The 2010 IEEE/RSJ International Conference on
Intelligent Robots and Systems
October 18-22, 2010, Taipei, Taiwan

Object Interaction Language (OIL): An Intent-based Language for Programming Self-Organized Sensor/Actuator Networks

Daniel J. Sutton, Peter T. Klein, Michael W. Otte and Nikolaus Correll

Abstract— **This paper introduces the Object Interaction Language (OIL) that allows programming and coordination of distributed, heterogeneous sensor-actuator networks, such as sensor networks and multi-robot systems. OIL is an interpreted, object oriented language and is contained in an OIL environment. An OIL environment provides communication between agents and allows agents to exchange code snippets among each other. Possible implementations of OIL environments can be — in the simplest case — a sheet of paper with OIL code literally printed on it, or a computational agent endowed with sensors, actuators and wireless communication. The atomic primitive in OIL is the intent for which implementation is resolved during runtime, potentially using code from other OIL environments and leading to distributed execution. We develop the structure of the language and demonstrate its key properties using a distributed computation task that is parallelized via an OIL environment. We evaluate the algorithm empirically by running OIL code on a team of six computational agents that communicate wirelessly. We then show experimentally how OIL can be used to allocate sensing and mobility in a multi-robot system using a case study in navigation, where one robot dynamically provides laser range data to another robot which is blind to its environment.**

I. Introduction

We wish to design formalisms and algorithms that allow for efficiently distributing sensing, computation, actuation and communication into an environment for automating chores at home and at work. We believe that interaction between computational agents and everyday objects such as appliances, furniture, and even tableware, e.g., would be tremendously facilitated when every object could share a description of its functionality and algorithms as well as its physical properties that supports the computational agent's perception and manipulation capabilities. Imagine a house with a set of mobile manipulators, mobile trays, cups, and a coffee-machine given the task to deliver a cup of coffee to a user. Instead of implementing this task using a single robotic agent, which interacts with a passive environment, we are interested in a distributed, self-organised task execution in which each agent contributes domain specific algorithms, sensing and actuation: the coffee-machine could orchestrate a mobile manipulator to pick up a cup, place it under the coffee machine's outlet, remove it once the coffee is brewed, place it on the tray, and have the tray finally deliver the coffee to you. In this example some of the sub-tasks can be executed in parallel, while others need to be executed sequentially.

We believe that this approach in which each agent provides the code and data structures that are relevant to its operation would drastically lower the perception abilities required from a single agent interacting with passive objects. Also, such an organization would be truly scalable as it can be extended by agents offering different functionality and algorithms, and it is robust as some functionality might be redundant in the system.

Towards this end, we envision each agent to be represented by a class in an object oriented programming (OOP) sense, which offers a series of methods and variables. Looking closer at our coffee-brewing example, namely the task of picking up the cup, we could imagine literally printing code for a *pick-up* method on the cup, which contains the object geometry and instructions on how to grasp it.

A. Contribution of this paper

We examine how individual agents' intentions to execute particular actions of complex tasks that are composed of sequential and parallel elements might drive behavior and communication in a multi-agent system. To this end, we provide a formal description, algorithms and an implementation of an interpreted, object oriented language *Object Interaction Language* (OIL) that automatically includes code provided by other agents in the system at run-time, selects the "best" implementation among comparable agents, allows for parallel execution of sub-tasks, and is robust to agent failure due to an exception handling mechanism. We empirically investigate the scalability limitations of a self-organized computation task on a team of six physical agents that communicate over an unreliable, stochastic wireless network. Finally we demonstrate how OIL might be used with real robots with a navigation task in a heterogeneous team of robots.

B. Outline of this paper

After concluding the introduction with an overview of related work, we provide an informal overview of the OIL language in Section II and a formal definition of concepts and algorithms in Section III. We then illustrate the developed concepts using computation and robotics examples in Section IV and evaluate them empirically in Section VI. Limitations, applications and further work are then discussed in Section VII.

D. Sutton, M. Otte, and N. Correll are with the Department of Computer Science, University of Colorado at Boulder, Boulder, CO 80309, USA firstname.lastname@colorado.edu

P.Klein is with the Department of Aerospace Engineering, University of Colorado at Boulder, Boulder, CO 80309, USA peter.klein@colorado.edu

978-1-4244-6676-4/10/$25.00 ©2010 IEEE

C. Related work

Self-organizing agent organisations, i.e. agent organizations that coordinate which agent is doing which task in a fully decentralized manner, has been brought forward as powerful alternative to centralized coordination for multi-agent systems [2], [11]. Advantages of self-organisation are that the solution is scalable due to the absence of a central agent that would eventually become a bottleneck of the system, and robust due to the absence of a single point of failure and potential redundancy in the agent population. Disadvantages are that a self-organized approach to coordination cannot always guarantee optimal allocation of resources. Kota et al. [6] present a self-organized agent framework where agents communicate service requests to their peers and compare the resulting performance of a simulated system with a centrally planned and a randomized approach. Deloach et al. [4] present a similar architecture that explicitly employs re-organization for responding to changing environments and agent failure.

Other approaches to team formation for heterogeneous agents include auction based algorithms [5], [1]. In [5], a box pushing task is carried out by a team of three heterogeneous robots and can only be achieved by the three robots joining their capabilities. All sub-tasks are then auctioned off sequentially and are bid on only by robots that have the appropriate capabilities. In [1], tasks are allocated redundantly among temporarily disconnected sub-teams, and tasks are re-allocated in response to agent failure.

Examples of real-world distributed physical agent systems that are strongly heterogeneous in terms of computation, sensing, actuation and communication and that are controlled in a distributed self-organizing manner are the "PEIS ecology" [10] and the distributed robotic garden [3]. In [10] a series of agents ranging from appliances to mobile robots are coordinated via a virtual environment abstraction that includes remote inspection of agent properties. In [3], tomato plants equipped with wireless routers coordinate robots for watering tasks and also store information about the plants that is then used for manipulation.

II. OIL: Overview

The Object Interaction Language (OIL) is an object-oriented, interpreted language that subsumes Python. Python has been chosen as it allows the environment to include code at run-time and offers features to look up classes and their properties. Therefore, OIL code can be executed on any platform that provides a Python interpreter. In fact, Python suits OIL so well that the current OIL implementation fuctions much like the C preprocessor: OIL directives in python source files are uesd at compile time to generate Python code that implements OIL. OIL also allows one to "program" objects that do not have any computational capabilities. In this case, OIL code can provide algorithms and data structures that are specific to this object and are used by computational agents interacting with this object.

OIL code is executed in an *OIL environment*, which is the union of all OIL environments an agent has previously interacted with. Possible examples of such interactions are wireless communication, detecting an RFID tag or simply reading code from objects using a camera.

For the example of picking up a cup, OIL code provided by the cup is then executed by a manipulator once it has perceived the cup and parsed its code contribution. The obvious advantage of this approach is that existing agents in the system will be immediately able to interact with objects and agents never encountered before as all relevant algorithms and information are stored directly on the object they describe.

OIL extends the object oriented programming paradigm by four concepts: intent, service discovery, intent resolution, and intent collections. These concepts are formally defined in Section III and illustrated by examples in Section IV. In practice, OIL code, consisting of a sequence of intents, is pre-processed into algorithms that implement concepts such as intent discovery, intent resolution, intent collections and exception handling transparently to the user.

III. Formal definition

OIL is an object oriented language that subsumes the Python language [9]. OIL extends Python by the following concepts intents, intent discovery, intent resolution, and intent collections.

Definition 1 (Intent): An *intent* ι is a high-level language primitive of OIL. An intent corresponds to a computing, sensing, actuation, or communication task that is implemented by at least one agent in the system. An intent can be a sequence of other intents as well as any high-level language primitive defined by the Python language [9]. We refer to the set of Python-based intents as $\mathcal{P}$ and the set of intents provided by OIL as $\mathcal{O}$.

Each intent ι has one ore more implementations. An implementation is a three-tuple (I, t, q) consisting of OIL code I, a time-stamp t corresponding to when the implementation can be made available by the agent, and a scalar quality metric q, which can refer to the computational complexity of a particular algorithm or the speed of a robotic agent, e.g.

We refer to the set of implementations as an *OIL environment* $\mathcal{E}_i$. Formally,

$$\iota \rightarrow (I, t, q) \in \mathcal{E}_i \subseteq \mathcal{O} \supseteq \mathcal{P} \tag{1}$$

Each time OIL interprets a statement, it needs to resolve where the intent is implemented. We refer to this process as *intent discovery*. An implementation contains code that leads to local or remote execution. This is transparent to the agent executing the intent.

Definition 2 (Intent discovery): Intent discovery is the process of determining whether an intent ι is implemented by agent i, i.e. $\iota \in \mathcal{E}_i$, or discovering its implementations by other OIL environments. We refer to the set of implementations that satisfy intent ι as $\mathcal{I}_\iota$. Formally,

$$\mathcal{E}_i \rightarrow \mathcal{E}_i \cup \mathcal{I}_\iota \tag{2}$$

The '$\rightarrow$' operator is defined by Algorithm 1.

Algorithm 1: INTENT DISCOVERY (PSEUDO CODE)

```
   Data: OIL high-level language primitive (intent) ι
   Result: set of OIL implementations of ι: I_ι
1  if ι ∉ P then
2      I_ι ← ∅
3      while I_ι = ∅ && !timeout do
4          Responses ← broadcast(ι)
5          foreach i ∈ Responses do
6              I_ι ← I_ι ∪ { i }
7      E_i ← E_i ∪ I_ι
8      if timeout then
9          // cannot resolve intent (user exception)
```

Algorithm 1 is executed by the interpreter at run-time and first checks whether the intent is part of the Python implementation (line 1). If not, the agent broadcasts the intent and processes all responses (line 4). This is repeated until the responses provide at least one implementation or a user-defined time-out occurs (line 3). The received implementations are then added to the local OIL environment (line 7).

Notice that intent discovery returns OIL code that is composed of intents that might again need discovery steps. We refer to this concept as *nesting* and OIL can be infinitely nested and also recursive.

Once, possible implementations of an intent have been discovered, the agent needs to resolve which implementation of an intent to choose. We refer to this process as *intent resolution*. Differences among intent implementations can have multiple reasons, e.g. agents having different computational power or sensor and actuator precision. These differences are represented by the implementation *quality* q (see Definition 1). Other important differences between implementations could be the availability of an agent, e.g. in case an agent is busy. This property is represented by the implementation's availability time stamp t (Definition 1).

Definition 3 (Intent resolution): Intent resolution is the process to find the optimal implementation I of intent ι from the set $\mathcal{I}_\iota$ that results from intent discovery. In this paper, an optimal implementation is an implementation that will be available in the shortest possible time. If there are multiple implementations with the same delay available, the intent with the best quality q is selected. The quality metric is not the focus of our current research, and so it is intentionally abstract in this description. For an implementation $\iota \rightarrow (I, q, t)_i$ provided by agent i, we can say

$$\iota \in \mathcal{E}_i, \qquad i = \arg\min_j \{t | (I, q, t)_j\} \tag{3}$$

The '$\arg\min$' operator is defined by Algorithm 2.

Algorithm 2 finds the implementation in $\mathcal{I}_\iota$ that leads to the lowest execution delay using a systematic search (line 3). If there are multiple implementations offering the same delay, the algorithm will select the implementation with the best quality (line 4). At the discression of the programmer ,OIL code can ues the mutex capabilites in the Python language to

Algorithm 2: INTENT RESOLUTION (PSEUDO CODE)

```
   Data: set of OIL implementations that satisfy ι: I_ι ⊆ E_i
   Result: OIL implementation of ι: I
1  BestSoFar ← ∅
2  foreach (I, q, t) ∈ I_ι do
3      if {t — (I,q,t)} - CurrentTime < AcceptableDelay then
4          if {q|(I, q, t) > quality(BestSoFar) then
5              BestSoFar ← (I, q, t)
6  i ← {i|BestSoFar}
7  if BestSoFar = ∅ then
8      // Timing constraints not matched (user exception)
```

Algorithm 3: INTENT COLLECTION (PSEUDO CODE)

```
   Data: set of OIL intents ι_1, ..., ι_n
   Result: set of OIL intent results r_1, ..., r_n
1  R ← ∅
2  forall ι_j ∈ ι_1, ..., ι_n do
3      I_ι ← Discovery(ι_j)
4      I ← Resolution(I_ι)
5      R ← R ∪ (r_j ← execute(ι_j, I))
6  while (|R| < n) do
7      // wait for all tasks to complete
```

ensure that resource allocation conflicts are managed cleanly. The algorithm completes in time $\mathcal{O}(\|\mathcal{I}_\iota\|)$ with $\|\mathcal{I}_\iota\|$ the number of implementations received. Notice that Algorithm 2 throws an exception only if no agents can provide an implementation with an accepted delay as $\|\mathcal{I}_\iota\| \geq 1$ after discovery (see Algorithm 1).

Although the concepts intent discovery and resolution allow for a distributed implementation of a given OIL script, they are not enough for implementing parallel execution. For this reason, we introduce the concept of an *intent collection*, which contains intents that can be executed in parallel.

Definition 4: An *intent collection* is a set of intents $[\iota_1, \ldots, \iota_n]$ whose execution is independent from each other. Intents that are part of a collection $[\ldots]$ are then discovered and executed in parallel. The '$[\ldots]$' operator is defined by Algorithm 3.

Algorithm 3 collects the results from a set of OIL intents in a data structure. For this, each intent is discovered, resolved and executed in parallel (line 2). The algorithm waits then until all intents are completed. Notice that exceptions are handled during discovery and execution and are transparent to intent collection. That is, an intent collection is always complete, unless discovery or execution fail permanently with a user exception.

In a multi-agent system, it might be that intent discovery fails or that intents become unavailable after their resolution. We thus define the concept of an exception.

Definition 5 (Exception): An *Exception* is a condition $E(\iota, i)$ that caused the execution of ι to fail and broke the expected control flow of the intent implementation i. In this case, the execution of ι might revert to the INTENT DISCOVERY or INTENT RESOLUTION operations. OIL handles

Algorithm 4: EXCEPTION HANDLING (PSEUDO CODE)

Data: an OIL intent ι and implementation I causing the exception
Result: execution or rejection of OIL intent ι

1 $\mathcal{I}_\iota \leftarrow \mathcal{I}_\iota \setminus I$
2 **if** $\mathcal{I}_\iota \equiv \emptyset$ **then**
3 $\mathcal{I}_\iota \leftarrow$ Discovery(ι_j)
4 **if** $\mathcal{I}_\iota \equiv \emptyset$ **then**
5 $\mathcal{E}_i \leftarrow \mathcal{E}_i \setminus \iota$
6 // cannot find replacement for ι (user exception)
7 **else**
8 $i \leftarrow$ Resolution($\mathcal{I}_\iota$)
9 execute(ι, I)

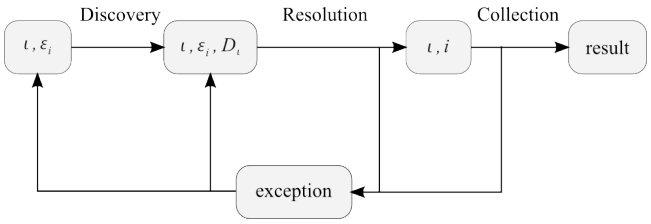

Fig. 1. Finite state machine generated by the OIL interpreter showing the interplay of Algorithms 1–4. An intent ι is broadcasted in the discovery step and leads to the set of implementations available in the team $\mathcal{I}_\iota$. The agent then resolves a suitable implementation, which might lead to an exception if the implementation has become unavailable in the mean time. Intents are then executed and their results collected.

exceptions in a way that ensures that either ι is satisfied eventually, or the intent is unsatisfiable in the current OIL environment, which leads to an *user exception*, telling a user or higher-level agent the reason for failure. This process is governed with timing parameters that are a parameter of the OIL language implementation (see Algorithms 1 and 2). The '$E()$' operator is defined by Algorithm 4.

Algorithm 4 will first remove the implementation I causing the exception from the set of implementations $\mathcal{I}_\iota$. In case, no other implementation is known (line 2), the agent will try to discover new implementations (line 3), and resolve a new implementation otherwise (line 8).

A finite state machine that is generated by the OIL interpreter showing the interplay of Algorithms 1–4 is depicted in Fig. 1.

IV. EXAMPLE: AN OIL ALGEBRA

This section demonstrates the OIL concepts of intent nesting and recursion, and parallelization using OIL programs that lead to distributed implementation of intuitive algorithms such as calculating the square root of a number recursively or enumerating prime numbers. These algorithms rely on basic algebraic operations, which we regard as OIL intents. We consider these example representative for a wider range of problems in the domain of distributed computation and robotics.

Let us assume that the OIL environment provides $+$, $-$, $*$ and $/$. We can now use these basic intents to write implementations of higher-level intents such as calculating the square root of a given number recursively, or enumerating all prime numbers in a given interval. We selected these examples as they are intuitive and demonstrate the nesting and parallelization capabilities of OIL.

An example of a nested and also recursive algorithm in OIL is an implementation of an algorithm to find the square root of a number:

```
intent SquareRoot(num, estimate=None):
   if estimate:
       sqrt = estimate
   else:
       sqrt = num
   if abs(sqrt * sqrt - num) > epsilon:
      return SquareRoot(num,
            (sqrt + num / sqrt)/2.0)
   else:
      return sqrt
```

Notice that the OIL syntax is identical to that of Python. In OIL, however, *every* language primitive of OIL is treated as an intent. For demonstrating the collection concept, we chose the task to enumerate all primes in an interval as it is well known algorithm that is fully parallelizable. Given an intent isPrime(x) which returns a boolean indicating if x is prime, the intent `enumeratePrimes` now demonstrates the collection concept:

```
intent enumeratePrimes(lower, upper):
    c = collection( (isPrime,i)
            for i in range(lower, upper))
    c.execute()
    primes = []
    for i in range(lower, upper):
        if c.result(i):
            primes.append(i)
    return primes
```

V. THE PRAIRIEDOG PLATFORM

The Prairiedog platform is a robotics platform that is based on the iRobot Create for mobility, with a Hagisomic Stargazer for localization and a Hokuyo laser scanner for environment sensing. Computation and communication is provided by a netbook carried by the robot. The software layout of the Prairiedog is organized as a ROS stack containing a number of independent ROS nodes that communicate via messages. Each external sensor has its own ROS node that exposes the functionality to the rest of the system. A planning node (`base_planner_cu`) calculates the most efficient path between the robot and a goal using a modified version of the Field D* algorithm [7]. A dedicated goal server (`goal_server_cu`) provides on-demand goal information to the rest of the system. Additional nodes are currently being developed for interaction with the mechanical arm, and multi-robot path-planning.

VI. RESULTS

Computational experiments are carried out on 6 netbook computers that are communicating wirelessly in ad-

hoc mode. Robotics experiments are carried out on the Prairiedog Platform, which is controlled by the same netbook computers. Each netbook runs an OIL environment and announces intents via UDP broadcast. In case another agent can service an intent, agents share code using a dedicated TCP connection. We opted for validation of the OIL concepts on a real system rather than simulation to demonstrate its distributed properties as well as to understand the scalability and robustness of self-organized organization under influence of real-world challenges such as unreliable communication channels.

Notice that coordination using wireless communication is only one possible implementation of exchanging intent implementations. Reading implementations from agents equipped with RFID tags or bar codes would be also possible, but would not allow us to demonstrate the remote execution capabilities of OIL. In practice, code that is supposed to be remotely executed contains commands that send function names and arguments over the network and receive the return value of the computation in response.

A. Parallel Execution

We are interested in experimentally validating the collection concept of OIL implemented by Algorithm 3. We also would like to find out the overhead resulting from wireless coordination by measuring the speed-up resulting from parallelization. In this experiment all agents have the same capabilities, i.e. each agent can service all OIL algebra intents.

We run a series of experiments in which one agent launches the prime number enumerating algorithm for numbers from 10^{10} to $10^{10}+10^3$ using the collection concept of OIL and dividing the task in 6 chunks. We run this algorithm with team sizes ranging from one to six agents incrementing the number of agents in the system by one in each experiment (20 experiments per team size, 120 experiments total).

Running this algorithm on a single machine took 38.2s in average. Fig. 2 shows the speed-up for parallelization. We observe super-linear performance for 2–4 agents, which we believe is due to artifacts of memory organization in Python (customary in parallel programming [8]). The speed-up is sub-linear for more than 4 agents, which we believe is due to collisions on the wireless channel and reduced throughput due to other parallelization overhead with a growing number of agents.

B. Multi-Robot Navigation

In a real-world robotics application, we use OIL to enable a robot with no environmental sensing capabilities (the blind robot) to cross an obstacle-filled room with the aid of a second robot which could provide laser scans of an area (the sensing robot). When the blind robot needs new information about objects in the environment, it queries the OIL environment to discover agents that implement a laser scanning intent. The sensing robot responds to this query with an intent implementation that can perform a laser scan of the environment if a desired location from which to take

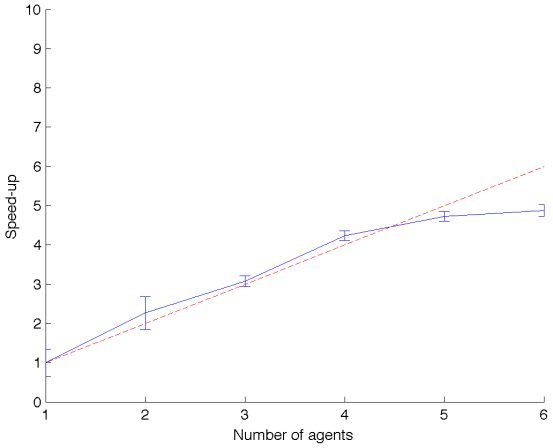

Fig. 2. Speed-up for parallel computation of prime numbers from 1 to 6 agents. Homogeneous agents. 20 experiments per team size. Errorbars show the standard deviation. Red line shows linear speedup.

the scan is specified. The blind robot can execute a laser scan intent every time it needs more data about the environment, and the laser scans will be implemented by the sensing robot. In our experiment, we limit the range of the laser scanner to 1 m, which means that every time the blind robot requests a laser scan, it receives information on all obstacles that are within 1 m of the requested scan position. The blind robot then plans a path to avoid these obstacles, and moves along this path until it approaches an area that has not been scanned yet, at which point it will request a new scan of the unknown area. In order to allow for successful path planning, the Blind robot stops at least 1 m before the unknown area to request a new scan. In this way, communication only via OIL intents leads to an emergent behavior where the seeing robot efficiently leads the blind robot through the obstacle course. This is possible because OIL intents allow the sensing robot's laser scanner capabilities to be used by the blind robot as a seamless extension of its own functionality. Fig. 3 shows the position of the robots as they navigate through an obstacle course in a typical run of this experiment. The video that accompanies this paper shows a more complete animation of this experiment, and a second one with a similar setup.

VII. Discussion

Intent resolution has been implemented using a systematic search algorithm that optimizes the time delay for which an implementation is available. In a productive system, we could also imagine using a combined metric. For example, consider the implementation of the $\otimes$ intent as sequence of $\oplus$ intents with complexity $\mathcal{O}(n)$ vs. an implementation with the native '$*$' operator (that runs like '+' in a single instruction on most machines). In this case, the quality q would be 1 and n, respectively, and selecting the slower implementation might make sense when the faster implementation has a certain delay. Similar cases can be made for robotic agents that differentiate themselves by speed.

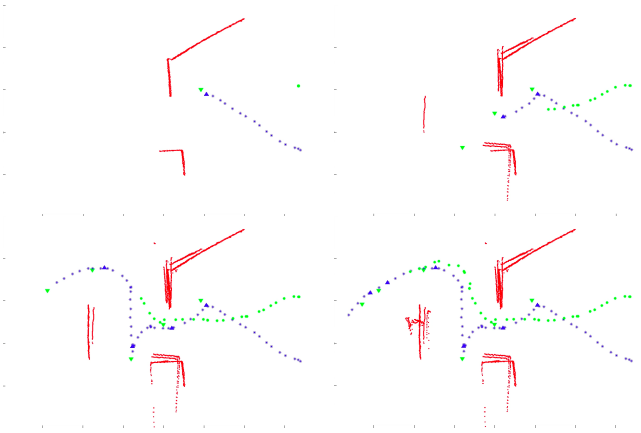

Fig. 3. Multi-robot navigation across an obstacle filled room, detailed step diagrams. Progress from initial to final is left to right, top to bottom. Scale tick marks represent 0.5 m increments. Circles (green) represent positions of the blind robot, and stars (blue)represent positions of the sensing robot. Upside down triangles (green) are the desired scan position of the blind robot, and right side up triangles (blue) are where the sensing robot actually performed the scans. The small squares (red) are the hits returned by the laser scanner indicating an obstacle is present.

In this paper, intent discovery exclusively relies on wireless broadcasting. We are interested in using OIL also for the interaction of passive objects with robots. In this case, we can imagine to extend the discovery algorithm by physical search, e.g. a robot systematically scanning an environment. Other possible implementations would be agents whose sole purpose is to provide discovery for passive agents, such as shelves that keep track of dishes that they store, e.g.

The nesting and parallelization capabilities of OIL allow for the design of powerful systems with emergent behavior. A possible example would be a robotic agent that empties and re-shelves an entire cupboard in order to reach a specific item. This property might make OIL code difficult to maintain, however, and can lead to unpredictable results. Potential pitfalls include infinite recursions, circular resolutions, and repeated discovery-exception cycles. In further work we are interested in investigating verification techniques that can automatically detect such problems given all the implementations of an OIL environment and outline its potential failure modes.

Algorithms presented in this paper such as discovery and resolution scale linearly in execution time with the number of agents in the system, i.e. with $\mathcal{O}(n)$ for n agents and with $\mathcal{O}(nm)$ for task sequences with m subtasks. Kota et al. [6] present algorithms that use heuristics for task selection that provide better performance for large agent population. We believe, however, that for applications in which communication is performed wirelessly with limited range, such as in a home automation setting, the effects of channel depletion quickly outweigh the computational complexity of the coordination algorithm (see also the results in Fig. 2).

An application of OIL in a multi-robot navigation problem allows a robot which has otherwise been deprived of environmental sensing capabilities to use the sensors provided by other robots. With one OIL intent implemented on the sensing robot and utilized by the sensor-deprived robot, an emergent behavior develops where the team of robots efficiently navigate an obstacle course. Applying this kind of set up to other situations could allow other teams of robots or smart objects to spontaneously share sensing or actuation capabilities as tasks arise. The emergent behaviors in these situations may allow a team of robots to accomplish a complex task by communicating solely via simple intents.

VIII. Conclusion

We presented a self-organizing agent model that allows for serial and parallel task decomposition for teams of heterogeneous agents. Agents process their tasks sequentially and recruit other agents by broadcasting intents for subtask that they cannot solve themselves. Individual failure is mitigated by an exception handling mechanism. We also presented a language implementation, the Object Interaction Language (OIL), that allows a user to write code describing the behavior of the entire system and coordinating the agents transparent to the user. We empirically evaluated an implementation of OIL and demonstrated load distribution in the agent system for recursive and parallel task sequences. We also showed that the coordination mechanism is robust to agent failure and communication loss. We demonstrated that OIL can be used to coordinate a real-world multi-robot navigation task with relative ease. An implementation of OIL will be made available open-source.

References

[1] P. Amstutz, N. Correll, and A. Martinoli. Distributed boundary coverage with a team of networked miniature robots using a robust market-based algorithm. *Annals of Mathematics and Artifcial Intelligence. Special Issue on Coverage, Exploration, and Search, Gal Kaminka and Amir Shapiro, editors*, 52(2–4):307–333, 2009.

[2] C. Bernon, V. Chevrier, V. Hilaire, and P. Marrow. Applications of self-organising multi-agent systems. *Informatica*, 30(1):73–82, 2006.

[3] N. Correll, N. Arechiga, A. Bolger, M. Bollini, B. Charrow, A. Clayton, F. Dominguez, K. Donahue, S. Dyar, L. Johnson, H. Liu, A. Patrikalakis, T. Robertson, J. Smith, D. Soltero, M. Tanner, L. White, and D. Rus. Building a distributed robot garden. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, St. Louis, MO, 2009.

[4] S. A. Deloach, W. H. Oyenan, and E. T. Matson. A capabilities-based model for adaptive organizations. *Autonomous Agents and Multi-Agent Systems*, 16(1):13–56, 2008.

[5] B. Gerkey and M. Mataric. Sold!: Auction methods for multi-robot coordination. *IEEE Transactions on Robotics and Automation, Special Issue on Multi-robot Systems*, 18(5):758–768, October 2002.

[6] R. Kota, N. Gibbins, and N. Jennings. Self-organising agent organisations. In *Proc. of the 8th Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS 2009)*, May 2009.

[7] M. W. Otte and G. Grudic. Extracting paths from fields built with linear interpolation. In *Proc. of the The 2009 IEEE/RSJ International Conference on Intelligent RObots and Systems (IROS 2009)*, 2009.

[8] P. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1st edition, 1996.

[9] Python Software Foundation. The python language reference - python v2.6.4 documentation, Oct. 2009.

[10] A. Saffiotti, M. Broxvall, M. Gritti, K. LeBlanc, R. Lundh, J. Rashid, B. Seo, and Y. Cho. The peis-ecology project: vision and results. In *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, Nice, France, 2008.

[11] G. Serugendo, M. Gleizes, and A. Karageorgos. Self-organization and emergence in multi-agent systems: an overview. *Informatica*, 30(1):45–54, 2006.