RADIC - Towards a General Method for Integrating Reactive and Deliberative Layers

James Kramer and Matthias Scheutz
Artificial Intelligence & Robotics Lab
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556, USA
jkramer3,mscheutz@cse.nd.edu

***Abstract*— Hybrid architectures have been developed to preserve the responsiveness of reactive layers while also providing the benefits of higher-level deliberative capabilities. The challenge of hybrid architecture design is to integrate layers that operate on different time scales, use different notions of spatial relations, maintain varying degrees of state, etc. Techniques for integration, however, are typically very specific to the application or particular hybrid architecture.**

A general method is proposed for constructing hybrid architectures via the integration of pre-existing reactive and deliberate layers that requires minimal modifications. Its potential is demonstrated in navigation experiments on a robot, achieving performance comparable to a commonly used hybrid system with specially designed layers.

***Index Terms*— Hybrid architectures. Navigation. Obstacle avoidance. Path planning. Reactive deliberative integration.**

1. Introduction

Hybrid architectures for autonomous robots combine reactive and deliberative layers, capitalizing on the strengths of each while overcoming their individual shortcomings. Specifically, hybrid architectures attempt to preserve the responsiveness of reactive layers and the computational potential of representational deliberative planning and reasoning capacities. Yet, as pointed out in [19], hybrid architectures often suffer from various high-level issues: (1) "*ad hoc* or unprincipled designs" that require hand-crafting of control modules, (2) a tendency "to be very application-specific", and (3) unspecified supporting theory.

These issues can be overcome by employing a component that mediates between deliberative and reactive layers. The "Reactive And Deliberative Layer Integration Component" (RADIC) addresses all three concerns by providing a generalized "bridge" between layers. By maintaining the separation of layers while defining the mapping of data between them, RADIC: (1) allows principled design of layers in isolation, (2) provides a general method of connection that reduces (but does not eliminate) application-related specificity, and (3) obviates the need for a specific hybrid theory for engineering applications.

This paper revises and extends [22], organized as follows: Section 2 summarizes some important problems inherent in the integration of deliberative and reactive layers, with a review of some common hybrid architectures and which problems they do not resolve. Section 3 presents the RADIC specifications, its functionality and algorithms, and a range of potential applications for hybrid and solely reactive architectures. Section 4 describes a particular RADIC implementation that results from the combination of fully functional, pre-existing reactive and deliberative layers for a navigation task. Section 5 provides a performance evaluation of robotic experiments that demonstrate that the hybrid system with RADIC performs at least as well as a widely used, special-purpose hybrid architecture navigation system, while requiring fewer computational resources. Finally, a brief discussion and conclusion is given in Sections 6 and 7.

2. Hybrid Architectures

In the domain of autonomous robotic agents, where fast-working, highly responsive reactive layers are necessary (e.g., [11], [16]), the design of hybrid architectures has been a challenge, largely because the operational principles of the layers are so different. For example, reactive layers keep state to a minimum, if state is kept at all [10], process sensory information continuously, and produce actions immediately, often using continuous sensor-motor mappings. In contrast, higher-level architectural layers typically store information as discrete representations, using it to perform complex processing in an extended time-scale. The challenge of hybrid architectures is to integrate the layers' disparate aspects of operation, at least achieving the following functional mappings:

- **F1**: between "global" and robot-centric spatial relations
- **F2**: from discrete actions to continuous motion
- **F3**: between logical update time and real-time operation
- **F4**: from stateful to stateless operation

Over the last two decades, a variety of hybrid architectures have been proposed. The following gives an overview of a small selection of them to illustrate their diversity, describing how each satisfies items **F1-F4**.

SSS: [6] (short for "*S*ervo, *S*ubsumption, *S*ymbolic"), the symbolic layer uses a coarse-grained world model that configures the reactive layer via parameterization of selected behaviors in the subsumption layer. SSS explicitly addresses the functional mappings by discretizing space between the servo and subsumption layers (**F1, F2**) and time between the subsumption and symbolic layers (**F3, F4**).

AuRA: [2] uses a deliberative layer composed of a planner, spatial reasoner, and sequencer, while the reactive layer consists of libraries of behavior schemas, activated by the sequencer and adaptable via a homeostatic component. Once activated, the schema manager controls and monitors active behavioral process(es), whose output is combined using vector summation to produce actuation commands. Functional mappings **F1**, **F3** and **F4** are made through the interaction of the schema controller with the plan sequencer, spatial reasoner, and planner, while **F2** is accomplished by the selection of appropriate behavior schemas.

3T: [4] architecture extends work on both the RAPs [7] and ATLANTIS [9] hybrid architectures, and is composed of three "tiers": (1) a set of (re-programmable) reactive skills (RAPs) coordinated by a skill manager, (2) a sequencer (RAP interpreter) that activates/deactivates skills and event monitors, and (3) a planner that synthesizes goals into a partially-ordered task list (where each task may be composed of skills grouped into task networks). Mapping **F3** is satisfied by having tiers operate at different time scales, rather than "managerial responsibility" (i.e., level of knowledge content): the skill level tier operates on the order of milliseconds, the sequencer on tenths of a second, and the planner on the order of seconds to tens of seconds. Mappings **F1** and **F4** are performed using the combination of tiers: the planner operates at a high level of abstraction, the sequencer selects a set of skills from the RAP library, while skills operate in a context dependent manner. Mapping **F2** is made via the integration of skills and sequencer: RAPs are symbolic, discrete steps for accomplishing a task composed of skills that implement continuous motion.

Saphira: [12] incorporates a strong internal world model called the *local perceptual space* (LPS) that coordinates sensory data with internal representations. Planning is performed by the *PRS-Lite* component, (de)activating sets of behaviors whose output is fused using fuzzy logic. Functional mappings **F1** and **F3** are accomplished via the LPS and PRS-Lite, while **F2** is accounted for by the behavior selection and fuzzy logic fusion. Abstract representations are integrated in the LPS to fulfill mapping **F4**.

4-D/RCS: [1] features a hierarchy of six levels (actuator, servo, primitive, subsystem, vehicle, and section) that explicitly operate on different time scales, incrementally increasing from five milliseconds at the lowest to ten minutes at the highest. Each contains a supervisor to manage that level's activity and a set of subordinate agents that supervise units at the next lower level. Moreover, each node contains behavior generation, world modeling, sensory processing, value judgment processes and a knowledge database. Essentially, each level has the same self-contained structure, and thus mappings **F1-F4** are explicitly made between adjacent levels.

Although all of the above hybrid architectures successfully combine reactive and deliberative layers, each subscribes to a specific design philosophy or method, if not in integrating layers, then by specifying the design of the layers themselves. These integrations are specific to the employed reactive and deliberative layers (i.e., the layers need to be designed together as a cohesive whole), forming restrictions that not only limit potential reuses of the developed layers and software but, more importantly, the extent to which the designs themselves generalize.

It would be desirable to achieve a more general connection or integration method that could be employed for many different reactive and deliberative layers without the need for much adaptation of either layer. Ideally, a single, generalized, "mediating component" would be able to connect existing reactive and deliberative layers that successfully achieves the requisite functional mappings **F1-F4**. Several design desiderata suggest themselves for such a component; in particular, the method should:

- **I1**: require minimal changes to existing layers
- **I2**: provide true integration of layers via independent, autonomous, and parallel execution
- **I3**: use minimal processing time
- **I4**: add functionality that improves the performance of systems containing the layers in isolation

All the hybrid architectures above fail to satisfy at least one of **I1-I4**, as shown in Table 1 (in addition to the various criticisms included in [9], [4], [15]). Neither Saphira nor 4-D/RCS satisfies **I1**, albeit for different reasons. While the large amount of integration performed by the LPS in Saphira has the benefit of *coherency*, it also means that functional changes necessitate large modifications. On the other hand, the strict specification of interfaces between levels and substantial data flow within a single level in 4-D/RCS has the effect that slight modifications to functionality may entail large changes across components or levels.

None of SSS, AuRA, and 3T satisfy **I2** because all switch completely back and forth between reactive and deliberative layers; the deliberative layer produces

Table 1
HYBRID ARCHITECTURE SATISFACTION OF ITEMS **I1-I4**

	I1	I2	I3	I4
SSS	√	–	√	√
AuRA	√	–	√	–
3T	√	–	√	–
Saphira	–	√	–	√
4-D/RCS	–	√	–	–

a fully formulated plan that is passed to the reactive layer, which then remains in control until a step either completes or fails.

Both Saphira and 4-D/RCS suffer from difficulties with **I3**; the use of the LPS in Saphira requires a substantial amount of processing, while the full complement of components in 4-D/RCS introduces possibly redundant or unnecessary processing.

Finally, **I4** presents difficulties for 3T, AuRA, and 4-D/RCS. In both 3T and AuRA, operation of the reactive layer is tightly knit and depends completely on the sequencer, making it difficult to add external functionality. A similar situation occurs with 4-D/RCS, where the functionality of a given level is specified completely by its internal operation.

The proposed RADIC component, described next, has been designed to satisfy all the functional mappings **F1-F4** and design desiderata **I1-I4**.

3. The RADIC Approach

The approach taken with RADIC can be described as a *general method* for integrating (1) a *sequence of goals* originating from a deliberative layer, (2) a stream of *sensor data*, and (3) a *sequence of actions* (e.g., motor commands) from the reactive layer (see Figure 1), as performed by the `updateRADIC` algorithm shown in Figure 2. The universality of a particular RADIC component is dependent on the representations used in mapping data between layers; at some application-dependent level, representations are either too abstract to be useful or too narrow to be applicable. For instance, a deliberative layer that produces a goal list composed of an ordered sequence of spatial locations to "visit" (such as in navigation or arm movement) requires translation of world-coordinates into egocentric positions. While a RADIC component integrating such layers may be applied to other tasks requiring spatial reasoning, it may be inadequate for, say, a goal list including what to do at each location. RADIC is a general technique for integrating layers, but its specificity is dependent on implementation.

Functional mapping **F1** is satisfied by the `convertGoal` function, detailed below. The `updateRADIC` function is called at least as frequently as the reactive layer produces outputs (e.g., sends

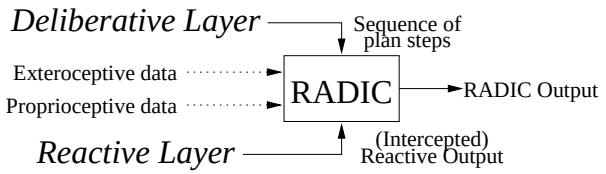

Fig. 1. Functional diagram of RADIC showing (possible) inputs and output.

FUNCTION updateRADIC(R-out,S-$change$,D-$goals$,G-$strat$)
static $M \leftarrow \emptyset$
static $currentState \leftarrow \emptyset$
static $oldState \leftarrow \emptyset$
$M \leftarrow M$ + **convertGoal**(D-$goals$, G-$strat$)
$oldState \leftarrow currentState$
$currentState \leftarrow$ **updateState**(S-$change$)
$action \leftarrow$ **defaultAction**(R-out)
for all $G \in M$ **do**
  for all $C \in C_G$ **do**
    if C_G **then**
      $M \leftarrow M - \{G\}$
    end if
  end for
  $G \leftarrow$ **updateGoal**(R-out, S-$change$)
  if $B_G = \emptyset$ **then**
    $action \leftarrow$ **chooseAction**(G, $action$)
  end if
end for
return $action$

Fig. 2. The generic update algorithm for RADIC.

motor commands), and thus satisfies item **F3**. The functional mappings **F2** and **F4** are an inherent part of the RADIC component's operation: a goal is a discrete representation, while RADIC's output effects continuous motion (**F2**, via either discrete actions or motor commands), and the state of the deliberative layer is maintained in internal memory, while not requiring any state from the reactive layer other than its outputs (**F4**).

In more detail, the `updateRADIC` function takes as arguments the outputs from the reactive layer R-out (typically, intercepted motor commands), the changes as determined by sensors S-$change$ (proprioceptive, exteroceptive, or missing), a sequence of goals from the deliberative layer D-$goals$ (which can be empty), and a strategy for goal attainment G-$strat$. The sequence of goals $D\text{-}goals = \langle G_1, G_2, ..., G_n \rangle$ is transformed, in accordance with item **F1**, from a "global" representation into an ego-centric representation (e.g., for navigation) $\langle \overline{G}_1, \overline{G}_2, ..., \overline{G}_n \rangle$ by the `convertGoal` function and stored in RADIC's internal memory M. `convertGoal` also associates a set of "blocking goals" B_G and a set of goal conditions C_G with each goal G that will depend on G-$strat$ (in general, G-$strat$ is a function that defines or influences the relation among G, B_G, and C_G–we will describe a particular strategy in the next section). Blocking goals are used to impose a priority ordering on goals (goals that are blocked are not considered for action selection). Similarly, goal conditions are used to determine when goals have been achieved. A goal is

either *active* or *inactive*, depending on whether or not its set of "blocking goals" is empty; a goal G is removed from M when at least one of its goal conditions $C \in C_G$ is met. The state of the robot is computed based on the old state using `updateState`, while the default action (i.e., output) is chosen in `defaultAction` based on $R\text{-}out$. All goals are then updated in `updateGoal` based on $R\text{-}out$ and any changes detected via sensory inputs $S\text{-}change$; the active goals determine the action for the robot via the `chooseAction` function, based on the goal and the actions chosen so far.

Unlike other hybrids, RADIC satisfies all of items **I1-4**. Minimal changes to each layer are required for integration (**I1**), as layer output is simply redirected as RADIC input; RADIC relies on internal data structures and functions for its operation. The RADIC component operates independently of either layer, performing its tasks in parallel with the layers' operation (**I2**). Computational cost is small (**I3**), as only state updates, goal list updates, and action choice are required. Updates of the goal list require both a measure of state change (via `updateState`) and a subsequent application of the calculated change for each point (via `updateGoal`). Only "active" goals are considered in modifications of the reactive layer's output at any given time. Finally, combining the layers improves overall performance (**I4**) by augmenting a reactive layer with planning capacities and providing a deliberative layer with the ability to act. Furthermore, custom improvements can easily be incorporated into the `updateState` (such as additional processing of $S\text{-}change$) and `updateGoal` (such as goal reordering or progress monitoring) functions.

Note that the RADIC algorithm is kept as general as possible so that it can be tailored to the broadest range of reactive and deliberative layers. To apply it to a particular robotic architecture (e.g., to integrate two pre-existing layers), each of the five functions `convertGoal`, `updateState`, `defaultAction`, `updateGoal`, and `chooseAction` must be implemented. Each of these may incorporate additional functionality, further promoting good design practices and potentially improving performance. For instance, `chooseAction` may be modified such that behavior selection can be dynamically altered according to the current state. In effect, this allows inclusion of mechanisms that avoid engaging the full deliberative layer, thereby preserving reactivity while increasing deliberative functionality.

4. A RADIC Navigation Component

This section shows how RADIC can be easily adapted to the typical mappings found in other hybrid architectures; since hybrids were historically developed to address robot navigation issues, RADIC is described in that context.

4.1. General Description

For navigation tasks, a *goal* is a location in the environment, a *plan* is an ordered sequence of points that form a *trajectory*, *reactive output* consists of motor commands, and *deliberative output* consists of trajectories that the robot has to follow. As planners typically use "global" coordinates in plan formulation, `convertGoal` must transform them to robot-centric points. To ensure that plan points will be visited in order, the strategy of automatically associating the following two sets $C_{G_i} := \{within(G_i, \epsilon)\}$ and $B_{G_i} := \{G_j | j < i\}$ with each plan point in G is used, where "within" is true if a plan point G_i is closer than ϵ distance to the current position of the robot. The first set guarantees that a plan point can only become active if all previous plan points have been visited (and subsequently removed), and the second ensures that the robot has to come close enough to the plan point to be able to count it as "visited".

Both *currentState* and *oldState* are poses; `updateState` calculates state change using sensory feedback, either exteroceptive (e.g., localization) or proprioceptive (e.g., a velocity-based estimation as presented in [13]). Similarly, `updateGoal` updates the relative positions of all plan points stored in M according to the state change. The technique used for navigation action selection in this particular RADIC component is *schema theory* (as in [2]), in which `defaultAction` converts reactive motor commands to a robot-centric motion vector, to which `chooseAction` adds a vector for a plan point, if one is active. The resultant action is a directional vector combining the direction of the intended reactive movement with that of the next plan point.

The exact combination of vectors could be application-dependent, but will, in general, be a linear sum of the individual vectors, where each vector is scaled by the inverse of the square of its distance from the robot's location, for example, thus contributing less to the overall direction the farther it is away [2]. This effectively amounts to creating an "artificial potential" based on directional vectors derived from the intercepted motor commands and the RADIC component's own directional output. The gradient of the resultant vector is then translated back into motor commands.

4.2. Navigation Applications of RADIC

Several applications are suggested by the functionalities of the navigation RADIC described above (shown in Figure 3). Probably the most direct is that of a translator between existing planner and reactive layers, where the planner is intended to improve the reactive performance (as shown in Figure 3-A). In this case, the planner creates an abstract trajectory path and passes it to the

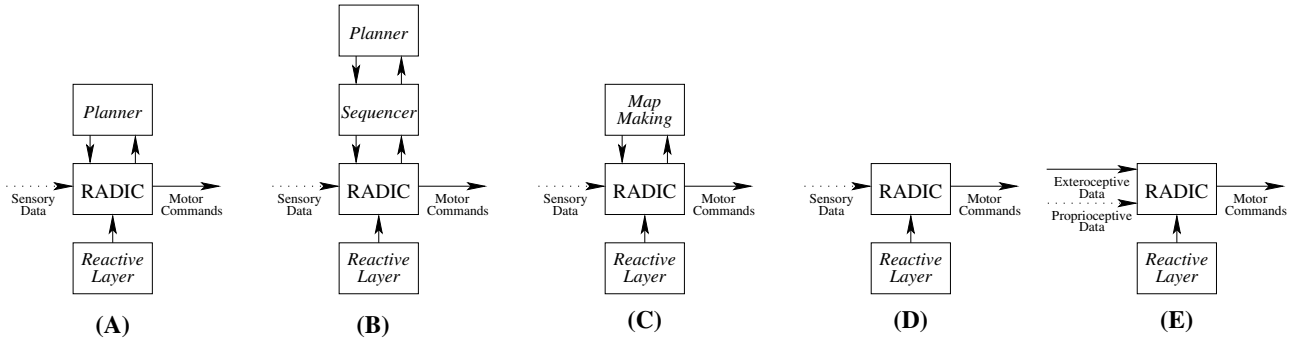

Fig. 3. RADIC Navigation Applications (as sequencer and plan execution mechanism for): **(A)** high-level planners, **(B)** as a discrete-continuous mapper between sequencer with discrete action representations and a reactive layer, **(C)** as part of a high-level map-making configuration, **(D)** for low-level trajectory improvement and tracking of past locations, and **(E)** low-level landmark detection and tracking.

RADIC component, which serves as a plan execution component (demonstrated in Section 5.2).

Another application of the RADIC component (Figure 3-B) is to serve as a translator between an action sequencer in a deliberative layer and a low-level reactive layer, where the action sequencer uses discrete representations of actions (e.g., as in [4]), while the reactive layer uses continuous representations (e.g., potential fields).

The RADIC component can also be used for multi-level map making, where RADIC records local properties (such as obstacles or other points of interest) under the guidance of the higher levels, returning the map to a higher level component, which can store it for subsequent use (Figure 3-C).

RADIC components can also be added to existing low-level reactive layers (Figure 3-D, demonstrated in Section 5.1) without a deliberative layer to improve the performance of the reactive system in various ways. A straightforward use is to provide simulated compass sensors. They can also be used to improve goal directed behavior (by virtue of an attractive or repulsive point for the goal location put in the list of points RADIC retains). Finally, and perhaps most importantly, RADIC components can track past locations, which allows for the implementation of mechanisms intended to get potential field-based systems out of local minima (such as the "avoid past" scheme [3]).

Finally, RADIC components with additional exteroceptive sensory input (Figure 3-E) can be used to detect and track landmarks or other items of interest in a robot-centric coordinate system.

4.3. RADIC Navigation Enhancements

Several enhancements can easily be incorporated into the navigation RADIC: (1) the means to improve the internal location memory (or "map") M accuracy by virtue of integrating proprioceptive adjustments, (2) failure detection as it relates to reaching plan points, and (3) improving the actual trajectory of the robot in relation to the navigation segments of the plan. Each of these improvements satisfies the design desiderata identified by items **I1-I4**.

1) Improving Location Accuracy: In navigation tasks, it is imperative that a robot's perceived location matches its actual location. To provide relatively accurate motion tracking, RADIC uses a separate dead-reckoning module in `updateState` that receives proprioceptive information from a separate dead-reckoning module that uses the following algorithm, whose development is based on [5]. Further improvement to the accuracy of RADIC's internal "map" might include a localization algorithm (e.g., [25]).

At a given update rate R, the left and right wheel shaft encoders are sampled with a count N_L and N_R, respectively. By relating the number of encoder counts of the left and right wheels $N_{L/R,i}$ at each instance i, the incremental travel distance of the left $\Delta U_{L,i}$ and right wheel $\Delta U_{R,i}$ can be computed by $\Delta U_{L/R,i} = c_{mL/mR} N_{L/R,i}$, where $c_{mL/mR}$ are the conversion factors for translating encoder counts into the respective left and right linear wheel displacement. These conversion factors are dependent on three physical properties of the wheels, in which the encoders are mounted: the nominal left and right wheel diameter $D_{L/R}$, the gear ratio between the encoder and drive wheel $n_{L/R}$, and the encoder counts per revolution C_e on the motor shaft, which are related by $c_{mL/mR} = \pi D_{L/R}/n_{L/R}C_e$.

The incremental center point displacement of the robot's center point C, denoted ΔU_i, is given by $\Delta U_i = (\Delta U_R + \Delta U_L)/2$, and the incremental change in orientation, denoted $\Delta\theta_i$, being inversely proportional to the wheel base width d, is given by $\Delta\theta_i = (\Delta U_R - \Delta U_L)/d$.

With these equations it can be shown that the global positions X, Y as well as the global angular position θ can be computed by: $\theta_N = \sum_{i=0}^{i=N} \Delta\theta_i$, $X_N = \sum_{i=0}^{i=N} \Delta U_i \cos\theta_i$, and $Y_N = \sum_{i=0}^{i=N} \Delta U_i \sin\theta_i$.

A similar argument works for relating the change in x, y and angular velocities $(\dot{x}), (\dot{y}), (\dot{\theta})$: $(\dot{x}) = \cos(\theta)\omega$, $(\dot{y}) = \sin(\theta)\omega$, and $(\dot{\theta}) = v$, where $\omega = (V_L + V_R)/2$ is the average of the left and right wheel velocities $v =$

$(V_R+V_L)/d$ is the angular velocity of the rotating robot.

2) Detecting Failure: Another navigation issue that must be dealt with failure detection (or lack of progress in reaching a goal). Two relatively simple methods have been implemented as part of `updateGoal`: (1) a timeout mechanism, in which a time condition is associated with a goal point P such that when P is active for longer than ϵ amount of time, the goal is no longer pursued and the next point is taken as the new goal (using the *within* predicate, as described above).

The second method of progress measurement uses an "avoid-past" behavior, similar to that described in [3]. RADIC's internal "map" is used to provide the reactive layer with memory capacities, allowing the component to "remember" where the robot has been. Since the RADIC navigator component uses a potential fields method to determine modified motor commands, the avoid past behavior takes the form of a set of repulsive locations that the robot has previously visited, each of which has a decay rate associated with it so that avoidance is temporary.

3) Improving the Trajectory: Finally, the RADIC navigational component can be used to improve the actual trajectory traveled by the robot. Gradient-based control in practice often leads to situations where the robot overshoots a target location (especially at high speeds) and subsequently deviates from the projected trajectory (especially when sharp turns are required). To reduce the deviation from a given plan trajectory, a special class of conditional points called "floating points" can be created for a sequence of plan points. The purpose of these points is to allow (1) the robot to stay on its planned trajectory between plan points, ensuring that it will come closer to them faster than otherwise, and (2) the robot to turn smoothly at sharp angles, where it otherwise would be locked in a sequence of back-and-forth moves.

Specifically, for each plan point $P_i \in (P_2, ..., P_{n-1})$, at least one "incoming" floating point I_i and one "outgoing" floating point O_i are inserted before and after P_i into the plan sequence. For P_1 and P_n only outgoing and incoming points are created, such that the resultant plan sequence is given by $(P_1, O_1, I_2, P_2, O_2, ..., I_n, P_n)$. Incoming floating points are defined by $I_{P_i} = P_i - f(s)\frac{P_i - P_{i-1}}{||P_i - P_{i-1}||}$, where $f(x) = c + x$, s is the maximum speed used by the robot on the trajectory segment (P_{i-1}, P_i), and $||..||$ is the Euclidean norm. Their removal condition is given by $C_{P_i} = \{within(P_i, \epsilon')\}$, where $\epsilon' > \epsilon$, thus making it less strict than for plan points. Outgoing floating points can be of two types: type 1 is defined analogous to incoming field points (with the same removal conditions): $O_{P_i} = P_{i+1} + f(s)\frac{P_{i+1} - P_i}{||P_{i+1} - P_i||}$. The second type is created dynamically whenever the angle α between two consecutive plan segments (P_{i-1}, P_i) and (P_i, P_{i+1}) is $90 < \alpha < 270$ (in degrees). The point is then placed at $P_i + (sin(\theta)d, P_i + sin(\theta))$, where $d = f(s)\frac{P_{i+1} - P_i}{||P_{i+1} - P_i||}$ and θ is the robot's current heading as recorded by RADIC.

4.4. An Implementation of RADIC Navigator

For experimental purposes, a hybrid architecture that integrates two pre-existing and fully functional, yet very different, reactive and deliberative systems has been implemented. The reactive layer of the test system is a schema-based system, similar to that proposed by Arkin [2], developed in Java prior to and separate from the RADIC implementation. *Perceptual* schemas are used to perform sensory processing, the results of which are passed to *motor* schemas. The motor schemas produce vectors that represent motor actions, summed to produce a single vector that is transformed into motor commands. Two behaviors are available to the system: *obstacle avoidance* and *wandering* [21].

The deliberative layer is a modified version of ThoughtTreasure [18], a system developed for natural language processing and designed to operate purely as stand-alone software. It has spatial and temporal representations, scripting capacities, and a high level planner. Of particular interest for integration via the RADIC component is the representation of objects and places in a discrete grid map. The planner is able to produce a sequence of grid spaces as an approximate trajectory from one object or place to another. Modification of the system, as concerns RADIC, consists of formatting the sequence of grid spaces as output.

These disparate systems are successfully integrated using RADIC and demonstrate the implementation items **I1**-**I4** presented in Section 2. Minimal changes to each system were required for integration (item **I1**); modification of ThoughtTreasure consisted of establishing a socket connection and reformatting the plan output, while modification to the reactive layer consisted of interception and redirection of motor commands. Combining the layers improves overall performance (item **I2**) by augmenting the reactive layer with planning capacities, while providing ThoughtTreasure the ability to physically act on otherwise disembodied commands. The RADIC component operates independently of either layer, performing its tasks in parallel (item **I3**). Finally, computational cost is small (item **I4**), as only point list updates and the calculation of new motor commands are required. Updates of the point list require both a measure of robot motion and a subsequent application of the calculated position change for each point. Only "active" points of the point list are considered in modification of the reactive layer's output at any given time.

The details of the system's operation are: a sequence of grid cells, produced by ThoughtTreasure at an arbitrary time, forms a navigation plan and is transmitted over a socket connection to RADIC. Upon reception, the points are converted by the `convertGoal` function

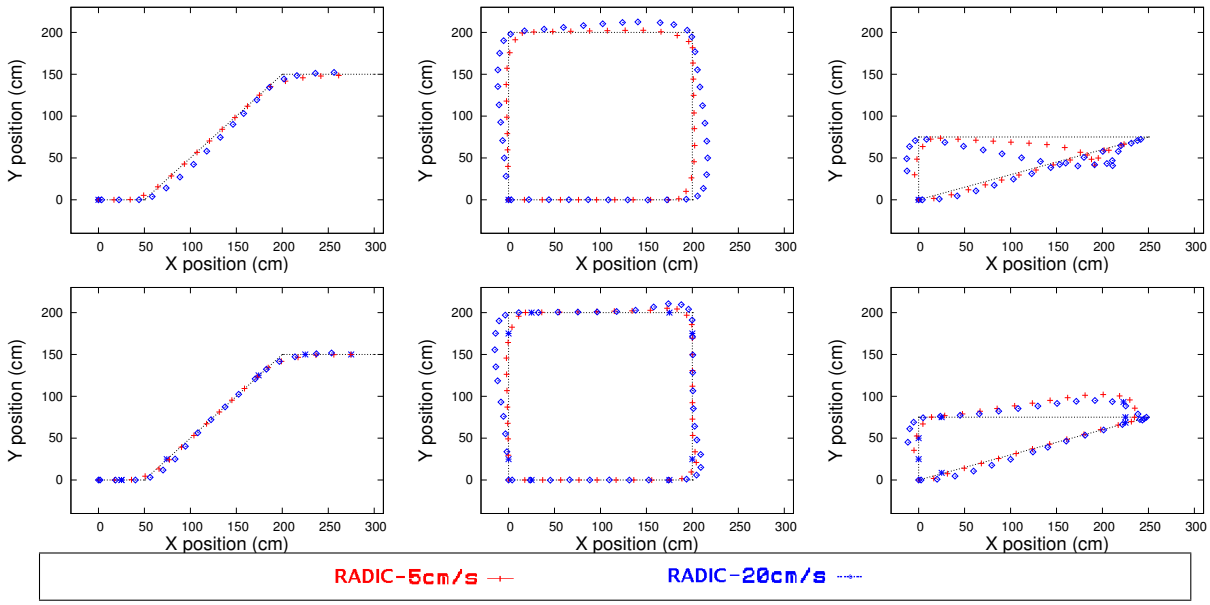

Fig. 4. **Top row**: RADIC validation of S-shaped, square, and triangular trajectories without "floating points". **Bottom row**: RADIC validation of S-shaped, square, and triangular trajectories with "floating points".

into a robo-centric point list with corresponding blocking points and conditions. Incoming sensor information (sonar and/or laser data) is used in `updateState` and `updateGoal` to calculate both the robot's and the relative goal point's location. The reactive layer's motor commands are intercepted and transformed into a vector by `defaultAction`, used by `chooseAction` in combination with the active point vectors to generate a resultant vector that is transformed into new motor commands, informing ThoughtTreasure of the updated robot position on request.

5. Experimental Validation and Evaluation

Validation and performance evaluation experiments were performed using the implementation of the RADIC navigation component described above. Navigation was *necessarily* chosen as the test task to permit comparisons with another hybrid architecture. In particular, CARMEN [17], whose authors say that it is "organized as an approximate three-tier architecture" (see the *3T* item in Section 2), was selected for trajectory-following comparisons over other navigation systems because of its free availability (thus avoiding any potential problems with replicating the algorithm) and accurate navigation (thus providing a worthy comparison). All experiments were conducted on an ActivMedia P2DXE robot with an on-board 850 MHz Pentium III PC104 board running Linux kernel 2.6.1, using the Java-based ADE architecture development environment [20], [14]. Throughout each experimental run the robot used only the onboard PC and thus operated completely autonomously.

5.1. Validation Experiments

A configuration like Figure 3-D that relies only on the reactive layer implementation described in Section 4.4 validated RADIC's operation. RADIC served as the deliberative layer, initialized with the way-points necessary for successive legs of the trajectory and operating as described in Section 4. Two classes of experimental validation were performed, the first using only the dead-reckoning enhancement and the second using both dead-reckoning and "floating points". Plans consisted of location sequences that form geometric trajectories; use of dead-reckoning was necessary for base-line validation as the accumulated error became too large, too quickly, to consider navigation successful.

Table 2
RESULTS OF RADIC GEOMETRIC FIGURE TRAJECTORIES

	Type	Speed (cm/s)	Dist. (m)	Time (s)
No Floating Points	S-shape	5	3.35	36
		20	3.37	9
	Square	5	7.45	82
		20	7.83	20
	Tri-angle	5	5.34	60
		20	5.71	17
With Floating Points	S-shape	5	3.35	36
		20	3.34	9
	Square	5	7.57	85
		20	7.86	21
	Tri-angle	5	5.52	63
		20	5.73	16

Three trajectory types are shown in Figure 4, starting at location (0,0): *S-shaped* via (50,0) and (200,150) to (300, 150); *squared* via (150,0), (150,150), and (0,150) back to (0,0); and *triangular* via (300,75) and (0,75)

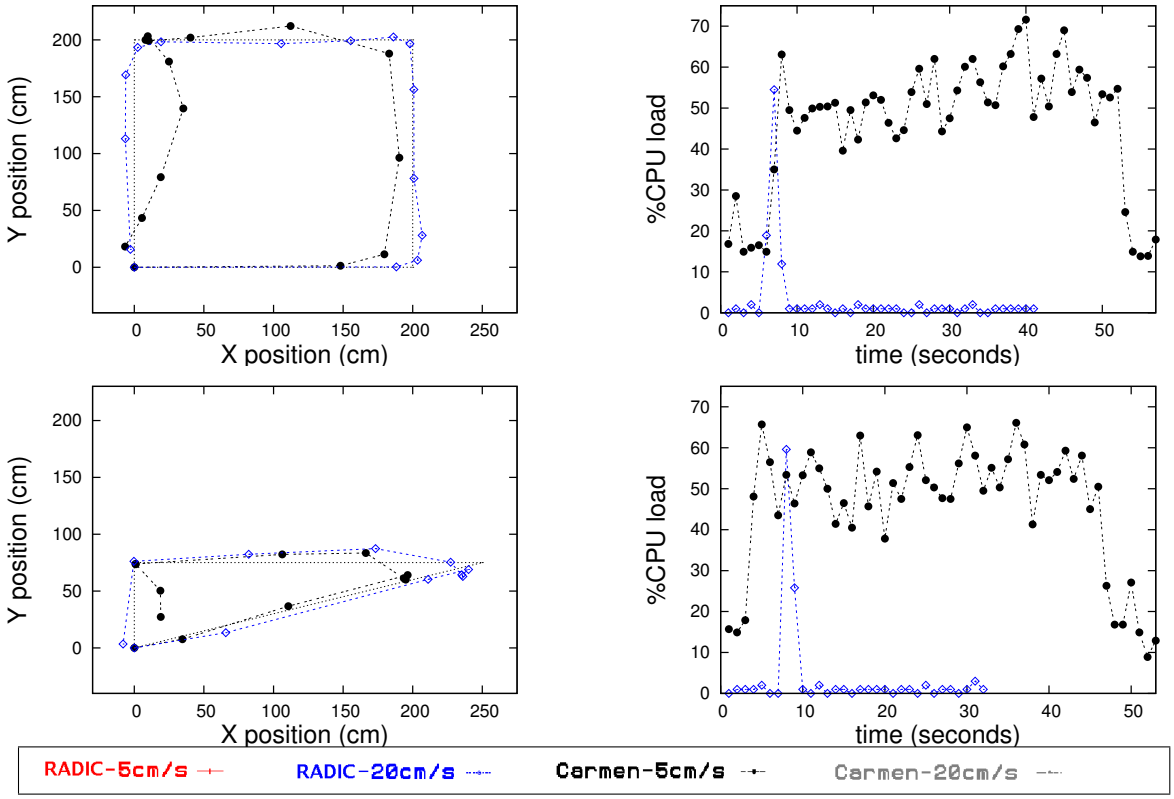

Fig. 5. Evaluation experiments comparing RADIC and CARMEN for square and triangular trajectories in relation to trajectories (left column) and %CPU load (right column).

back to (0,0) (all units are in cm). Shown for each trajectory type are the paths traced using the RADIC navigator, with and without "floating points" (top and bottom rows, respectively) at two different velocities (5 cm/s and 20 cm/s).

Speed, distance, and time measurements of the trajectories, both with and without floating points, are shown in Table 2. In all conditions, the robot is able to reach all plan points in order. The main difference in the trajectories lies in the use of "floating points", which reduce the deviation from the ideal trajectory (indicated in the figures by the solid lines). This is especially clear in the case of sharp turns (as in the case of the triangles), where the robot has to go back-and-forth several times without floating points. Given the narrow valley of attraction imposed by the potential field-based control, the placement of near-by floating points can smoothen the trajectory (effectively, widening the basin of attraction). Both total distance and elapsed time can be expected to increase when using floating points, due to the tighter trajectory fit. However, the time of the 20 cm/s triangle using floating points decreases; this can be attributed to the fact that by improving the trajectory tightness, the robot does not perform any forward and back movements.

5.2. Performance Comparison on Geometric Trajectories

To evaluate the performance of RADIC's navigation function, a configuration like that shown in Figure 3-A is used, where the RADIC component acts as the interface between the reactive and deliberative layers (as described in Section 4.4). Comparisons are made to CARMEN [17], an example of a 3T hybrid architecture (see Section 2). Experiments compare the performance of *squared* and *triangular* geometric trajectories where the robot's top speed was limited to 20 cm/s, capturing the %CPU load while the robot's path was physically recorded on the ground with an ink trail. After successful trajectory completion, the path was measured by hand. Results are shown in Figure 5)

Table 3
COMPARISON OF TRAJECTORY NAVIGATION (20 CM/S)

Type	Hybrid	Dist. (m)	Time (s)	CPU (%)
Square	RADIC	7.74	29	2.9
	CARMEN	7.56	48	46.3
Triangle	RADIC	5.81	25	3.5
	CARMEN	5.11	45	45.9

Navigation using RADIC provides a trajectory comparable to that of CARMEN. Total distance travelled, time

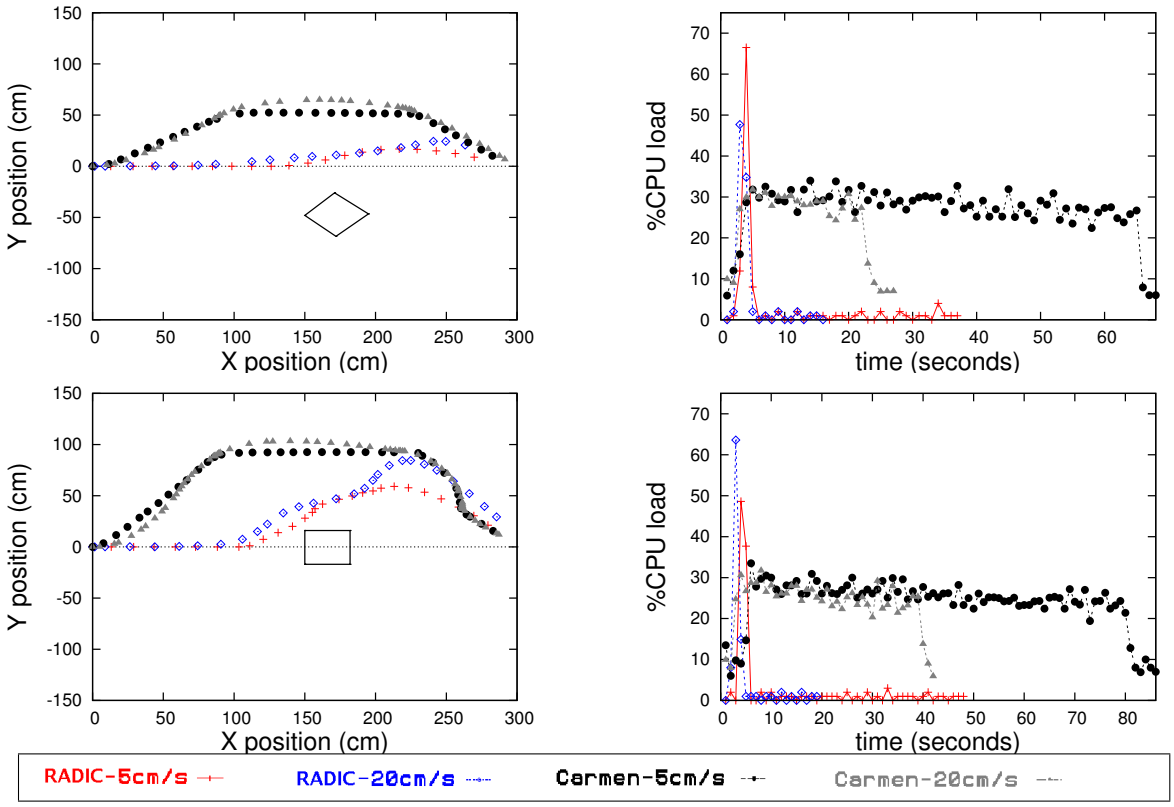

Fig. 6. **Left column**: Evaluation experiments comparing RADIC and CARMEN trajectories for unknown obstacles. Obstacles that are tangential to the ideal trajectory are shown in the top figure, while the bottom figure illustrates obstacles that completely obstruct the goal location. **Right column**: Comparison of RADIC and CARMEN %CPU load for the trajectories in the left column.

to completion, and average %CPU load[1] are shown in Table 3. In each case, the total distance traveled using RADIC navigation is slightly longer, due to the fact that task completion ends closer to the final goal point. At the same time, the total time taken to complete the total trajectories are just below half of that taken by CARMEN, while the average CPU load never rises above 3.5% for RADIC, compared to 46.3% for CARMEN. The reduced time is due to the operation of the reactive layer, where the robot moves at close to its maximum speed whenever it is in motion, whereas CARMEN often pauses during movement. Reduced CPU load is due to RADIC's linear update time, relative to the expense of CARMEN's navigator and localizer, which continually reformulate a plan.

Because the layers in a 3T architecture are tightly integrated, both the navigator and localizer are *necessary* for trajectory completion and the computational load cannot be substantially reduced. RADIC's mid-level, potential-field based "guidance" of the reactive layer allows the deliberative layer to produce the plan only once, which is then dynamically adjusted in a computationally inexpensive way during plan execution.

[1]The "spike" seen at the start of an RADIC run is due to the initialization of the Java virtual machine.

5.3. Performance Comparison with Unknown Obstacles

Comparing the effect of unknown obstacles on performance also uses a configuration like Figure 3-A. A single, straight-line trajectory of 3m is considered, with an obstacle placed either tangential to or directly obstructing the path (as shown in Figure 6). In both cases, the nearest obstacle edge is placed 1.5m in front of the robot's initial position, one centered on the path, the other shifted 26 cm to the side. Each show two trajectories, with maximum speeds of 5 and 20 cm/s.

Table 4
COMPARISON OF OBSTRUCTED-PATH NAVIGATION

Type	Hybrid	Dist. (m)	Time (s)	%CPU
Tangent (5cm/s)	RADIC	2.82	34	3.1
	CARMEN	3.34	51	26.7
Tangent (20cm/s)	RADIC	2.77	13	5.8
	CARMEN	3.42	30	23.4
Obstruct (5cm/s)	RADIC	3.09	44	2.6
	CARMEN	4.31	81	23.8
Obstruct (20cm/s)	RADIC	3.41	15	5.2
	CARMEN	4.47	49	23.7

Results shown in Table 4 represent the trajectory of the robot executing autonomously to reach the goal. In all cases, navigation using the RADIC component trav-

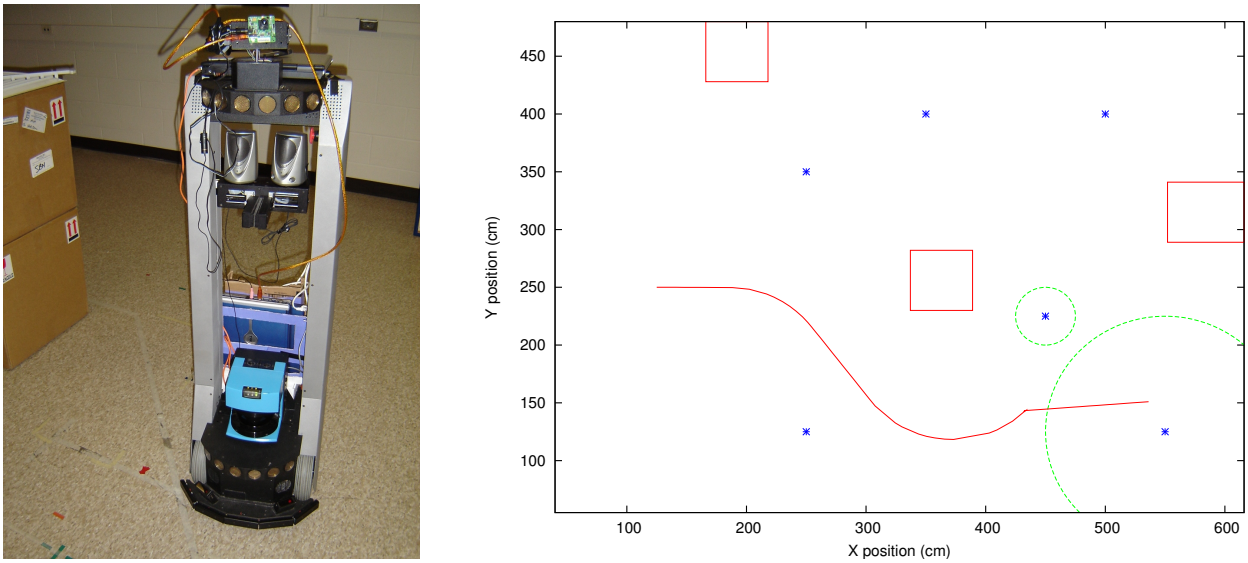

Fig. 7. **Left:** The robot used in the HRI experiment described in Section 5.4. **Right:** A typical trajectory of the robot from an experimental run (circles indicate transmission regions with sufficient transmission strength, stars indicate local interference minima, and boxes indicate "rocks", i.e., obstacles).

elled a shorter distance in one-half to two-thirds of the time and used a fraction of the CPU load compared to CARMEN. Similar to the results from Section 5.2, performance improvements can be attributed to RADIC's mid-level decoupling of layers that yields navigation ability without requiring replanning or consulting the deliberative layer.

Results for dynamic (i.e., moving) obstacles exhibit similar performance in both systems. For RADIC, if an obstacle is placed such that the removal condition cannot be met (e.g., because the obstacle is *on* the point), the robot will continue attempts to reach the location. Dynamic obstacles do not pose a problem, as they will eventually move and allow the robot passage. Static obstacles, on the other hand, cause the robot to get "stuck"; the implicit assumption is that the planner does not pick unreachable points, consistent with the design specification (as is the case in CARMEN). It is, however, possible to attach time-outs to plan points, such that a location is skipped after a given time, allowing the robot to proceed.

5.4. Other Applications of RADIC

A final validation of RADIC's flexibility and utility in robotic applications is provided by its use in research regarding an *affective architecture* (see [24] for a detailed overview). In addition, a variant of the RADIC component is used as a *virtual sensor* in a joint human-robot task [23], similar to those laid out in [8], which is described next.

The scenario takes place on a remote planet, simulated in a room of approximately 5m x 6m (see Figure 7), where the human is directing the robot using natural language. The task is to find an appropriate location from which to transmit data to an orbiting space craft; the RADIC variant is used as a "field detector" that detects the "transmission strength" at the robot's current location. Rather than being a sequence of goal locations, points in the internal map are assigned numerical values that represent local "strength" minima of transmission interference, which increases proportionally with the distance from the point's location.

As the robot moves across the "terrain", the transmission strength at its current location is determined according to its distance from the nearest "field strength point". In the figure, these points are depicted as "stars" (unknown to and directly undetectable by the operator), sufficient transmission strengths are depicted by a circle centered around a "field strength point", squares represent actual obstacles in the experiment configuration, and the curved line is the trajectory the robot traces under direction from the human.

6. Related Work

To provide a "bridge" between reactive and deliberative layers, any hybrid architecture must satisfy the functional mappings **F1-4** specified in Section 2. However, a *generalized* component that integrates reactive and deliberative layers–most aptly demonstrated when integrating *pre-existing* layers–should also satisfy the design desiderata specified by items **I1-4** in that section.

The *SSS* [6] architecture does not meet item **I2** (that is, independent, autonomous, and parallel execution of all layers) due to its total transfer of control between reactive and architectural layers. A RADIC component

accepts input asynchronously from either layer, allowing all three free to continue independent, parallel execution.

For the same reason, neither *AuRA* [2] nor *3T* [4] satisfy item **I2**. Furthermore, item **I4** presents a difficulty, as the symbolic plans produced by their deliberative layers are passed to sequencers for execution, which then select the appropriate controller (a behavior schema in the case of AuRA and a RAP for 3T) from a library. The intimate relation between layers makes it difficult to separate and use them in isolation, while the total dependence of the reactive layers' operation on the mid-level sequencer makes it difficult to add functionality not already contained therein. RADIC, on the other hand, transforms and replaces the other layers' output; additional functionality can be added, either as part of the transformation or as internal processing.

Saphira [12] fulfills neither items **I1** nor **I3** due to the large amount of information integration performed by the LPS. Changes to overall functionality require extensive modification (**I1**) to preserve coherency, while substantial computational resources are necessary (**I3**). In its capacity to operate as a separate component, RADIC requires little in the way of computation and no modifications of the layers themselves.

Finally, operation of the *4-D/RCS* architecture relies on a consistent structure of all layers, strictly regimented by timescale. Item **I1** is not satisfied because functional changes that cross timescale boundaries would require corresponding (likely internal) modification of the affected layers. The need to supply each layer with a full complement of internal components likely violates **I3**, leading to redundant processing. Similar to AuRA and 3T, the tight relationship between layers makes adding functionality **I4** problematic. By preserving the decoupled operation of layers, RADIC avoids each of these issues.

7. Conclusion

The validation and evaluation experiments show that RADIC is capable of (1) integrating existing reactive and deliberative layers–even those developed without consideration of robotic platforms–to form a hybrid architecture and (2) improving the performance of the combined system by improving trajectories and compensating for missing obstacle information at the deliberative layer (i.e., the hybrid architecture is "more than the sum of its parts"). Furthermore, trajectory following performance is comparable to that of CARMEN–a system *specifically designed with integrated planning and plan execution components*–while requiring far less CPU resources. The successful use of RADIC in the DIARC architecture and its variants in HRI experiments indicate its utility and flexibility.

Beyond the demonstrated performance improvements (and unlike other hybrid architectures), RADIC has the benefit of satisfying design desiderata **I1-I4**: it requires minimal modifications to pre-existing reactive and deliberative layers, executes independently, autonomously, and in parallel with the layers, uses minimal computational resources, and improves the functionality and performance of systems containing individual layers. While the reactive layer used in the experimental evaluation is schema-based, RADIC can be attached to *any reactive layer*, by virtue of its method of intercepting motor commands. Moreover, it can also be used with *any deliberative layer* for which the `convertGoal` function is implemented. Finally, the RADIC algorithm is not limited to navigation tasks, but provides a general structure for an action sequencer in hybrid architectures for *any task*.

References

[1] J. Albus. 4-D/RCS reference model architecture for unmanned ground vehicles. Proceedings of IEEE International Conference on Robotics and Automation (ICRA2000), San Francisco, USA, 24-28 April 2000, pp. 3260–3265.

[2] R. Arkin and T. Balch. AuRA: Principles and practice in review. JETAI, Vol. 9, No. 2-3, 1997, pp. 175–189.

[3] T. Balch and R. Arkin. Avoiding the past: A simple but effective strategy for reactive navigation. Proceedings of IEEE International Conference on Robotics and Automation (ICRA1993), Atlanta, USA, May 1993, pp. 678–685.

[4] R. Bonasso, J. Firby, E. Gat, D. Kortenkamp, D. Miller, and M. Slack. Experiences with an architecture for intelligent, reactive agents. JETAI, Vol. 9, No. 2/3, 1997, pp. 237–256.

[5] J. Borenstein, H. Everett, and L. Feng. Mobile robot positioning: Sensors and techniques. Journal of Robotic Systems, Vol. 14, No. 4, 1996, pp. 231–249.

[6] J. Connell. SSS: A hybrid architecture applied to robot navigation. Proceedings of IEEE International Conference on Robotics and Automation (ICRA1992), Nice, France, May 1992, pp. 2719–2724.

[7] R. James Firby. Task networks for controlling continuous processes. Artificial Intelligence Planning Systems, Chicago, USA, June 1994, pp. 49–54.

[8] T. Fong and I. Nourbakhsh. Interaction challenges in human-robot space exploration. ACM Interactions, Vol. 12, No. 2, 2005, pp. 42–45.

[9] E. Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. Proceedings of the 10th National Conference on Artificial Intelligence, San Jose, USA, 12-16 July 1992, pp. 809–815.

[10] E. Gat. On three layer architectures. Artificial Intelligence and Mobile Robots, 1998, pp. 195–210.

[11] R. Jensen and M. Veloso. Interleaving deliberative and reactive planning in dynamic multi-agent domains. Proceedings of the 1998 AAAI Fall Symposium on Integrated Planning for Autonomous Agent Architectures, Orlando, USA, 22-24 October 1998.

[12] K. Konolige, K. Myers, E. Ruspini, and A. Saffiotti. The Saphira architecture: A design for autonomy. JETAI, Vol. 9, No. 1, 1997, pp. 215–235.

[13] J. Kramer and M. Scheutz. GLUE–a component connecting schema-based reactive to higher-level deliberative layers for autonomous agents. Proceedings of FLAIRS2003, St. Augustine, USA, 12-14 May 2003, pp. 22–26.

[14] J. Kramer and M. Scheutz. ADE: A framework for robust complex robotic architectures. Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS2006), Beijing, China, 9-15 October 2006, pp. 4576–4581.

[15] D. Lyons and A. Hendriks. Planning as incremental adaptation of a reactive system. Robotics and Autonomous Systems, Vol. 14, No. 4, June 1995, pp. 255–288.

[16] P. Maes. Situated agents can have goals. Designing Autonomous Agents, 1990, pp. 49–70.
[17] M. Montemerlo, N. Roy, and S. Thrun. Perspectives on standardization in mobile robot programming: The carnegie mellon navigation (CARMEN) toolkit. Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS2003), Las Vegas, USA, October 2003, pp. 2436–2441.
[18] E. Mueller. Natural Language Processing with ThoughtTreasure. Signiform, New York, 1998.
[19] H. Nwana. Software agents: An overview. Knowledge Engineering Review, Vol. 11, No. 2, 1995, pp. 205–244.
[20] M. Scheutz. ADE–steps towards a distributed development and runtime environment for complex robotic agent architectures. Applied Artificial Intelligence, Vol. 20, No. 4-5, Vienna, Austria, 2006, pp. 275–304.
[21] M. Scheutz and V. Andronache. Architectural mechanisms for dynamic changes of behavior selection strategies in behavior-based systems. IEEE Transactions of System, Man, and Cybernetics Part B, Vol. 34, No. 6, 2004, pp. 2377–2395.
[22] M. Scheutz and J. Kramer. RADIC–a generic component for the integration of existing reactive and deliberative layers. Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS06), Hakodate, Japan, 8-12 May 2006, pp. 488–490.
[23] M. Scheutz, P. Schermerhorn, J. Kramer, and C. Middendorff. The utility of affect expression in natural language interactions in joint human-robot tasks. Proceedings of the First ACM Conference on Human-Robot Interaction (HRI2006), Salt Lake City, USA, 2-4 March 2006, pp. 226–233.
[24] M. Scheutz, P. Schermerhorn, C. Middendorff, J. Kramer, D. Anderson, and A. Dingler. Toward affective cognitive robots for human-robot interaction. AAAI 2005 Robot Workshop, Pittsburgh, USA, 9-13 July 2005, pp. 1737–1738.
[25] S. Thrun, D. Fox, and W. Burgard. A probabilistic approach to concurrent mapping and localization for mobile robots. Machine Learning, Vol. 31, 1998, pp. 29–53.

James Kramer is pursuing his Ph.D. in Computer Science and Engineering at the University of Notre Dame, South Bend, IN, where he previously earned his M.Sc in 2005. His primary interests concern the role of system infrastructure as related to agent architectures, focussing on the use of reflective mechanisms in integrating system and cognitive architectures and particularly concerning their use in failure detection and recovery. These ideas are being implemented in the APOC Development Environment (ADE), of which he is a primary developer.

Matthias Scheutz received the M.Sc.E. degrees in formal logic and computer engineering from the University of Vienna and the Vienna University of Technology, respectively, in 1993, and the M.A. and Ph.D. of philosophy in philosophy at the University of Vienna, Austria, in 1989 and 1995 respectively. He also received the joint Ph.D. in cognitive science and computer science from Indiana University Bloomington in 1999. He is an assistant professor in the Department of Computer Science and Engineering at the University of Notre Dame and director of the Artificial Intelligence and Robotics Laboratory. He has over 80 peer-reviewed publications in artificial intelligence, artificial life, agent-based computing, cognitive modeling, foundations of cognitive science, and robotics. His current research interests include agent-based modeling, complex cognitive and affective robots for human-robot interaction, computational models of human language processing in mono- and bilinguals, distributed agent architectures, and interactions between affect and cognition.