine allows for five programs that can be chosen from buttons located on the RCX. Each program may consist of up to 32 variables,

1 Cognitive Robotics Group, Department of Computer Science, University of Toronto, Toronto, ON, M5S 3H5, Canada. Email: {hector, morri}@cs.toronto.edu

2 Legolog is in no way associated with LEGO® or its products.

8 tasks and 9 subroutines. Tasks can execute concurrently with one special task—*main*—being the starting point of any program.

Once the firmware is in place it is possible to use LEGO's own visual programming environment under Windows to write programs for this virtual machine. However, this method is limited in the types of behaviour that can be programmed. A more flexible alternative is an independently developed language called NQC (Not Quite C) [1] that provides a C-like language for programming the RCX firmware. Other firmware programming environments also exist but we shall not investigate these here. We opted for NQC as it appears to provide the best compromise between ease of use and flexibility.

An alternative to the firmware oriented approach is LegOS [13]. This makes use of a C cross-compiler for the Hitachi processor. Programs are a little more sophisticated than NQC since they obviate the need for the underlying firmware but, in so doing, take longer to download. This approach is likely to be more powerful and flexible than the others on offer, and remains one aspect worthy of further investigation.

One other factor that makes the RIS an attractive option is its large user base. Designs for robots and program code are readily available on the world wide web. The main sites for such information are the official RIS site [8] and the LUGNET site [10]. Other pages describe how to manufacture sensors [3], how the RCX functions [14], how to use the infrared transmitter/receiver on the RCX to implement a proximity sensor etc. When one considers the vast number of LEGO® components—from gears and pulleys, pneumatics, motors, etc.—the number and sophistication of devices is quite vast. While the quality of sensors does not yet approach that of more expensive robots, they still permit an adequate level of experimentation.

1.2 Golog

To provide high level control of the robot, we use the programming language Golog [6]. Golog is similar to traditional imperative programming languages, but with three significant differences: (a) the primitive statements in a program are domain-dependent actions to be carried out externally by the robot; (b) the primitive tests in a program are domain-dependent conditions in the world (or fluents) that are changed by the actions of the robot; and, (c) a program may contain non-deterministic choice points where reasoned (non-random) choices need to be made to ensure successful completion of the rest of the program.

To be able to execute a Golog program, an interpreter needs to calculate how fluents are affected by the actions of the robot. To this end, a Golog program is coupled with a basic action theory [7] formulated in the language of the Situation Calculus [11, 16]. A basic action theory contains axioms that specify the initial state of the fluents, and for each fluent, a successor state axiom [15] which specifies how the fluent is changed as the result of performing any action. Basic action theories also contain precondition axioms for each action, stating under what conditions the action can be successfully executed. With these axioms and others in hand, the execution of a Golog program becomes a form of theorem-proving: *find a sequence of robot actions such that it follows from the axioms that these actions go from an initial state to a legal terminating state of the program.*

2 Legolog

Legolog is currently written in Prolog and runs a Golog interpreter augmented with Prolog code to communicate with the RCX via the serial port (to which the infrared tower is attached). The RCX is programmed using NQC. At an abstract level the operation of Legolog is quite simple. The Golog controller determines the next action (if any) for the RCX to execute and sends the appropriate message to the RCX. The RCX acknowledges this message and executes the action. Each primitive action is assumed to take no longer than three seconds to execute.[3] Actions that might take longer to execute can be easily handled and we shall discuss this aspect further in Section 3. The RCX can also signal the occurrence of exogenous actions to Golog. In the model adopted for Legolog, Prolog initiates all communication and therefore must "query" the RCX to determine whether any exogenous action was detected. It is also possible for the RCX to send sensing values to Golog but we have only recently begun to fully utilise this feature.

The structure of a Legolog application can be viewed as in Figure 1. While the implementation consists of a greater number of files than the components indicated in this diagram, this is only to facilitate portability by keeping operating system and Prolog implementation specific code separate from generic Prolog code. However, the overall structure of Legolog is best described at the level of this diagram. A Golog program is run by the Golog interpreter (in this case Indigolog, which will be described in more detail below). Whenever the Indigolog interpreter wishes to execute an action or to determine whether exogenous actions have occurred, it uses Prolog communication predicates to signal the RCX. The NQC program running on the RCX will acknowledge, returning a sensing value or indicate which exogenous action has occurred as appropriate. We shall now describe some of these aspects in further detail.

2.1 RCX User Messages

The RCX is capable of sending and receiving a large variety of messages using a simple error-checking protocol. We shall not delve into the details here but refer the interested reader to the work of Proudfoot [14] on the internals of the RCX (most messages deal with programming the firmware) or to the documentation related to the recent alpha release of RIS 2.0 [9]. Of particular interest are the numbered *user messages* for which NQC (and most other RCX languages) provides primitive functions for sending and receiving. These messages can be used to send numbers in the range 1 to 255. Prolog code was similarly written to provide analogous primitive predicates for sending and receiving RCX numbered message packets. All communication and coordination between Golog and the RCX is achieved in this way. While on the surface this method of communication may appear relatively crude, it functions quite well in practice.

One of the desiderata for our approach was to be able to send arbitrarily large numbers. This facility was mainly seen as a way of sending sensor information back to Golog. We devised a scheme (to be explained shortly) whereby numbers could be transmitted using a series of RCX numbered messages. However, it soon became apparent that such a scheme could also be used to send all manner of information back and forth between Golog and the RCX without tying up large

[3] This length of time is dictated by a characteristic of the infrared tower. It remains able to receive data for only a few seconds after having last transmitted data.

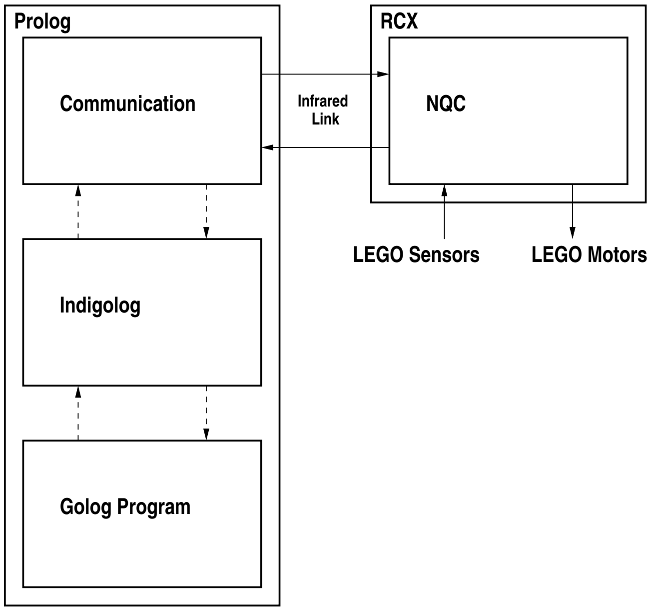

Figure 1. Idealised view of Legolog structure.

portions of the 255 numbered RCX messages at our disposal. This would allow multiple RCXs to be used with no additional effort.

Arbitrary (positive) integers are easily sent using numbered RCX messages over a small range. The method is perhaps easier to explain using an example. We reserve the RCX message numbers 32–47 to stand for the values 0–15; 32 stands for 0, 33 for 1, etc. The RCX message numbers 48–63 also stand for the values 0–15 (respectively) but signify that another "packet" of information is to follow (they have a "continuation" bit set, as it were). In this way, via the values 0–15, we can transmit four bits of information in each RCX message packet. Using the continuation bit, we can send arbitrarily long bit strings, four bits at a time (least significant bits first). Each packet will need to be multiplied by an increasing power of 16 (starting at 16^0) before adding it to the sum of the previous packets (this is easily achieved using a recursive procedure). For example, the value 65 would be sent as two RCX messages: 49 followed by 36 which is 1 (with continuation bit set) followed by 4 (no continuation bit) and, decoded, gives the value $1 + (4 \times 16) = 65$ as desired. This method is used to send action numbers as well as sensing values. To avoid conflict, separate ranges are used for each. When actions are sent and received, a translation table is used to determine the number corresponding to the action (or vice versa).

Each part of a message sent by Prolog must be acknowledged by the RCX (recall that all communication is initiated by Prolog). When integer values (action numbers) are communicated using multiple RCX messages as described above, each individual part is acknowledged by the RCX with a special *continue* message telling Prolog that it can proceed with the next part of the message. When the last part is received from Prolog (it does not have a continuation bit set), the RCX acknowledges with the start of a sensing value (which can be an arbitrary value for non-sensing actions). Likewise, Prolog signals the RCX that it can send the next part using the same special *continue* message. The last part of a sensing value (not having a continuation bit set) need not be acknowledged. If the RCX needs to return an integer value corresponding to the occurrence of an exogenous action, a similar scheme is used with the RCX responding to a special message from Prolog requesting that any exogenous action that has been detected be returned. In this way, message parts are sent in "pairs" with Prolog initiating and RCX "responding". Messages awaiting acknowledgement will time-out after 3.5 seconds and be re-sent.

In addition there are a small number of special message numbers used for various purposes. These messages are used as follows:

- RCX/Golog *continue* message for sending arbitrary integers in multiple parts as described above
- Golog signal to RCX that it should abort what it is doing and reset its state
- Golog request to RCX for any pending exogenous actions (if one has occurred, RCX responds with the appropriate action number)
- RCX reply to Golog that no exogenous action has occurred
- RCX reply to Golog request for the execution of a primitive action that it will require more time (an additional 3 seconds is allowed). This can be sent in lieu of a sensing value. Prolog will send a *continue* message to allow the RCX to subsequently communicate the sensing value.

In this way it can be seen that the 255 numbered RCX messages can be divided into distinct ranges; one for each RCX being used.

2.2 NQC

The NQC code running on the RCX revolves around a simple endless event loop (illustrated in Figure 2) in the main task. At any point in time, the RCX itself is considered to be in one of three possible states:

– OK: all is fine, the RCX is ready to receive and transmit messages
– PANIC: Prolog is not acknowledging RCX attempts at sending a message
– ABORT: reset RCX

When in a PANIC state, the RCX continuously transmits a special PANIC message, ignoring all incoming messages and awaiting to be reset. An abort message causes the RCX to enter the ABORT state where all tasks are stopped (except the main one) and the RCX is reset.

As already noted, all communication is initiated by Prolog. Prolog will signal the RCX whenever it has a (primitive) action to perform and Prolog will query the RCX for pending exogenous actions. When the RCX state is OK, it checks for incoming messages from Golog requesting the execution of a primitive action or querying for the occurrence of exogenous actions. If a primitive action request message arrives, the RCX performs the necessary translation and executes the action before returning a sensing value to Golog. If the primitive action is a non-sensing action, Golog ignores the return value. It is convenient to simply return the value 0 in these cases. If an exogenous action query message is received, the RCX checks whether any user-defined exogenous events have occurred (e.g., button pressed, light sensor reaches a threshold, etc.) and returns the relevant action number or a special reserved message indicating that no exogenous action has taken place. At present, due to the memory limitations of the RCX firmware (and partly due to the lack of sophisticated NQC data structures), only one exogenous action is stored and sent at a time.

In an earlier version of Legolog we had experimented with a protocol allowing the RCX to asynchronously send messages to Prolog but found this unsatisfactory for a number of reasons. Firstly, such a scheme is prone to message “collision” which would require either Prolog or the RCX to defer sending messages. Also, as the infrared tower is only able to receive data for a few seconds after transmitting, in an asynchronous protocol it would be necessary to continually transmit at regular intervals in order to ensure that no RCX transmissions are lost. This becomes cumbersome to manage and does not lead to any significant gain in performance. The difference in the amount of data transmitted under either of these schemes is negligible.

Action requests from Golog result in the execution of a behaviour on the RCX (e.g., start motor, line following, turning, etc.). Each behaviour corresponds to a primitive action and can be coded as a function, subroutine or task.[4] Since primitive actions must complete within three seconds, they are usually written as a function or subroutine. However, it is also possible for the RCX to return a message to Prolog requesting an additional 3 seconds to complete a primitive action. This adds some flexibility in the way that actions are to be dealt with. Actions which have the potential of taking considerable time (such as line following) can be dealt with in two ways. They can be “split” into a primitive action that initiates the task and an exogenous action detected by the RCX signalling that the task has been completed (or a failure has occurred). The resulting Golog program will need to cater for a greater number of (exogenous) actions. Alternatively, the RCX can monitor the elapsed time and continually return requests for an additional 3 seconds until the action is completed. In some cases this latter method is not advisable as the planner has essentially suspended execution at this time, waiting for the primitive action to end and a sensing value to be returned.

```
initialize();
while (true) {
    if (status == ABORT) {
        stopAllBehaviours();
        status = OK;
    }
    if (status == PANIC) {
        panicAction();          //Move around, wiggle, beep, whatever
        SendMessage(PANIC_MESG);
        ReceiveMessage(result);    //Hope for an abort command
    }
    if (status == OK) {
        ReceiveMessage(result);
        if (validActionMesg(result)) {
            startBehaviour(result);
            SendMessage(sensingValue);    //Return sensor value
        } else if (exogRequestMesg(result)) {
            SendMessage(exogAction);
            exogAction = NO_EXOG_ACTION;
        }
    }
}
```

Figure 2. Main event loop in Legolog NQC program.

The NQC code is written in a modular fashion so that the main task and most of the functions called from it need not be changed from one robot to the next. The user needs to supply code for each of the behaviours and provide code to perform the necessary translation from the incoming numbers to call the appropriate behaviour function, subroutine or task. This code is placed in the function startBehaviour.

[4] In NQC, functions are expanded to in-line code. Subroutines are separate procedures. Tasks are procedures that can execute concurrently.

2.3 Indigolog Interpreter

As originally formulated, the Golog framework is completely off-line: we calculate a full sequence of actions to perform, examining the entire program, and only then send the sequence to the robot for execution. To be able to allow sensing information or exogenous actions to help determine which actions should be performed, we need to execute actions in the world, retrieve any sensing results, check for exogenous occurrences, and only then decide on the next actions to perform. This is perhaps just as well, since for large programs, it was somewhat unrealistic to expect to go through the entire program before doing anything at all with the robot.

One of the formulations of Golog lends itself nicely to this type of incremental execution. In this formulation, used to describe a concurrent version of Golog called ConGolog [2], program execution is specified in terms of single steps, using two predicates $Final$ and $Trans$: $Final(\delta,\ s)$ holds if program δ can legally terminate in situation s; $Trans(\delta_1,\ s_1,\ \delta_2,\ s_2)$ holds if one step of δ_1 in situation s_1 leads to situation s_2 with δ_2 remaining to be executed. For off-line execution, we look for a sequence of $Trans$ steps leading to a $Final$ termination. But for an on-line incremental execution, we look for any single action A such that $Trans(\delta,\ s,\ \delta',\ do(A,\ s))$ is entailed, we commit to it, get the robot to execute it, and repeat.

The top level execution loop for an interpreter `indigo(`*program*`,`*situation*`)` based on this idea is as follows:

```
indigo(P,S)  :- exog_occurs(A), !,
                indigo(P,do(A,S)).
indigo(P,S)  :- final(P,S).
indigo(P,S)  :- trans(P,S,P1,S), !,
                indigo(P1,S).
indigo(P,S)  :- trans(P,S,P1,do(A,S)),
                execute(A), !,
                indigo(P1,do(A,S)).
```

In this code, `exog_occurs` is an application-dependent predicate used to check if an exogenous action has occurred, and `execute` is an application-dependent predicate which gets the robot to physically perform the action. (For simplicity, we leave out sensing results.)

We call the resulting dialect of Golog, Indigolog (incremental deterministic Golog). The execution is deterministic in the sense that there is no provision for backtracking once an action has been selected. Yet there may be two actions A_1 and A_2 for which $Trans$ holds and yet only A_2 leads ultimately to a legal termination. To deal with this form of non-determinism, Indigolog contains a search operator Σ where executing $\Sigma(\delta)$ means executing δ making sure that at each step there is a sequence of further steps leading to a legal termination. Unlike a purely off-line execution, however, the search operator allows us to control the amount of lookahead to use at each step. For example, no step would be taken in $\Sigma(\delta_1;\delta_2)$ without looking ahead to the end of δ_2; but with $\Sigma(\delta_1);\delta_2$, we would only lookahead to the end of δ_1. Further details on Indigolog can be found in [18].

2.4 Prolog—RCX Communication

In order to allow for communication between the Indigolog controller (written in Prolog) and the RCX (running NQC code) it was necessary to write some supplementary Prolog communication code. This code is essentially in two parts: a Prolog implementation-specific and operating system specific part containing a small number of primitive predicates, and a system independent part implementing the RCX communication protocol and providing some higher level predicates that distance the user from having to deal with the RCX at the lower level.

The system dependent code implements predicates for the following purposes:

- initialise the serial port (to which the infrared tower is connected) to appropriate baud rate, parity, etc.
- open the serial port for read/write and close it
- obtain a character from the serial port within a specified time-out
- send a character out the serial port
- return the system time in hundredths of seconds
- wait for a specified period of time

With these primitives in place it is possible to write generic (i.e., Prolog and operating system independent) higher-level predicates to communicate with the RCX. The initial version of these primitive predicates was written in SWI-Prolog and subsequently ported to LPA Prolog and ECLiPSe Prolog with a small amount of effort.

The predicates at a higher level of abstraction allow the user to send and receive RCX numbered messages and, more importantly, communicate with the RCX using the protocol outlined in Section 2.1. It must be emphasised that this code is all independent of Golog and could be used by any Prolog program to communicate with the RCX via numbered messages.

With all this in place, enabling Golog to control the RCX requires a small amount of Prolog code. A translation predicate is used to map primitive actions (and exogenous actions) to numbers (from numbers, respectively). These are sent to the RCX (received from the RCX) using predicates implementing the scheme outlined above.[5] It should be noted that this scheme can easily be adapted to a planner other than Golog with a minimum of fuss.

3 Discussion

Legolog has been implemented in such a way as to allow for portability in a number of different directions. Firstly, the Prolog code used for the Indigolog interpreter and, arguably more importantly, that used for communication with the RCX is structured in such a way that it can be quickly and easily rewritten for any Prolog capable of providing or mimicking a small set of low-level primitives. At another level it should be quite simple to "plug-in" another planner in place of Golog and have it interact with the RCX. This has already been successfully demonstrated with a Fluent Calculus [19] planner controlling a MINDSTORMS™ robot on a delivery task.[6] In fact, the code need not interface with a planner at all but could be used as part of any Prolog program to communicate with the RCX. Similarly, although we use NQC to program the RCX it would be quite feasible to use many other RCX programming languages as long as they adhere to the simple protocol outlined here. Of course, Legolog need not even use the RCX but could work with other robot platforms.

One of the attractive features of Legolog is that it provides a physical platform on which to experiment with various reasoning about action problems. We have already mentioned that primitive actions, exogenous actions and sensing actions can be dealt with in the current framework. Primitive actions are assumed to take no longer than three seconds to execute (otherwise Golog will time-out) although extensions can be requested. Actions with longer duration can be easily dealt with using exogenous actions. They are initiated via primitive actions and run as separate tasks on the RCX. When they complete, they signal Golog via an exogenous action. This also allows Golog to deal with any errors that might arise if it so wishes (e.g., the robot becomes lost in its environment). Exogenous actions need not only come from the RCX either. At present Legolog is also able to deal with exogenous actions generated by typing at the keyboard of the desktop or laptop computer. This can be utilised to provide a level of remote control of the robot. Additional sources of exogenous events would also be possible. Other problems that could be addressed include continuous actions, concurrent actions, execution monitoring and multiple agents.

4 Conclusions and Future Work

In this paper we have attempted to demonstrate a flexible system that brings practical cognitive robotics within easy reach of researchers and instructors. The current version of Legolog is written using SWI-Prolog or ECLiPSe Prolog running under the Linux operating system. This version was also ported to LPA-Prolog running DOS on a HP 200LX palmtop computer. However, by providing a small number of system dependent predicates, this code should be easily ported to any Prolog and operating system provided that they can read/write to a serial port where the infrared tower is connected. Another feature of Legolog is that the Golog interpreter can be easily replaced with an alternate planner.

Legolog has been tested using various delivery tasks. The robot design used is capable of following a black line (which need not be straight) using a light sensor until it reaches a marker (landmark), turn $180°$, raise and lower a carrying tray and accept pushbutton presses. A picture of one of these robot designs is shown in Figure 3. This figure shows the RCX with the light sensor attached to the front of the robot which is on the edge of a black line and just about to arrive at a landmark. Figure 4 shows another view of the robot just after it has reached the landmark. The RCX is placed with the infrared transmitter/receiver facing upwards so as to facilitate communication with the infrared tower.

The use of the RIS means that implementors are not stuck with one robot configuration but can experiment with many designs and scenarios. The LEGO® constructions are quite durable and there is much less worry of damage to the robot than with more expensive platforms.

Legolog is only at an initial stage and there is great potential for future extensions. Currently we are working on examples that show the potential of sensing actions and having sensing information returned to Golog. The affordability of the MINDSTORMS™ kit means that it is much easier to experiment with multiple robots. Golog control of multiple RCXs would be of great interest and the current design

---

[5] We also provide routines for debugging in the case that the predicates fail. However, these are Golog dependent. An alternate planner would need to provide its own debugging routines.

[6] The authors would like to thank Michael Thielscher for adapting a Fluent Calculus planner for this purpose.

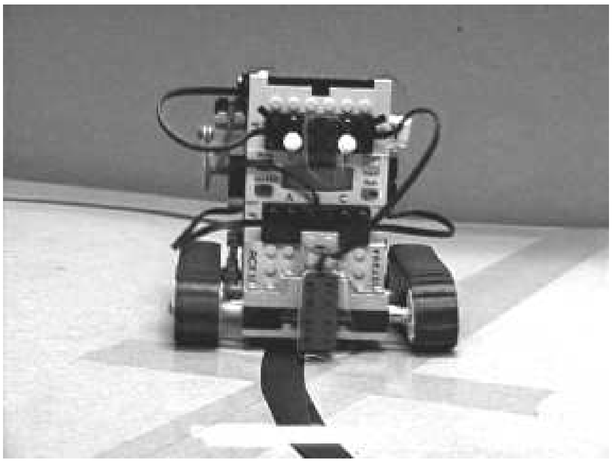

Figure 3. A delivery robot used for experimentation.

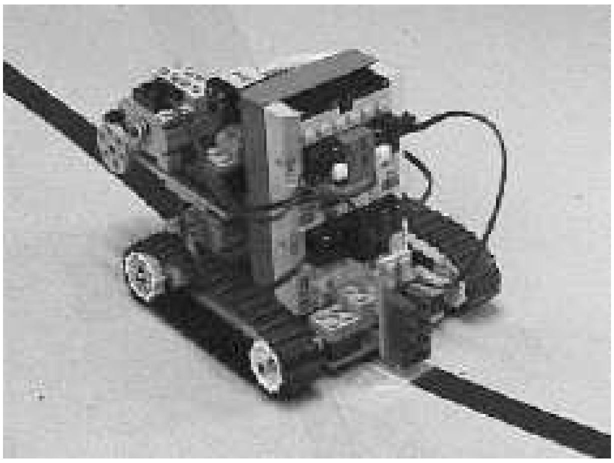

Figure 4. Alternative view of the delivery robot.

of Legolog is such as to accommodate this with minimal effort. Also, communication between RCXs is also of interest.

In conclusion, we would like to invite the cognitive robotics community to make use of Legolog; to adapt their planners to it and experiment with the opportunities that are now within their reach. A version of the Legolog distribution may be obtained from `http://www.cs.toronto.edu/~cogrobo/`.

ACKNOWLEDGEMENTS

The authors would like to thank Michael Thielscher for adapting an early version of Legolog so that it would work with a Fluent Calculus planner written by him in Prolog. They would also like to thank members of the Cognitive Robotics Group at the University of Toronto for their many useful comments and suggestions.

REFERENCES

[1] Dave Baum. *Dave Baum's Definitive Guide to LEGO® MINDSTORMS™*. APress, 2000.

[2] Giuseppe De Giacomo, Yves Lespèrance and Hector J. Levesque. Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in the situation calculus. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'99)*, 1221–1226, Nagoya, August 1997.

[3] Michael Gasperi. MindStorms RCX Sensor Input Page. `http://www.plazaearth.com/usr/gasperi/lego.htm`.

[4] *Hitachi Single-Chip Microcomputer H8/3297—Hardware Manual*. `http://semiconductor.hitachi.com/products/pdf/h33th0142.pdf`. Hitachi, Third Edition, 1999.

[5] Jonathon B. Knudsen. *The Unofficial Guide to LEGO® MINDSTORMS™ Robots*. O'Reilly & Associates, Inc. 1999.

[6] Hector J. Levesque, Raymond Reiter, Yves Lespèrance, Fangzhen Lin and Richard Scherl. GOLOG: A Logic Programming Language for Dynamic Domains. *Journal of Logic Programming*, **31**:59-84, 1997.

[7] Fangzhen Lin and Raymond Reiter. How to progress a database (and why) I: Formal foundations. In *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning (KR'94)*, Morgan-Kaufmann, 1994.

[8] LEGO® MINDSTORMS™ official web site. `http://www.legomindstorms.com/`.

[9] LEGO® MINDSTORMS™ RIS 2.0 Software Developers Kit. `http://www.legomindstorms.com/sdk2/`.

[10] LEGO® user network group. `http://www.lugnet.com/`.

[11] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of Artificial Intelligence. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 4*. Edinburgh University Press, 1969.

[12] Not Quite C web site. `http://www.enteract.com/~dbaum/nqc/`.

[13] Markus Noga. LegOS operating system for MINDSTORMS™. `http://www.noga.de/legOS/`

[14] Kekoa Proudfoot. RCX Internals. `http://graphics.stanford.edu/~kekoa/rcx/`.

[15] Raymond Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, San Diego, CA, 1991.

[16] Raymond Reiter. *Knowledge In Action: Logical Foundations for Describing and Implementing Dynamical Systems.* In press.

[17] Mitchel Resnick, Fred Martin, Randy Sargent, and Brian Silverman. Programmable Bricks: Toys to Think With. *IBM Systems Journal*, **35**(3-4):443–452, 1996. (See also `http://el.www.media.mit.edu/projects/programmable-brick/`.)

[18] Giuseppe de Giacomo, Hector J. Levesque and Sebastian Sardinã. Executing programs over guarded theories. Submitted, 2000.

[19] Michael Thielscher. Ramification and Causality. *Artificial Intelligence*, **89**(1–2):317–364, 1997. (See also `http://pikas.inf.tu-dresden.de/~mit/FC/Tutorial/index.htm`.)