Heterogeneous Team Coordination Using Plays

Kristina Rohlin Brett Browning M. Manuela Veloso
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213, USA
{krohlin, brettb, mmv}@cs.cmu.edu

Abstract— **Coordinating the actions of heterogeneous robot teams is a challenging problem, especially when operating in complex domains. In this paper, we develop a framework for coordinating team actions through distributed *team programs*, that we call plays. To be portable our framework aims to be platform and task independent, yet support a rich representation so different tasks can be performed by different robot systems. We present an implementation that achieves these goals by fusing a modern programing language with a play selection and distributed execution management system. We also introduce a general message passing framework for synchronizing actions between teammates. We validate our approach on three different tasks using two different robot platforms, Segway RMPs and Pioneer robots, each with different control architectures. Our results show that the *same* framework can coordinate the teams despite the differences, simply by changing the team programs used.**

I. INTRODUCTION

Heterogeneous robot teams are becoming increasingly popular as researchers recognize the effectiveness of combining specialized mobile robots to create a well equipped team. This trend suggests the development of a *robot-independent* framework for coordinating the actions of such a heterogeneous robot team. To be widely useful, such a framework also needs to be easily reconfigured for different tasks, and different team strategies. Clearly this is a large problem that requires the challenges of task decomposition, role assignment, team formation, plan selection and execution, learning, and behavior recognition to be addressed. While there has been significant work on many of these areas (e.g. for role assignment [1], [2]), we instead focus on the problem of developing a portable framework for coordinating the actions of robots in a team. That is, like [3], [4] we focus on how to select and coordinate the sequence of actions each team member performs.

We start with the idea of a distributed *team program*, which is a recipe of action for each team member for achieving some specific goal. We call these team programs, plays[1], in deference to their similarity to sporting plays and earlier work [5]. Typically, the program or play is only applicable in some situations so team strategy consists of a number of different plays (i.e. a playbook). Coordination then consists of repeatedly selecting and executing plays.

We present three contributions in this paper. First, we introduce a rich team program representation by using a modern programming language, in this case Python[2]. The resulting plays support the usual control flow primitives, variables, functions, etc. Second, we develop a framework for selecting and executing plays in a distributed team that is robot and task independent. The same coordination module can be used in different domains with different robots simply by changing the playbook. In practice, writing plays is much easier than modifying the coordination framework, leading to significant gains. Our third contribution focuses on including a general message passing system between teammates using peer-to-peer communications. In this paper, we focus on synchronizing execution of different team members using this framework.

In the following section, we introduce the concept for team programs that forms the main contribution of this paper. We then focus on the implementation details in Section III. Section IV presents three tasks using two different classes of robot platforms used to validate our approach. Section VI concludes the paper.

II. COORDINATION USING DISTRIBUTED TEAM PROGRAMS

We now define plays, or team programs, and how they are used for platform-independent distributed coordination in a heterogeneous robot team. As we do so, we will endeavor to explicitly define the assumptions our approach relies on and our decision reasoning. We will begin by defining the play language, and the primitives available to a developer for expressing a team program. We will then illustrate how we collect these together to define the overall team behavior.

To help elucidate the discussion, we will begin by describing a simple example – a site visit problem – that we will use throughout this section. Our simple task requires a team of two robots that must visit two sites A and B, where a visit to B is required before returning home. In contrast, a visit to A is desired but is

[1]We will use the terms play and team program interchangeably.

[2]http://www.python.org

non-essential. Fig. 1 shows the problem, and indicates the desired behavior of the robot team members.

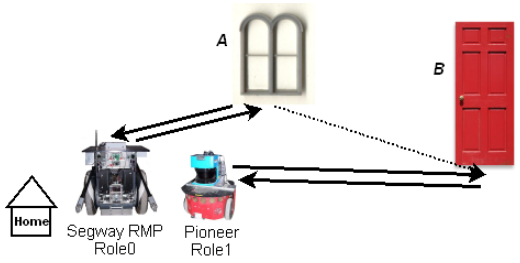

Fig. 1. The figure shows two robots executing a team program for the site visit problem where site A=window and B=door, for example.

TABLE I
TWO-AGENT PLAY FOR THE SITE VISIT PROBLEM

```
PLAYName()
  return "SimpleSiteVisit"
PLAYTask()
  return "visit"
PLAYTimeout()
  return 120

Role0Capabilities()
  return "camera"
Role0Needed()
  return true
Role0(A,B,timeout=30)
  act("goto", A)
  if(!recvMsg(1,100,timeout))
    act("goto", B)
    act("go","home")
  return true

Role1Capabilities()
  return "laser"
Role1Needed()
  return true
Role1(A,B,timeout=30)
  act("goto", B)
  sendMsg(0,100)
  act("go", "home")
  return true
```

A. Play Language

For our discussion, team strategy consists of a set of plays. Each play defines a team program for some flexible team size. This means, that it defines a program of action for each team member as well as additional information required to ensure the play executes robustly and correctly. Table I shows pseudo code for a play to solve the example problem given in figure 1. Note, in the implementation presented later in this paper, this play would more correctly be written in Python, however for brevity and clarity we will focus on pseudo code representations.

1) Definition of a Play: Each play defines, using methods which can be extended in complexity:

- *PLAYName*: A name for the play
- *PLAYTask*: An optional applicability condition
- *PLAYTimeout*: An optional timeout
- A variable number of roles

A play can only be selected if it is *applicable*, meaning that it is appropriate to run given the current task. The *PLAYTask* method defines the set of applicable tasks by returning an array of task names. By evaluating whether the current task is in the *PLAYTask* set, we can determine whether or not the play is applicable. In this way, we can encode a play for a specific, or general, situation as desired. Moreover, we can define multiple plays for the same, or overlapping situations. In addition to applicability, the timeout defines the maximum time expected for the play to execute. If this timeout is exceeded, then the play is considered to have failed, providing a robustness check on the system.

2) Definition of a Role: The core part of the play, is then encoded in each of the roles. Each role is a program of execution for a member in the team. We assume that roles must be assigned to unique robots, although we enable optional roles that may or may not be used. Role assignment is not a focus of this paper and we will assume the existence of a role assignment mechanism. That is, given the play with M necessary roles and K optional roles, and a team of $N \geq M$ robots, we assume that there exists an algorithm available to assign robots to the roles in some near optimal and efficient way.

A role (e.g. role Q) consists of the following methods:

- *Role*Q*Capabilities*: A set of strings defining the capabilities of the robot
- *Role*Q*Needed*: Returns true if this is a necessary role, false if it is not
- *Role*Q: The program of actions for the role

The capabilities and needed methods are used to aid in role assignment. Each robot is tested to see if it has the necessary capabilities. If so, it can be considered as a candidate for the specified role. Needed defines whether the role must be assigned for the play to be able to execute. Hence, the needed method enables the same role to execute for variable team sizes.

The heart of the role is the *Role*Q method that defines the program of actions for the assigned robot. As we derive the play language from a modern programming language, all the usual features of conditional branches, loops, functions, and variables are available to the developer. As a result, it is relatively easy to create a rich, complex program of execution.

3) Role Actions: The role program consists of robot action primitives and play action primitives. Robot actions are the mechanism used by a role to command the robot to perform actions. Each robot action has a unique name, and a variable set of parameters, and returns either true or false depending upon the outcome of the action. We can divide actions into active and passive actions. Active actions are ones where the action can determine intrinsically the success or failure

of the action. For example, driving to a point, is an active action. In contrast, passive actions are ones that are dependent upon some external event happening that is outside of the robot's control. More specifically:

- **act(***name,parameters***)**: defines an action that can determine intrinsically when it is done. It returns success or failure of the action when completed.
- **passive(***name,parameters,roleID,msgID,timeout***)**: defines a passive action that executes until the external event, which is the reception of a message from role *roleID*, with message content *msgID*, or the *timeout* has expired.

4) Play Actions: Play actions differ in that they affect the execution of this or another role within the play. The set of play actions include:

- **wait(***dt***)**: waits for *dt* seconds before continuing. Robot halts during this time.
- **sendMsg(***roleID,msgID***)**: sends a unique message containing *msgID* to the robot executing role *roleID*.
- **recvMsg(***roleID,msgID,timeout***)**: waits for up to the specified *timeout* to receive a message sent by role *roleID* with message content *msgID*. If it receives the message in time it will return true, otherwise it returns false.

The message passing primitives provide communication between agents executing different roles to enable coordinated team behavior. Messages are passed between individual roles using the **sendMsg** and **recvMsg** messages. The content of the message is an integer value chosen by the caller. Message passing primitives make use of peer-to-peer network communications between roles. The peer-to-peer model minimizes the number of agents involved in a single communication by providing direct links between only those agents specified in the primitive and involved in the particular part of the play, rather than routing communication through a central server. In this paper we will focus on message passing as a general method to help solve the synchronization problem. Fig. 2 illustrates the use of message passing in a state diagram of the SimpleSiteVisit play given in Table I.

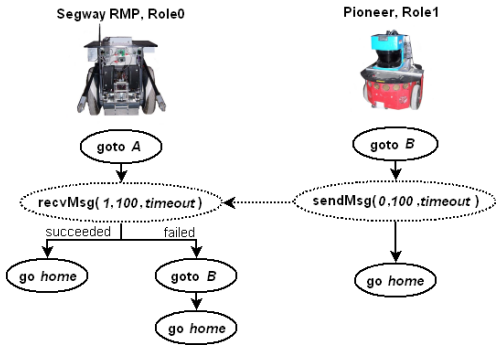

Fig. 2. State diagram of a two role play with synchronization described in Table I and diagrammed in Fig. 1.

*5) **Playbook**:* We conclude the definition of plays with the playbook. The playbook is nothing more than a collection of different plays. With a combination of plays for different situations, a playbook becomes a rich resource for generating a wide range of behavior and results in the complex behavior of the whole team.

III. IMPLEMENTATION

This section describes the components of our current implementation, discussing how each contributes to effective team coordination, and illustrates play execution and role assignment.

A. OVERVIEW OF COMPONENTS

The *PlayManager* and *RoleExecutor* are the high level managerial and execution engines for our coordination architecture. Fig. 3 diagrams the main components of the architecture for a team of N agents.

1) PlayManager: The *PlayManager* oversees execution of a play and enables communications between an external role assignment mechanism (not shown in Fig. 3) and the *RoleExecutors*. The *PlayManager* is the highest level in the hierarchy that directly or indirectly manages the entire system, first by selecting a play, then by monitoring play execution. Each agent on a team employs a *PlayManager*, though there is only a single active *PlayManager* per play.

2) RoleExecutor: A *RoleExecutor* runs on each robot and is responsible for initiating and monitoring role execution and providing status reports to the *PlayManager*. From the *PlayManager*, a *RoleExecutor* receives a play and a list of role assignments, then initiates execution of the role to which it has been assigned. To interpret role actions we employ an *Interpreter*.

3) Interpreter: Since plays are programs designed using a modern programming language (Python), an interpreter is used to transform the play language into executable methods. We utilize methods from the commercially available Python/C interpreter and create a standard Python *Interpreter* for C programs.

4) Callbacks: Callbacks include a set of methods called from within a role to trigger a coordination or action related event. Because a role is a program for execution, rather than a sequential list of actions, *Callbacks* are

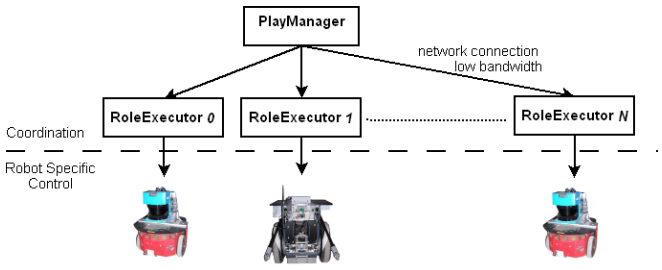

Fig. 3. The figure illustrates the coordination architecture employed for a team of N agents showing the abstraction of coordination from any specific robot control.

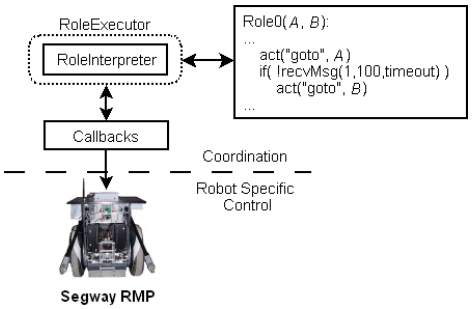

Fig. 4. The figure shows how the *RoleExecutor* interprets a role and communicates with a robot through *Callbacks*.

used to extract information from the program, package it as a message, and send it directly to its robot for execution or to another *RoleExecutor* for peer-to-peer coordination. *Callbacks* provide the direct interface between the robot and the entire coordination module. In this way, roles can contain a variety of methods including action and message passing primitives requiring varying levels of participation from the agent and the coordination module.

B. Life Cycle of a Play

Before play execution, team members must be outfitted with tools for communication. Each robot on the team runs a *RoleExecutor* to open networked communications to the *PlayManager*. *PlayManagers* also run on each robot such that any agent can temporarily assume command for a single play execution. The method for determining the active *PlayManager* at any given time depends on the role and subteam assignment mechanism employed for the particular domain. With a market-based assignment mechanism for example, the agent proposing the lowest bid wins the task and temporarily becomes the active *PlayManager* for that particular play.

Using an *Interpreter*, the *PlayManager* evaluates conditions for all plays in the playbook to determine applicability. The *PlayManager* then selects a play to match a given task or state of the team. The play is then passed to a role assignment mechanism which selects a subteam and role assignment combination and returns assignments to the *PlayManager*. The *PlayManager* contacts *RoleExecutors* running on each of the individual agents to begin execution of the play. Each *RoleExecutor* executes its assigned role by interpreting the program and calling *Callbacks* to send actions to its robot and coordinate synchronized multi-robot actions with other *RoleExecutors*. Fig. 5 diagrams the life cycle of a play with two assigned roles.

C. Subteam Formation and Role Assignment

Subteam and role assignment are an integral part of any play-based model as the active *PlayManager* must know which team members to contact to execute each

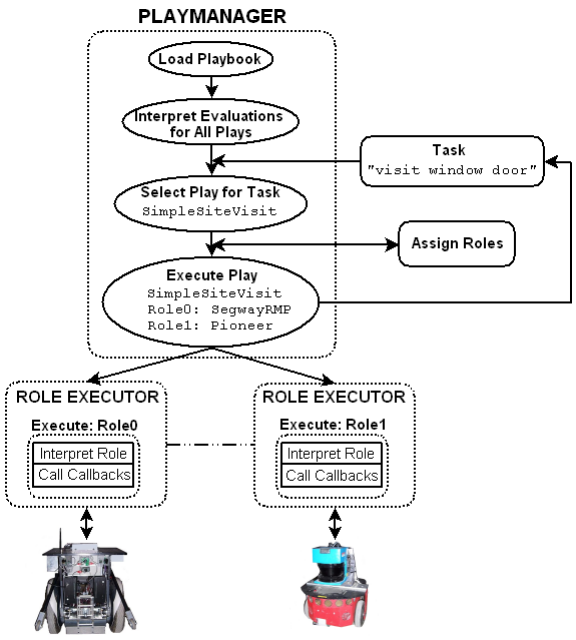

Fig. 5. The figure illustrates the life cycle of a play chosen for a *visit* task from its beginnings within the playbook to execution on individual agents.

role. Because subteam and role assignment are external to the *PlayManager* and can be completed using any accessible method, the *PlayManager* sends a message containing play and role requirements to the assignment mechanism through a networked connection.

One method of assignment, TraderBots, developed by Dias and Stentz [1] is an example of a market-based assignment approach that we have tested with the *PlayManager*. For details on combining this approach with the *PlayManager* to enable dynamic subteam formation, we refer to our earlier work in [6].

IV. Validation

We have tested our model to coordinate teams of autonomous Segway RMPs (Robotic Mobility Platform) developed by Segway LLC [7], Pioneer robots, and human team members, in three very different domains including Segway Soccer, Synchronous Site Patrol, and Treasure Hunt domains.

We show effective coordination of real heterogeneous robots through video demonstrations. [3]

A. Segway Soccer Passing

In the Segway Soccer domain, teams of Segway RMPs and humans riding Segway Human Transporters play a game of soccer. In this dynamic and adversarial environment, it is important to maintain possession of the ball whenever possible to deny the opposing team a chance to score. By synchronizing actions, a passer

[3]The accompanying video illustrates the soccer and patrol plays. All videos are available at: http://www.cs.cmu.edu/~coral-downloads/segway/movies/06-12-plays/

TABLE II

TWO-ROBOT PASS AND SHOOT PLAY

```
...
Role0(timeout=30)
  act("grab","ball")
  if(act("search", "teammate"))
    sendMsg(1,100)
    recvMsg(1,200,timeout)
    act("kickto","teammate")
  else act("kickto", "goal")
  return true
...
Role1(timeout=30)
  act("grab", "ball")
  recvMsg(0,100,timeout)
  if(act("search", "teammate"))
    sendMsg(0,200)
    act("catchkickto", "goal")
  else act("goto", "goal")
  return true
```

TABLE III

TWO-ROBOT SYNCHRONOUS PATROL PLAY

```
...
Role0( A,B,timeout=30 )
  while(PLAY_TIMEOUT)
    act("goto", A)
    act("search", B)
    sendMsg(1,200)
    act("goto", B)
    act("go", "home")
    sendMsg(1,300)
    recvMsg(1,400,timeout)
  return true
...
Role1( A, B, timeout=30 )
  while(PLAY_TIMEOUT)
    recvMsg(0,200,timeout)
    act("goto", A)
    act("search", B)
    sendMsg(0,400)
    recvMsg(0,300,timeout)
    act("goto",B)
    act("go", "home")
  return true
```

can inform a receiver prior to the pass to ensure the receiver is ready to receive, minimizing the chance for a free ball and overturned possession. A play can easily be designed to implement this coordination strategy using message passing and action primitives. Table III illustrates the play program designed for two agents, and the corresponding video shows two Segway RMPs executing the play.

B. Synchronous Site Patrol

The Synchronous Site Patrol task involves two agents consecutively and continuously patrolling three target sites. We can easily design a play for this task to sequence site visits by synchronizing role events such that each team member sequentially visits each site in turn. Using message passing primitives, we can ensure that no two team members will visit the same site at the same time. Table III illustrates the play program designed for two agents performing a continuous set of actions until a PLAY_TIMEOUT and sequencing actions using message passing primitives. This program enables continuous action through the use of loops and sequenced coordination through peer-to-peer message passing. The play is performed by two Segway RMPs equipped with cameras for visual identification of targets, marked by pioneer robots.

C. Treasure Hunt with Dynamic Heterogeneous Subteams

The aim of the Treasure Hunt Domain is to build a single heterogeneous human-robot team capable of effectively locating objects of interest (treasure) spread over a complex, previously unknown environment [6]. This domain involves teammates combining individual resources to achieve a task that no single team member is able to accomplish alone. The team is composed of a human team member and two robot platforms, including modified versions of the Pioneer robot and Segway RMP. The two robot platforms run very different processes to control the functions of the specific platforms; however they are both able to interpret the generalized action messages used in the play. Additionally, there are three different research groups collaborating to create this heterogeneous team. The *PlayManager* proved to be an effective tool for communication amongst different platforms and for reasonably quick formation of these teams including robots developed by different research groups.

The video illustrates a sequence of three team tasks including an area exploration, treasure search, and treasure retrieval. The first task is the exploration of a specified area directed by the human and performed by two pioneers. Each pioneer executes an independent exploration play with a single role. The second task is the organization of a search subteam including multiple robots with different capabilities (e.g. Segway with camera and Pioneer with local area map data) to search for treasure. As discussed in Section III-C, the TraderBots provide the mechanism for team and role assignment and combined with the *PlayManager* allow for the dynamic formation of this two-robot subteam. The executed play involves two roles, one which maps a set area and is performed by the Pioneer, and the other which follows the Pioneer searching for treasure, and is performed by the Segway. Following coordination for this play is done at the behavior level, illustrating that our architecture does not preclude execution of behaviors involving embedded coordination. Once a treasure is found, the subteam generates a retrieve task won by the available Pioneer. Finally, the Pioneer executing a retrieve play leads a human to obtain the identified treasure and return it to the home base.

V. RELATED WORK

The work we have described in this paper is related to the notion of of team plans [3], [4], or similar ap-

proaches of [8], where we have extended this concept to a much richer programming representation and incorporated the message passing primitives. Additionally, our approach is aimed at being platform-independent by making as few as possible assumptions about the nature of the underlying robot control architectures.

Our work clearly builds on previous play-based coordination approaches [5], [9], but differs significantly in that *(a)* the coordination mechanism is robot and task independent, *(b)* a much richer play language is used, *(c)* message passing is used as a general synchronization mechanism. Earlier play-based approaches used a centralized system [5]. McMillen and Veloso [9] extended this approach to a distributed selection mechanism. Our work differs in that the team executes the play in a distributed fashion, but play selection, instantiation, and monitoring are all performed by a single agent – the play manager.

Much of multi-robot literature focuses on the problem of task or role assignment. In our approach, role assignment is a key part of play execution, but the mechanism used to assign the roles is interchangeable. Indeed, our earlier work [6] made use of a market-based approach [1] for role assignment. Finally, our approach does not preclude the use of tight coordination mechanisms (e.g. [10], [11]), provided these mechanisms can be suitably described by an action string with associated parameters.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we introduce a concept for heterogeneous team coordination using distributed team programs. We illustrate three main contributions, namely (a) a rich team program representation using a modern programming language (b) a task and robot-independent coordination framework enabling the same coordination module to be effective across many robots in different domains and (c) a message passing peer-to-peer communication model employed to enable synchronized team actions.

We show video results in three different domains using two different robot platforms and a human team member. Future work will include improving robustness and reliability, expanding our play language primitives to include a robot's perceived information, evaluating the model within other domains, and designing an interface to integrate humans as proper peers.

VII. ACKNOWLEDGMENTS

This research was sponsored in part by the Boeing Company under Grant No. CMU-BA-GTA-1 and in part by the United States Department of the Interior under Grant No. NBCH-1040007. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing official policies or endorsements, either expressed or implied of the Boeing Company or the US Department of the Interior.

REFERENCES

[1] M. B. Dias, "Traderbots: A new paradigm for robust and efficient multirobot coordination in dynamic environments," Ph.D. dissertation, Robotics Institute, Carnegie Mellon University, 2004.

[2] L. Parker, "Alliance: An architecture for fault-tolerant multi-robot cooperation," *IEEE Transactions on Robotics and Automation*, vol. 14(2), pp. 220–240, 1998.

[3] M. Tambe, "Towards flexible teamwork," *Journal of Artificial Intelligence Research*, vol. 7, pp. 83–124, 1997. [Online]. Available: citeseer.ist.psu.edu/tambe97towards.html

[4] P. Scerri, D. Pynadath, N. Schurr, A. Farinelli, S. Gandhe, and M. Tambe, "Team oriented programming and proxy agents: The next generation," in *Proceedings of 1st international workshop on Programming Multiagent Systems*, 2004.

[5] M. Bowling, B. Browning, and M. Veloso, "Plays as effective multiagent plans enabling opponent-adaptive play selection," in *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS'04)*, 2004, under submission.

[6] E. Jones, B. Browning, M. B. Dias, B. Argall, M. Veloso, and A. T. Stentz, "Dynamically formed heterogeneous robot teams performing tightly-coordinated tasks," in *International Conference on Robotics and Automation*, 2006.

[7] H. G. Nguyen, J. Morrell, K. Mullens, A. Burmeister, S. Miles, K. Thomas, and D. W. Gage, "Segway robotic mobility platform," in *SPIE Mobile Robots XVII*, October 2004.

[8] R. Simmons, T. Smith, M. B. Dias, D. Goldberg, D. Hershberger, A. T. Stentz, and R. M. Zlot, "A layered architecture for coordination of mobile robots," in *Multi-Robot Systems: From Swarms to Intelligent Automata, Proceedings from the 2002 NRL Workshop on Multi-Robot Systems*, 2002.

[9] C. McMillen and M. Veloso, "Distributed, play-based coordination for robot teams in dynamic environments." in *Proceedings of the RoboCup International Symposium*, 2006.

[10] N. Kalra, D. Ferguson, and A. T. Stentz, "Hoplites: A market-based framework for planned tight coordination in multirobot teams," in *Proceedings of the International Conference on Robotics and Automation*, 2005, pp. 1170–1177.

[11] G. Pereira, V. Kumar, and M. Campos, "Decentralized algorithms for multi-robot manipulation via caging," *International Journal of Robotics Research*, 2004.