Reactivity in a Logic-Based Robot Programming Framework

Yves Lespérance, Kenneth Tam, and Michael Jenkin

Dept. of Computer Science, York University, Toronto, ON Canada, M3J 1P3
{lesperan,kenneth,jenkin}@cs.yorku.ca

Abstract

A robot must often react to events in its environment and exceptional conditions by suspending or abandoning its current plan and selecting a new plan that is an appropriate response to the event. This paper describes how high-level controllers for robots that are reactive in this sense can conveniently be implemented in ConGolog, a new logic-based robot/agent programming language. Reactivity is achieved by exploiting ConGolog's prioritized concurrent processes and interrupts facilities. The language also provides nondeterministic constructs that support a form of planning. Program execution relies on a declarative domain theory to model the state of the robot and its environment. The approach is illustrated with a mail delivery application.

Introduction

Reactivity is usually understood as having mainly to do with strict constraints on reaction time. As such, much work on the design of reactive agents has involved non-deliberative approaches where behavior is hardwired (Brooks 1986) or produced from compiled universal plans (Schoppers 1987; Rosenschein & Kaelbling 1995). However, there is more to reacting to environmental events or exceptional conditions than reaction time. While some events/conditions can be handled at a low level, e.g., a robot going down a hallway can avoid collision with an oncoming person by slowing down and making local adjustments in its trajectory, others require changes in high-level plans. For example, an obstacle blocking the path of a robot attempting a delivery may mean that the delivery must be rescheduled. Here as in many other cases, the issue is not real-time response. What is required is *reconsideration of the robot's plans in relation to its goals and the changed environmental conditions.* Current plans may need to be suspended or terminated and new plans devised to deal with the exceptional event or condition.

To provide the range of responses required by environmental events and exceptional conditions, i.e. reactivity in the wide sense, the best framework seems to be a hierarchical architecture. Then, urgent conditions can be handled in real-time by a low-level control module, while conditions requiring replanning are handled by a high-level control module that models the environment and task, and manages the generation, selection, and scheduling of plans.

Synthesizing plans at run-time provides great flexibility, but it is often computationally infeasible in complex domains, especially when the agent does not have complete knowledge and there are exogenous events (i.e. actions by other agents or natural events). In (Levesque *et al.* 1997), it was argued that *high-level program execution* was a more practical alternative. The idea, roughly, is that instead of searching for a sequence of actions that takes the robot from an initial state to some goal state, the task is to find a sequence of actions that constitutes a legal execution of some high-level program. By high-level program, we mean one whose primitive instructions are domain-dependent actions of the robot, whose tests involve domain-dependent predicates that are affected by the actions, and whose code may contain nondeterministic choice points where lookahead is necessary to make a choice that leads to successful termination. As in planning, to find a sequence that constitutes a legal execution of a high-level program, one must reason about the preconditions and effects of the actions within the program. However, if the program happens to be almost deterministic, very little searching is required; as more and more nondeterminism is included, the search task begins to resemble traditional planning. Thus, in formulating a high-level program, the user gets to control the search effort required.

In (Levesque *et al.* 1997), *Golog* was proposed as a suitable language for expressing high-level programs for robots and autonomous agents. Golog was used to design a high-level robot control module for a mail delivery application (Tam *et al.* 1997). This module was interfaced to systems providing path planning and low-level motion control, and successfully tested on several different robot platforms, including a Nomad 200, a RWI B21, and a RWI B12.

A limitation of Golog for this kind of applications is that it provides limited support for writing reactive programs. In (De Giacomo, Lespérance, & Levesque 1997), *GonGolog*, an extension of Golog that provides concurrent processes with possibly different priorities as well as interrupts was introduced. In this paper, we try to show that ConGolog is an effective tool for the design of high-level reactive control modules for robotics applications. We provide an example of such a module for a mail delivery application.

ConGolog

As mentioned, our high-level programs contain primitive actions and tests of predicates that are domain-dependent. Moreover, an interpreter for such programs must reason about the preconditions and effects of the actions in the program to find a legal terminating execution. We specify the required domain theories in the situation calculus (McCarthy & Hayes 1979), a language of predicate logic for representing dynamically changing worlds. In this language, a possible world history, which is simply a sequence of actions, is represented by a first order term called a *situation*. The constant S_0 is used to denote the initial situation — that in which no actions (of interest) have yet occurred. There is a distinguished binary function symbol do and the term $do(\alpha, s)$ denotes the situation resulting from action α being performed in situation s. Relations whose truth values vary from situation to situation, called *predicate fluents*, are denoted by predicate symbols taking a situation term as the last argument. For example, $Holding(o, s)$ might mean that the robot is holding object o in situation s. Similarly, functions whose value varies with the situation, *functional fluents*, are represented by function symbols that take a situation argument. The special predicate $Poss(\alpha, s)$ is used to represent the fact that primitive action α is executable in situation s. A domain of application will be specified by theory that includes the following types of axioms:

- Axioms describing the initial situation, S_0.
- Action precondition axioms, one for each primitive action α, which characterizes $Poss(\alpha, s)$.
- Successor state axioms, one for each fluent F, which characterize the conditions under which $F(\vec{x}, do(a, s))$ holds in terms of what holds in situation s; these axioms may be compiled from effects axioms, but provide a solution to the frame problem (Reiter 1991).
- Unique names axioms for the primitive actions.
- Some foundational, domain independent axioms.

Thus, the declarative part of a ConGolog program implementing a high-level controller for a robot will be such a theory.

A ConGolog program also includes a procedural part which specifies the behavior of the robot. This is specified using the following constructs:

α,	primitive action
ϕ?,	wait for a condition[1]
$(\sigma_1; \sigma_2)$,	sequence
$(\sigma_1 \mid \sigma_2)$,	nondeterministic choice between actions
$\pi\, \vec{x}\, [\sigma]$,	nondeterministic choice of arguments
σ^*,	nondeterministic iteration
if ϕ **then** σ_1 **else** σ_2 **endIf**,	conditional
while ϕ **do** σ **endWhile**,	loop
$(\sigma_1 \parallel \sigma_2)$,	concurrent execution
$(\sigma_1 \rangle\!\rangle\, \sigma_2)$,	concurrency with different priorities
$\sigma^{\parallel}$,	concurrent iteration
$<\vec{x} : \phi \rightarrow \sigma>$,	interrupt
proc $\beta(\vec{x})$ σ **endProc**,	procedure definition
$\beta(\vec{t})$,	procedure call
noOp	do nothing

The nondeterministic constructs include $(\sigma_1 \mid \sigma_2)$, which nondeterministically choses between programs σ_1 and σ_2, $\pi\vec{x}[\sigma]$, which nondeterministically picks a binding for the variables $\vec{x}$ and performs the program σ for this binding of $\vec{x}$, and σ^*, which means performing σ zero or more times. Concurrent processes are modeled as interleavings of the primitive actions involved. A process may become blocked when it reaches a primitive action whose preconditions are false or a wait action ϕ? whose condition ϕ is false. Then, execution of the program may continue provided another process executes next. In $(\sigma_1 \rangle\!\rangle\, \sigma_2)$, σ_1 has higher priority than σ_2, and σ_2 may only execute when σ_1 is done or blocked. $\sigma^{\parallel}$ is like nondeterministic iteration σ^*, but the instances of σ are executed concurrently rather than in sequence. Finally, an interrupt $<\vec{x} : \phi \rightarrow \sigma>$ has variables $\vec{x}$, a trigger condition ϕ, and a body σ. If the interrupt gets control from higher priority processes and the condition ϕ is true for some binding of the variables, the interrupt triggers and the body is executed with the variables taking these values. Once the body completes execution, the interrupt may trigger again. With interrupts, it is easy to write programs that are reactive in that they will suspend whatever task they are doing to handle given conditions as they arise. A more detailed description of ConGolog and a formal semantics appear in (De Giacomo, Lespérance, & Levesque 1997). We give an example ConGolog program in section 4.

A prototype ConGolog interpreter has been implemented in Prolog. This implementation requires that the axioms in the program's domain theory be expressible as Prolog clauses; note that this is a limitation of the implementation, not the framework.

In applications areas such as robotics, we want to use ConGolog to program an embedded system. The system must sense conditions in its environment and update its theory appropriately *as it is executing* the ConGolog control program.[2] This requires adapting the high-level program execution model presented earlier: the interpreter cannot simply search all the way to a final situation of the program. An adapted model involving incremental high-level program execution is developed in (De Giacomo & Levesque 1998). However in this paper, we sidestep these issues by making two simplifying assumptions:

1. that the interpreter immediately commits to and executes any primitive action it reaches when its preconditions are satisfied, and
2. that there is a set of exogenous events detectable by the system's sensors (e.g. a mail pick up request is received or the robot has arrived at the current destination) and

[1] Here, ϕ stands for a situation calculus formula with all situation arguments suppressed; $\phi(s)$ will denote the formula obtained by restoring situation variable s to all fluents appearing in ϕ.

[2] Here, the environment is anything outside the ConGolog control module about which information must be maintained; so the sensing might only involve reading messages from another module through a communication socket.

that the environment is continuously monitored for these; whenever such an exogenous event is detected to have occurred, it is immediately inserted in the execution.

We can get away with this because our application program performs very little search and the exogenous events involved are easy to detect.

Interfacing the High-Level Control Module

As mentioned earlier, we use a hierarchical architecture to provide both real-time response as well as high-level plan reconsideration when appropriate. At the lowest level, we have a reactive control system that performs time-critical tasks such as collision avoidance and straight line path execution. In a middle layer, we have a set of components that support navigation through path planning, map building and/or maintenance, keeping track of the robot's position, etc. and support path following by interacting with the low-level control module. On top of this, there is the ConGolog-based control module that supports high-level plan execution to accomplish the robot's tasks; this level treats navigation more or less as a black box.

In this section, we describe how the ConGolog-based high-level control module is interfaced to rest of the architecture. The high-level control module needs to run asynchronously with the rest of the architecture so that other tasks can be attended to while the robot is navigating towards a destination. To support this and allow for interaction between the high-level control module and the navigation module, we have defined the following simple model. With respect to navigation, the robot is viewed by the high-level control module as always being in one of the following set of states:

$$RS = \{Idle, Moving, Reached, Stuck, Frozen\}.$$

The current robot state is represented by the functional fluent $robotState(s)$. The robot's default state is $Idle$; when in this state, the robot is not moving towards a destination, but collision avoidance is turned on and the robot may move locally to avoid oncoming bodies. With the robot in $Idle$ state, the high-level control module may execute the primitive action $startGoTo(place)$; this changes the robot's state to $Moving$ and causes the navigation module to attempt to move the robot to $place$. If and when the robot reaches the destination, the navigation module generates the exogenous event $reachDest$, which changes the robot's state to $Reached$. If on the other hand the navigation module encounters obstacles it cannot get around and finds the destination unreachable, then it generates the exogenous event $getStuck$, which changes the robot's state to $Stuck$. In any state, the high-level control module may execute the primitive action $resetRobot$, which aborts any navigation that may be under way and returns the robot to $Idle$ state. Finally, there is the $Frozen$ state where collision avoidance is disabled and the robot will not move even if something approaches it; this is useful when the robot is picking up or dropping off things; humans may reach into the robot's carrying bins without it moving away. All other actions leave the robot's state unchanged. This is specified in the following successor state axiom for the $robotState$ fluent:

$$\begin{array}{l} robotState(do(a,s)) = i \equiv \\ \quad \exists p\, a = startGoTo(p) \wedge i = Moving \vee \\ \quad a = reachDest \wedge i = Reached \vee \\ \quad a = getStuck \wedge i = Stuck \vee \\ \quad a = resetRobot \wedge i = Idle \vee \\ \quad a = frezeRobot \wedge i = Frozen \vee \\ \quad i = robotState(s) \wedge \forall p\, a \neq startGoTo(p) \wedge \\ \qquad a \neq reachDest \wedge a \neq getStuck \wedge \\ \qquad a \neq resetRobot \wedge a \neq frezeRobot \end{array}$$

We also have precondition axioms that specify when these primitive actions and exogenous events are possible. For example, the following says that the action of directing the robot to start moving toward a destination p is possible whenever the robot is in $Idle$ state:

$$Poss(startGoTo(p), s) \equiv robotState(s) = Idle$$

We omit the other precondition axioms as they are obvious from the model description.

We also use two additional functional fluents: $robotDestination(s)$ refers to the last destination the robot was set in motion towards, and $robotPlace(s)$ refers to the current location of robot as determined from the model. Their successor state axioms are:

$$\begin{array}{l} robotDestination(do(a,s)) = p \equiv \\ \quad a = startGoTo(p) \vee \\ \quad p = robotDestination(s) \wedge \forall p\, a \neq startGoTo(p) \end{array}$$

$$\begin{array}{l} robotPlace(do(a,s)) = p \equiv \\ \quad \exists p'\, a = startGoTo(p') \wedge p = Unknown \vee \\ \quad a = reachDest \wedge p = robotDestination(s) \vee \\ \quad p = robotPlace(s) \wedge \\ \qquad \forall p\, a \neq startGoTo(p) \wedge a \neq reachDest \end{array}$$

A Mail Delivery Example

To test our approach, we have implemented a simple mail delivery application. The high-level control module for the application must react to two kinds of exogenous events:

- new shipment orders, which are represented by the event $orderShipment(sender, recipient, priority)$, and
- signals from the navigation module, namely the $reachDest$ event announcing that the destination has been reached and the $getStuck$ event announcing that the robot has failed to reach its destination.

The first kind is typical of the communication interactions a robot may have with its environment, while the second kind is typical of the control interactions a task-level module may have with the rest of the robot's architecture. To require more reactivity from the robot, we assume that shipment orders come with different priority levels and that the system must interrupt service of a lower priority order when a higher priority one comes in. Also, we want the robot to make a certain number of attempts to get to a customer's mailbox as some of the obstacles it runs into may be temporary. This is handled by assigning a certain amount of credit to customers initially and reducing their credit when an attempt to go to their mailbox fails. When customers run out of credit, they are suspended and shipments sent to them are returned to the sender when possible.

In addition to the navigation primitive actions and exogenous events already described, the application uses the following primitive actions:

$ackOrder(n)$	acknowledge reception of servable order
$declineOrder(n)$	decline an unservable order
$pickUpShipment(n)$	pick up shipment n
$dropOffShipment(n)$	drop off shipment n
$cancelOrder(n)$	cancel an unservable order
$reduceCredit(c)$	reduce customer c's credit
$notifyStopServing(c)$	notify unreachable customer

Note that shipment orders are identified by a number n that is assigned from a counter when the $orderShipment$ event occurs. We have precondition axioms for these primitive actions, for example:

$$\begin{aligned}&Poss(pickUpShipment(n), s) \equiv\\ &\quad orderState(n, s) = ToPickUp \wedge\\ &\quad robotPlace(s) = mailbox(sender(n, s))\end{aligned}$$

The primitive fluents for the application are:

$orderState(n, s) = i$	order n is in state i
$sender(n, s) = c$	sender of order n is c
$recipient(n, s) = c$	recipient of order n is c
$orderPrio(n, s) = p$	priority of order n is p
$orderCtr(s) = n$	counter for orders arriving
$credit(c, s) = k$	customer c has credit k
$Suspended(c, s)$	service to customer c is suspended

We have successor state axioms for these fluents. For example, the state of an order starts out as $NonExistent$, then changes to $JustIn$ when the $orderShipment$ event occurs, etc.; the following successor state axiom specifies this:

$$\begin{aligned}&orderState(n, do(a, s)) = i \equiv\\ &\quad \exists c, r, p\; a = orderShipment(c, r, p) \wedge orderCtr = n\\ &\qquad \wedge\, i = JustIn \,\vee\\ &\quad a = ackOrder(n) \wedge i = ToPickUp \,\vee\\ &\quad a = pickUpShipment(n) \wedge i = OnBoard \,\vee\\ &\quad a = dropOffShipment(n) \wedge\\ &\qquad robotPlace(s) = mailbox(recipient(n, s))\\ &\qquad \wedge\, i = Delivered \,\vee\\ &\quad a = dropOffShipment(n) \wedge\\ &\qquad robotPlace(s) = mailbox(sender(n, s))\\ &\qquad \wedge\, i = Returned \,\vee\\ &\quad a = dropOffShipment(n) \wedge\\ &\qquad robotPlace(s) = CentralOffice\\ &\qquad \wedge\, i = AtCentralOffice \,\vee\\ &\quad a = cancelOrder(n) \wedge i = Cancelled \,\vee\\ &\quad a = declineOrder(n) \wedge i = Declined \,\vee\\ &\quad i = orderState(n, s) \wedge\\ &\quad \neg(\exists c, r, p\; a = orderShipment(c, r, p) \wedge orderCtr = n)\\ &\qquad \wedge\, a \neq ackOrder(n) \wedge a \neq pickUpShipment(n) \wedge\\ &\qquad a \neq dropOffShipment(n) \wedge a \neq cancelOrder(n)\end{aligned}$$

We omit the rest of the successor state and action precondition axioms for space reasons.

The initial state of the domain might be specified by the following axioms:

$$\begin{aligned}&Customer(Yves) \quad Customer(Ken)\\ &Customer(Hector) \quad Customer(Michael)\\ &Customer(c) \supset credit(c, S_0) = 3\\ &orderCtr(S_0) = 0\\ &orderState(n, S_0) = NonExistent\\ &robotState(S_0) = Idle\\ &robotPlace(S_0) = CentralOffice\end{aligned}$$

Let us now specify the behavior of our robot using a ConGolog program. Exogenous events are handled using prioritized interrupts. The main control procedure concurrently executes four interrupts at different priorities:

```
proc $mainControl$
  $\langle\, n : orderState(n) = JustIn$
      $\rightarrow handleNewOrder(n) \,\rangle$
  $\gg$
  $\langle\, n : (orderState(n) = ToPickUp$
        $\wedge\, Suspended(sender(n)))$
      $\rightarrow cancelOrder(n) \,\rangle$
  $\|$
  $\langle\, n : (orderState(n) = ToPickUp$
        $\vee\, orderState(n) = OnBoard$
        $\vee\, robotPlace \neq CentralOffice)$
      $\rightarrow robotMotionControl \,\rangle$
  $\gg$
  $\langle\, robotState = Moving \rightarrow$ noOp $\rangle$
endProc
```

The top priority interrupt takes care of acknowledging or declining new shipment orders. This ensures that customers get fast feedback when they make an order. At the next level of priority, we have two other interrupts, one that takes care of cancelling orders whose senders have been suspended service, and another that controls the robot's motion. At the lowest priority level, we have an interrupt with an empty body that prevents the program from terminating when the robot is in motion and all other threads are blocked.

The top priority interrupt deals with a new shipment order n by executing the following procedure:

```
proc $handleNewOrder(n)$
  if $Suspended(sender(n)) \vee Suspended(recipient(n))$
    then $declineOrder(n)$
  else
    $ackOrder(n);$
    if $robotState = Moving \wedge$
        $orderPrio(n) > curOrderPrio$ then
      $resetRobot$     % abort current service
    endIf
  endIf
endProc
```

This sends a rejection notice to customers making an order whose sender or recipient is suspended; otherwise an acknowledgement is sent. In addition, when the new shipment order has higher priority than the order currently being served, the robot's motion is aborted, causing a reevaluation of which order to serve ($curOrderPrio$ is a defined fluent whose definition appears below).

The second interrupt in $mainControl$ handles the cancellation of orders when the sender gets suspended; its body executes the primitive action $cancelOrder(n)$.

The third interrupt in $mainControl$ handles the robot's navigation, pick ups, and deliveries. When the interrupt's condition is satisfied, the following procedure is called:

```
**proc** $robotMotionControl$
  **if** $\exists c\, CustToServe(c)$ **then** $tryServeCustomer$
  **else** $tryToWrapUp$;
  **endIf**
**endProc**
```

This tries to serve a customer when there is one to be served and tries to return to the central office and wrap up otherwise. $CustToServe(c,s)$ is a defined fluent:

$$\begin{array}{l} CustToServe(c,s) \overset{\text{def}}{=} \exists n[\\ \quad (orderState(n,s) = ToPickUp \wedge sender(n,s) = c \\ \qquad \wedge \neg Suspended(recipient(n,s),s)) \vee \\ \quad (orderState(n,s) = OnBoard \wedge (recipient(n,s) = c \\ \qquad \vee\, sender(n,s) = c \wedge Suspended(recipient(n,s),s)))] \\ \quad \wedge \neg Suspended(c,s) \end{array}$$

To try to serve a customer, we execute the following:

```
**proc** $tryServeCustomer$
  $\pi\, c\, [BestCustToServe(c)?;$
     $startGoTo(mailbox(c))$;
     $(robotState \neq Moving)?$;
     **if** $robotState = Reached$ **then**
        $freezeRobot$;
        $dropOffShipmentsTo(c)$;
        $pickUpShipmentsFrom(c)$;
        $resetRobot$
     **else if** $robotState = Stuck$ **then**
          $resetRobot$;    % abandon attempt
          $handleServiceFailure(c)$
     % else when service aborted nothing more to do
     **endIf** ]
**endProc**
```

This first picks one of the best customers to serve, directs the robot to start navigating towards the customer's mailbox, and waits until the robot halts. If the robot reaches the customer's mailbox, then shipments for the customer are dropped off and shipments from him/her are picked up. If on the other hand, the robot halts because it got stuck, the $handleServiceFailure$ procedure is executed. Finally, if the robot halts because a higher priority order came in and the top priority interrupt executed a $resetRobot$, then there is nothing more to be done. $BestCustToServe(c,s)$ is a defined fluent that captures all of the robot's order scheduling criteria:

$$\begin{array}{l} BestCustToServe(c,s) \overset{\text{def}}{=} CustToServe(c,s) \wedge \\ \quad custPriority(c,s) = maxCustPriority(s) \wedge \\ \quad credit(c,s) = maxCreditFor(maxCustPriority(s),s) \end{array}$$

$$\begin{array}{l} custPriority(c,s) = p \overset{\text{def}}{=} \\ \quad \exists n\, OrderForCustAtPrio(n,c,p,s) \wedge \\ \quad \forall n',p'(OrderForCustAtPrio(n',c,p',s) \supset p' \leq p) \end{array}$$

$$\begin{array}{l} OrderForCustAtPrio(n,c,p,s) \overset{\text{def}}{=} \\ \quad orderState(n,s) = ToPickUp \wedge sender(n,s) = c \wedge \\ \qquad orderPrio(n,s) = p \vee \\ \quad orderState(n,s) = OnBoard \wedge orderPrio(n,s) = p \wedge \\ \qquad (recipient(n,s) = c \vee \\ \qquad\; sender(n,s) = c \wedge Suspended(recipient(n,s),s)) \end{array}$$

$$\begin{array}{l} maxCustPriority(s) = p \overset{\text{def}}{=} \exists c\, custPriority(c,s) = p \\ \qquad\quad \wedge\, \forall c'\, custPriority(c',s) \leq p \end{array}$$

$$\begin{array}{l} maxCreditFor(p,s) = k \overset{\text{def}}{=} \\ \quad \exists c[custPriority(c,s) = p \wedge credit(c,s) = k \wedge \\ \qquad \forall c'(custPriority(c',s) = p \supset credit(c',s) \leq k)] \end{array}$$

This essentially says that the best customers to serve are those that have the highest credit among those having the highest priority orders. We can now also define the priority of the order currently being served as follows:

$$\begin{array}{l} curOrderPrio(s) = p \overset{\text{def}}{=} \\ \quad \forall c[robotState(s) = Moving \wedge \\ \qquad robotDestination(s) = mailbox(c) \\ \qquad \supset p = custPriority(c,s)] \wedge \\ \quad [\neg(robotState(s) = Moving \wedge \\ \quad\; \exists c\, robotDestination(s) = mailbox(c)) \supset p = -1] \end{array}$$

The $handleServiceFailure$ procedure goes as follows:

```
**proc** $handleServiceFailure(c)$
  $reduceCredit(c)$;
  **if** $credit(c) = 0$ **then**
    $notifyStopServing(p)$;
  **endIf**;
**endProc**
```

When the robot gets stuck on the way to customer c's mailbox, it first reduces c's credit and then checks whether it has reached zero and c has just become $Suspended$. If so, c is notified that he/she will no longer be served.

The $tryToWrapUp$ procedure is similar to $tryServeCustomer$:

```
**proc** $tryToWrapUp$
  $startGoTo(CentralOffice)$;
  $(robotState \neq Moving)?$;
  **if** $robotState = Reached$ **then**
    $freezeRobot$;
    $dropOffToCentralOffice$
    $resetRobot$
  **else if** $robotState = Stuck$ **then**
       $resetRobot$    % abandon attempt
  % else when service aborted nothing more to do
  **endIf**
**endProc**
```

It starts the robot on its way to the central office. The thread then waits until the robot halts. If the robot reaches the central office, then all undeliverable shipments on board are dropped off, and unless a new order comes in the program terminates. If the robot gets stuck, then the robot is reset and the procedure ends. If motion is aborted, the procedure ends immediately. In both cases control returns to the $mainControl$ procedure, which will serve a new order if

Figure 1: Our robot facing an obstacle.

one has come in or make a new attempt to return to the central office.

Procedure $dropOffShipmentsTo(c)$ delivers to customer c all shipments on board such that c is the shipment's recipient or c is the shipment's sender and the recipient has been suspended:

```
**proc** $dropOffShipmentsTo(c)$
  **while** $\exists n\, (orderState(n) = OnBoard \wedge$
      $(recipient(n) = c \vee$
       $sender(n) = c \wedge Suspended(recipient(n))))$ **do**
    $\pi\, n\, [(orderState(n) = OnBoard \wedge$
      $(recipient(n) = c \vee$
         $sender(n) = c \wedge Suspended(recipient(n))))?;$
      $dropOffShipment(n)]$
  **endWhile**
**endProc**
```

Procedure $pickUpShipmentsFrom(c)$ simply picks up all outgoing shipments from customer c's mailbox. Procedure $dropOffToCentralOffice$ drops off all shipments whose recipient and senders are both suspended to the central office. These are similar to $dropOffShipmentsTo(c)$ and we omit their code.

Note that by handling the cancellation of pick ups in a separate thread from that dealing with navigation and order serving, we allow the robot to be productive while it is in motion and waiting to reach its destination. This makes a better use of resources.

To run the system, we execute $mainControl$ after placing a few initial orders:

$$\begin{array}{l} orderShipment(Yves, Ken, 0) \parallel \\ orderShipment(Ken, Hector, 1) \gg \\ mainControl \end{array}$$

Experimentation

The high-level control module for the mail delivery application has been ported to an RWI B12 mobile robot and tested in experiments (see figure 1). The other software components for this were based on a system developed during an earlier project concerned with building an experimental vehicle to conduct survey/inspection tasks in an industrial environment (Jenkin *et al.* 1994). The system supports point to point navigation in a previously mapped environment and can use pre-positioned visual landmarks to correct odometry errors. It relies on sonar sensors to sense unmodeled obstacles.

The system's architecture conforms to the general scheme described earlier. It provides two levels of control. An onboard low-level controller (Robinson & Jenkin 1994) performs all time-critical tasks such as collision avoidance and straight line path execution. The low-level controller assumes that the robot is always in motion and communicates with an offboard global path planner and user interface module known as the *Navigator*. The Navigator takes as inputs a metric/topological map of the environment in which the robot is located and the coordinates (as defined in the map) of the two end points, i.e., the source and the destination of a path. By making use of some predefined path-finding algorithms such as breadth-first search or A^* the Navigator identifies a feasible path between the source and the destination. To follow the path, the Navigator decomposes it into segments (a segment is a straight line between two adjacent way-points) and then forwards the segments to the low-level controller for execution. The Navigator supervises the low-level controller and identifies failures in the low-level controller's ability to execute a path segment.

The ConGolog-based high-level control module interacts with the rest of the architecture by communicating with the Navigator through a socket interface. The high-level controller, Navigator, and low-level controller all run asynchronously. The primitive actions in the ConGolog interface model are implemented using operations provided by the Navigator (currently, the mail pickup and drop off actions are only simulated). For example, the ConGolog primitive action $startGoTo(p)$ is implemented as `[planPath(coordinatesOf(`p`));` `followPath]`, where `planPath` and `follow_path` are operations supplied by the Navigator.

Our experiments confirmed the system's ability to deal with navigation failures and to interrupt the current task when an urgent shipment order is made. For more details on the system, see (Tam 1998).

Discussion

We have described how ConGolog can be used to implement high-level robot controllers that can cope with dynamic and unpredictable environments — controllers that are reactive and support high-level plan reconsideration in response to exogenous events and exceptional conditions. Our work shows how a logic-based approach can be used to build effective systems while retaining features such as clear specifications, inferential power, and easy extensibility.

One limitation of the work accomplished so far is that the system developed is rather small. We need to experiment with more complex systems to see whether the general approach and the use of prioritized interrupts to provide reactivity scales up. As well, interrupts support the suspension of the current plan but not its termination. The addition of a conventional exception throwing and catching mechanism that terminates the current plan is being investigated.

Another limitation concerns the lack of search/planning in the current high-level program. However, the new model of De Giacomo and Levesque (De Giacomo & Levesque 1998) provides a clean specification of incremental high-level program execution in the presence of sensing. This will allow us to incorporate controlled search/planning in our programs while retaining a clean semantics. This might be most useful for dealing with unexpected plan failures. A Golog-based approach to execution monitoring and plan repair is developed in (De Giacomo, Reiter, & Soutchanski 1998). The use for Golog for planning is investigated in (Reiter 1998).

Another area under investigation is perceptual tasks. Such tasks often require sophisticated plan selection and involve information acquisition. We are currently working on an application where packages must be delivered to the recipient "in person" and where the robot must use sophisticated search strategies to locate the recipient, for example, asking whether a co-worker has seen the recipient (Tam 1998).

The high-level program execution model of robot/agent control that underlies our approach is related to work on resource-bounded deliberative architectures (Bratman, Israel, & Pollack 1988; Rao & Georgeff 1992) and agent programming languages (Shoham 1993). One difference is that in our approach, plan selection is coded in the program. This makes for a less declarative and in some cases more complex specification, but eliminates some overhead. On the other hand, the robot's world is modeled using a domain action theory and the world model is updated automatically using the successor state axioms; there is no need to perform asserts and retracts. Moreover, the evaluation of a test may involve arbitrary amounts of inference, although following logic programming philosophy, we take the programmer to be responsible for its efficiency/termination. Perhaps a more central difference is that our robots/agents can be understood as executing programs, albeit in a rather smart way; they have a simple operational semantics. Modeling the operation of an agent implemented using a resource-bounded deliberative architecture requires a much more complex account.

Acknowledgements

This research received financial support from the Information Technology Research Center (Ontario, Canada) and the Natural Science and Engineering Research Council (Canada). Hector Levesque came up with the idea of handling sensing through exogenous events. We thank him as well as Ray Reiter, Jeff Lloyd, Mikhail Soutchanski, Giuseppe De Giacomo, and Daniele Nardi for helpful discussions related to this work. Many of our papers are available at `http://www.cs.yorku.ca/∼lesperan/`.

References

Bratman, M.; Israel, D.; and Pollack, M. 1988. Plans and ressource-bounded practical reasoning. *Computational Intelligence* 4:349–355.

Brooks, R. 1986. A robust layered control system for a mobile robot. *IEEE Journal on Robotics and Automation* 2(1):14–23.

De Giacomo, G., and Levesque, H. J. 1998. An incremental interpreter for high-level programs with sensing. In *Working Notes of the 1998 AAAI Fall Symposium on Cognitive Robotics*.

De Giacomo, G.; Lespérance, Y.; and Levesque, H. J. 1997. Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in the situation calculus. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, 1221–1226.

De Giacomo, G.; Reiter, R.; and Soutchanski, M. E. 1998. Execution monitoring of high-level robot programs. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR'98)*, 453–464.

Jenkin, M.; Bains, N.; Bruce, J.; Campbell, T.; Down, B.; Jasiobedzki, P.; Jepson, A.; Majarais, B.; Milios, E.; Nickerson, B.; Service, J.; Terzopoulos, D.; Tsotsos, J.; and Wilkes, D. 1994. ARK: Autonomous mobile robot for an industrial environment. In *Proc. IEEE/RSJ IROS*.

Levesque, H. J.; Reiter, R.; Lespérance, Y.; Lin, F.; and Scherl, R. B. 1997. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming* 31(59–84).

McCarthy, J., and Hayes, P. 1979. Some philosophical problems from the standpoint of artificial intelligence. In Meltzer, B., and Michie, D., eds., *Machine Intelligence*, volume 4. Edinburgh, UK: Edinburgh University Press. 463–502.

Rao, A., and Georgeff, M. 1992. An abstract architecture for rational agents. In Nebel, B.; Rich, C.; and Swartout, W., eds., *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference*, 439–449. Cambridge, MA: Morgan Kaufmann Publishing.

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

Reiter, R. 1998. Knowledge in action: Logical foundations for describing and implementing dynamical systems. Draft Monograph, available at `http://www.cs.toronto.edu/~cogrobo`.

Robinson, M., and Jenkin, M. 1994. Reactive low level control of the ARK. In *Proceedings, Vision Interface '94*, 41–47.

Rosenschein, S. J., and Kaelbling, L. P. 1995. A situated view of representation and control. *Artificial Intelligence* 73:149–173.

Schoppers, M. J. 1987. Universal plans for reactive robots in unpredictable environments. In *Proceedings of the Tenth*

International Joint Conference on Artificial Intelligence, 1039–1046.

Shoham, Y. 1993. Agent-oriented programming. *Artificial Intelligence* 60(1):51–92.

Tam, K.; LLoyd, J.; Lespérance, Y.; Levesque, H.; Lin, F.; Marcu, D.; Reiter, R.; and Jenkin, M. 1997. Controlling autonomous robots with GOLOG. In *Proceedings of the Tenth Australian Joint Conference on Artificial Intelligence (AI-97)*, 1–12. Perth, Australia.

Tam, K. 1998. Experiments in high-level robot control using ConGolog — reactivity, failure handling, and knowledge-based search. Master's thesis, Dept. of Computer Science, York University.